# Accelerating Concurrent Priority Scheduling Using Adaptive in-Hardware Task Distribution in Multicores

Mohsin Shan and Omer Khan

**Abstract**—Task parallel algorithms execute tasks in some programmer-specified order on shared-memory multicores. A Concurrent Priority Scheduler (CPS) selects high priority tasks, and schedules them among cores to exploit parallelism. The main objective of a CPS is to deliver work efficient algorithmic execution at low communication cost. Selecting high priority tasks among cores requires high synchronizations, but results in near-optimal work-efficiency. The communication cost can be reduced by processing tasks without a strict priority order, but it potentially degrades work-efficiency. This letter proposes HAPS, a novel CPS architecture that utilizes in-hardware core-to-core messages to accelerate task distribution among cores. Moreover, a set of dynamically tunable heuristics are proposed to co-optimize work-efficiency and communication in shared-memory multicores.

**Index Terms**—Task-parallelism, concurrent priority scheduler, shared memory, multicore, hardware messages

## 1 Introduction

Task-parallel algorithms are ubiquitously deployed in data processing and analytics domains, where an algorithm designer decomposes the algorithm into tasks comprising of several instructions. Tasks are generated at runtime, and pending tasks are scheduled dynamically on different cores for processing. Most task-parallel algorithms execute tasks in any order until convergence, thus exposing plentiful parallelism. However, many algorithms demonstrate faster convergence using *priority based scheduling*, where tasks with high priority are processed ahead of low priority tasks. Faster convergence is achieved because priority-based scheduling improves work-efficiency, which is the total amount of work performed by the parallel algorithm relative to the work performed by a sequential baseline.

Task-parallel frameworks use Concurrent Priority Schedulers (CPS) to implement priority-based scheduling of tasks on shared-memory multicore machines [2], [6], [7]. A good CPS design aims to schedule high priority tasks among the cores, while keeping synchronizations and CPS overheads low. State-of-the-art CPS designs use different methods to select high priority tasks. If a CPS optimizes to find high priority tasks, it leads to an increase in synchronization overheads. On the other hand, if a CPS aims to reduce synchronizations, it ends up picking tasks that are sub-optimal in their priority order, which results in redundant work. Thus, the objective of a CPS is to reduce synchronization costs, while high priority tasks are propagated to different cores for concurrent processing.

This paper makes a key observation that work-efficiency in a CPS is related to *priority drift*, which is the average difference of priority between the global highest priority task, and the tasks being processed by the cores at any instance. Maintaining minimal priority drift during execution improves work-efficiency. Minimal priority-drift can be achieved by fast propagation of high priority tasks among the cores. However, the additional burden of task communication and synchronizations must remain low for a CPS design to deliver superior performance. State-of-the-art CPS designs follow rigid task scheduling policies without dynamically adapting to priority drift among cores. There is a need for an adaptive task scheduling scheme that monitors priority drift at runtime, and adapts task scheduling to keep communication overheads low.

To demonstrate this idea, we adopt a state of the art CPS, Remote Enqueue, Local Dequeue (RELD) [9]. RELD implements a fine grain task distribution model that focuses on improving work-efficiency, while incurring relatively high synchronizations. It maintains a distributed array of concurrent priority queues, where each priority queue (PQ) is associated with a core. Each core dequeues a task from its PQ, executes it, and distributes the children tasks to other cores by selecting a remote core at random. Tasks are inserted in a PQ via atomic operations. The three main challenges with RELD are (1) the use of atomic operations on PQs for task transfers that incur high synchronizations, (2) aggressive task distribution that leads to increased on-chip network traffic, and (3) task distribution at a fine granularity that incurs PQ related computation overheads. We propose a hardware-assisted adaptive priority scheduler (HAPS) that aims to reduce priority drift among concurrent tasks, while keeping the synchronization and communication costs low. The key contributions of HAPS are outlined below.

(1) The inter-core transmission of tasks is accelerated using non-blocking asynchronous hardware messages that carry task information between cores at the hardware level. The use of hardware messages reduces synchronization costs as they bypass the requirement of atomic operations on PQs. Since the atomic operations are blocking in nature, they can potentially obstruct a core from performing its operations during task transfers. On the other hand, hardware messages are non-blocking and enable both sender and destination cores to continue without waiting to complete task movement. Hardware messages enable faster propagation of high priority tasks among the cores, thus reducing priority drift to improve work-efficiency.

(2) RELD aims to maintain high work-efficiency by continuously distributing tasks among cores. Uncontrolled task distribution among cores stresses the traffic injected in the on-chip network that must be considered alongside the reduction of redundant work. HAPS takes a novel approach by focusing on the reduction of priority drift between tasks being executed in different cores. A feedback-driven runtime heuristic is proposed that periodically exchanges latest task priority values among cores to estimate priority drift. It dynamically adjusts the rate of per-core task distribution, such that the priority drift is minimized among the cores. The heuristic keeps redundant work under control and focuses on reducing communication costs.

(3) Under RELD, each task destined for a remote core is first injected into the network and later enqueued in the destination core's priority queue. A large number of tasks significantly increase network traffic, and PQ enqueue/dequeue operations that hinder performance. We observe that it is advantageous to cluster and distribute tasks into containers with priorities in close proximity. When a task is processed, its children tasks are clustered into containers using their priorities, or proceed with tasks individually when multiple tasks do not share close proximity priorities. A container is highly input dependent and can range in size from a few or many tasks. However, each container is treated as a single entry for insertion and removal from a core's PQ. When a container with multiple tasks is dequeued from a PQ, its tasks are all processed in bulk before moving to the next dequeue operation. Selective task clustering in HAPS enables performance benefits by preventing unnecessary compute operations on PQ, and preserving locality of tasks by clustering them into a container.

• *The authors are with the Department of Electrical and Computer Engineering, University of Connecticut, Storrs, CT 06269 USA. E-mail: {mohsin.shan, khan}@uconn.edu.*

*Manuscript received 1 Oct. 2020; revised 13 Nov. 2020; accepted 8 Dec. 2020. Date of publication 0 . 0000; date of current version 0 . 0000.*
*(Corresponding author: Omer Khan.)*
*Digital Object Identifier no. 10.1109/LCA.2020.3045670*

The proposed CPS architecture mitigates synchronization and communication costs, while minimizing priority drift among concurrently executing tasks. HAPS, as well as state-of-the-art CPS schedulers, OBIM [6], PMOD [9], and RELD are evaluated using a real Tilera *TileGx-72* multicore machine that implements hardware-based directory cache coherence and explicit in-hardware messaging under the shared memory paradigm. For a set of representative task parallel algorithms, HAPS accelerates RELD by 2.5×, improves performance over OBIM by 1.9×, and PMOD by 1.6×.

## 2 Related Work

Recent work conducted an empirical performance analysis of CPS designs for modern shared-memory multicores [9]. It observes that the performance of Galois' OBIM [6] CPS is remarkable, followed by RELD. OBIM implements a task distribution model that merges tasks with priorities in close range into a single priority, distributed, and unordered bag. Each core first searches for the highest priority bag and then process all the tasks in that bag. Following this approach, OBIM schedules a few bags instead of many tasks, which leads to reduced synchronizations. Moreover, processing tasks clustered in a bag takes advantage of task locality. However, this approach is helpful when there is a sufficient number of tasks in the bags. Otherwise, the cores need to perform bag search operations repeatedly, thus incurring synchronization costs. To avoid these costs, OBIM creates large bags by increasing the range of merged priorities, which reduces communication costs but leads to loss of priority order and hence results in redundant work. To solve this challenge, a recent work PMOD [9] dynamically optimizes the utilization of bags at runtime to improve performance of OBIM. HAPS takes a fine grain task distribution approach to improve work-efficiency by focusing on minimizing priority drift across concurrently executing tasks. It overcomes the relatively high communication costs of RELD to deliver superior performance compared to both RELD and PMOD.

Prior works, such as Swarm [5] and Minnow [10] also aim to improve work-efficiency and/or overcome communication costs in concurrent priority schedulers. However, they all require invasive hardware modifications to the multicore processor. Swarm adopts RELD to distribute tasks among cores, but requires multiple per-core task, order and commit queues (totaling 10s of $KB$ hardware) with complex parallel lookup capabilities. It preserves the task ordering among cores, while introducing hardware to address the communication challenge. On the other hand, Minnow builds on OBIM, and introduces a per-core hardware accelerator to offload task (worklist) scheduling, and task pre-fetching from shared memory to improve communication costs. However, it does not directly address the task priority drift challenge of the OBIM scheduler.

Prior work ADM [8] uses hardware messages for task transfers. However, it was implemented primarily for message passing schedulers, and was not evaluated in the context of CPS designs. HAPS optimizes priority drift, which is correlated with communication costs. Hence, fast transfer of tasks using in-hardware messages is shown to improve priority drift that results in work-efficient execution.

Prior works have also adopted unordered algorithms, where the task priority scheduling is ignored to maximize concurrency. These algorithms suffer from redundant work that can be mitigated by utilizing a large number of cores in modern GPU and supercomputing machines. However, recent works have quantified that concurrent priority schedulers that tradeoff between work-efficiency and task priority drift deliver superior performance in a shared memory setup [6], [9].

## 3 HAPS Concurrent Priority Scheduler

Prior CPS approaches lean towards optimizing opposite sides of the work-efficiency and communication tradeoff. OBIM and its optimized PMOD variant aim to reduce communication costs by relaxing task orderings. On the other hand, RELD focuses on continuous distribution of tasks to spread high priority tasks among cores, but incurs high communication costs. HAPS hypothesizes that work-efficiency and communication cost are both crucial for overall performance. Therefore, it proposes in-hardware and dynamically adaptive task distributions that minimize priority drift among cores. Fig. 1 shows the per-core view of the proposed HAPS architecture, which is built on top of the baseline RELD (shown in shaded arrows/boxes).

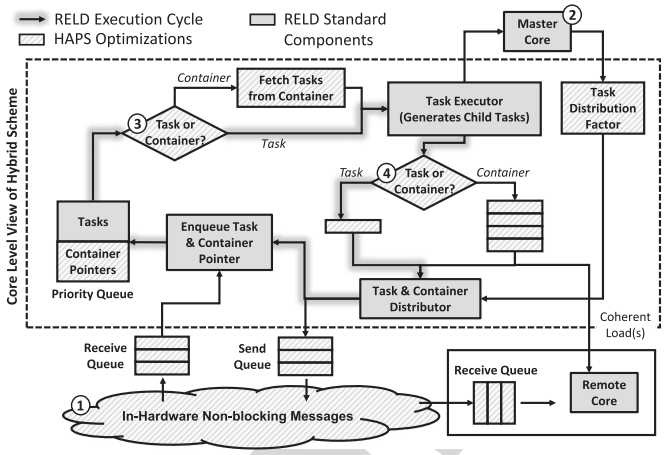


Fig. 1. Hardware and software components of the HAPS architecture.

### 3.1 Hardware Support for Accelerating Task Transfers

In RELD, each task transfer between cores is achieved using atomic enqueue and dequeue operations on the per-core priority queue (PQ). These atomic operations are blocking and stall both the sender and destination cores. Moreover, these operations are time-consuming as the PQ needs to re-balance based on new task priorities, resulting in performance loss. We illustrate that the task transfer mechanism can be accelerated using asynchronous non-blocking in-hardware messages [1] (c.f. Fig. 1:①). (1) The sender core moves forward without having to wait for the completion of a task transfer. (2) Asynchronous communication enables the remote core to process incoming messages at its convenience. (3) Hardware transfer of task results in low latency and reduced metadata computations at both sender and receiver cores. These benefits allow fast and high bandwidth propagation of high priority tasks among cores, resulting in decreased priority drift and faster algorithmic convergence. However, in the presence of finite sized per-core send and receive queues, application level deadlocks must be prevented. Each sender core maintains a dedicated flag (boolean) for each destination core. When a sender core chooses a destination core to send a task, it checks the destination core's flag. If the flag is not set, the sender core sends the task and atomically increments the flag. However, if the flag is set, the sender core picks another randomly selected remote core to send the task. In case no remote core can accept the task, it is added to the local PQ of the core generating the task. The flag is atomically cleared by the destination core when the associated task is removed from the hardware receive queue. Note that the tasks are moved from the hardware receive queue to the software managed PQ with high priority. The software-based distributed flow control mechanism ensures forward progress in HAPS.

The messaging scheme can also be built using software-only shared memory support [3]. A software per-core shared buffer implements communication between cores. Each buffer slot contains a flag and a place holder for the data to be sent. A requesting core atomically increments the write pointer of the corresponding buffer 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. The receiving core

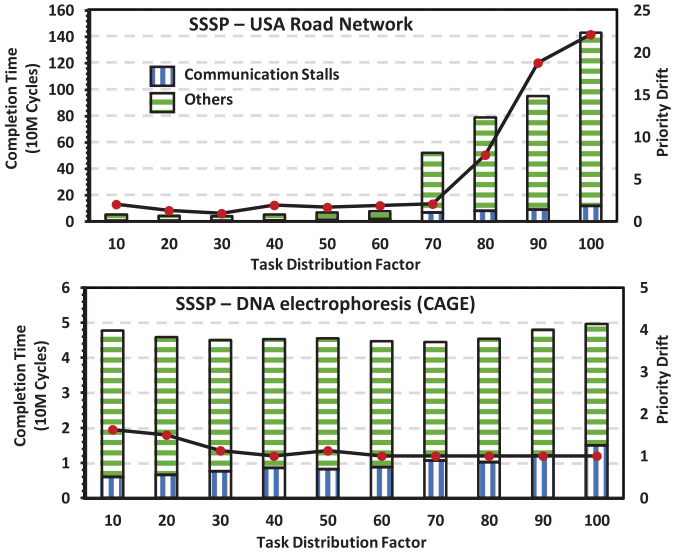


Fig. 2. Completion time breakdown and priority drift of RELD at different task distribution factors using 72 threads.

removes valid tasks from its buffer, and put them into its PQ for further processing. Our preliminary results show that hardware messages are faster than the software-only counterparts.

### 3.2 Adaptive Task Distribution to Manage Priority Drift

In RELD, the cores always distribute tasks generated at runtime to randomly selected cores. The aim is to reduce priority drift since the underlying assumption is that when a core executes a high priority task, it is probable that the priority of the children tasks will be in close proximity of high priority. However, aggressive distribution by RELD overlooks scenarios where children tasks do not always exhibit priorities closer to the highest priority task in the system. Moreover, it is also possible that PQ utilization remains low during execution, leading to overall divergence of priority drift among cores when tasks are continuously distributed. Even when priority drift does not get worse, the traffic on the on-chip network may increase due to unwarranted task distributions among cores.

To quantify the impact of task distribution on performance, Fig. 2 shows the breakdown of parallel completion times and priority drift for a representative single source shortest path benchmark, SSSP for two graph inputs. A metric, *task distribution factor (TDF)* is introduced that is defined as "the ratio of remote enqueue operations and the total number of enqueues performed by a core." The baseline RELD scheme utilizes 100 percent TDF, where each task is distributed to a randomly picked destination core. However, TDF is varied from 10 to 100 percent to quantify the impact of choosing a variable number of tasks for distribution to remote or local core's PQ. For the USA road network, the priority drift increases with an increase in TDF. In this graph, a small number of children tasks (1.2 on average) are generated with diverging priorities. Thus, aggressive TDF results in slow propagation of high priority tasks among cores due to communication costs, resulting in the priority drift to increase. Consequently, the completion time dramatically increases past 60 percent TDF due to increased work-inefficiency and communication stalls. For the CAGE graph, the generated children tasks (16 on average) end up in close proximity to high priority. Thus, initially when the TDF keeps most tasks in local core's PQ, the high priority tasks are not well distributed among cores. However, as TDF increases, priority drift decreases, but saturates at ~40 percent. Completion time is minimum at this saturation TDF point since communication stalls increase due to unnecessary network traffic past this point. It is clear that TDF needs to be adjusted at runtime for near-optimal performance.

HAPS proposes a heuristic that quantifies priority drift among cores at different time intervals. The heuristic uses an initial value of 50 percent TDF. After processing a fixed number of tasks (2K used in this paper using empirical data), each core sends the priority of its latest task processed to a master core (c.f. Fig. 1:②). The master core receives priorities from all cores, and calculates the priority drift using the average pairwise differences. The priority drift of each sampling period serves as a reference for the next sampling period. The heuristic also tracks whether TDF was increased or decreased in the previous sampling period. For a given sampling period, the priority drifts are compared for the latest and previous sampling periods to compute whether the drift is increasing or decreasing. An increase in difference implies priority drift getting worse. However, if TDF was increased in previous sample and latest drift gets worse, then TDF is decreased. But the TDF is increased for current sample, if previous sample's TDF was decreased. Similarly, a decrease in difference implies priority drift is improving. Regardless of TDF was increased or decreased for previous sample, the current sample's TDF is now decreased. TDF is increased or decreased in intervals of 10 percent. The heuristic's goal is to keep the priority drift low while ensuring unnecessary traffic is not injected into the on-chip network. In future, we plan to incorporate deeper history tracking and gradient descent in the heuristic for better prediction of task distribution factor. We also plan to compare the heuristic to a dynamic oracle that selects the best performing TDF at each interval.

### 3.3 Adaptive Bulk Processing of Tasks Using Containers

In RELD, each task is processed individually by inserting and removing it from its respective PQ. Each PQ maintains the highest priority elements at the top of the queue so that dequeue is an O(1) operation. However, the addition and removal of tasks from the PQ results in re-ordering operations required to maintain the priority constraint. This process is compute-intensive, therefore, it is beneficial to reduce these operations on the PQ. Task containers are proposed to tackle this challenge. The container constitutes the payload for its tasks and metadata to track the container identifier and its priority. The container metadata is enqueued in the PQ of a core (local or remote), while its payload is stored either at the sender core, or at the destination core. The PQ at the destination core dequeues the container metadata when it becomes the highest priority at that core. The storage of container payload offers two implementation options. (1) Store the container payload at the sender core and use coherent loads to retrieve data when the container is dequeued from the PQ. (2) Communicate container payload alongside its metadata and store payload and metadata at the destination core's PQ. Option 1 is chosen in this paper, but this decision may offer computation and communication tradeoffs that we plan to investigate in our future work. The advantage is that a container prevents unnecessary PQ operations by limiting the number of insertions and removals from the PQ. Moreover, coherent loads with option 1 efficiently transfer large data payload of a container by exploiting its inherent data locality.

The use of containers is beneficial only when there are a sufficient number of tasks in the container. If a container only has a few tasks, then its overheads can overshadow the benefits. OBIM and PMOD both suffer from this problem since they always create containers (or bags) that may be under-utilized. HAPS proposes a heuristic to address this challenge, and selects a task or a container for insertion into the PQ. After processing a parent task or container, if there are at least three children tasks for a given priority, they are bundled together in a container. Otherwise, they are left alone individually. The tasks and containers are then scheduled for insertion into a local or remote PQ (c.f. Fig. 1:④). This approach avoids unnecessary computations associated with handling individual

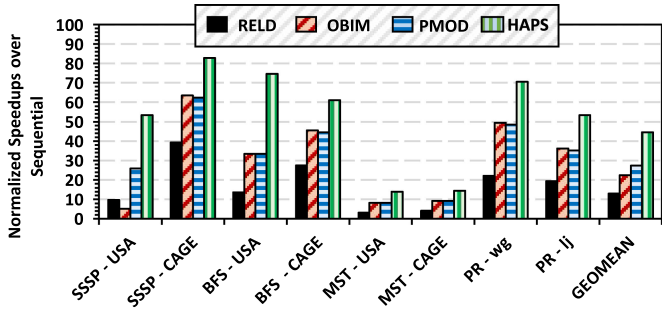


Fig. 3. Performance of RELD, OBIM, PMOD, and HAPS normalized to the optimized sequential implementation.

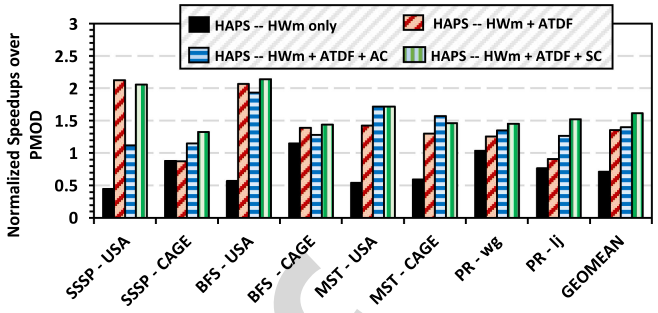


Fig. 4. Performance evaluation of different HAPS's configurations.

tasks as containers, i.e., insert container metadata in a PQ, and then separately retrieve its payload on dequeue.

## 4 Methodology

The evaluation is performed on a real multicore machine, *Tilera Tile-Gx72™*. The processor chip integrates 72 tiles, where each tile consists of a 64-bit multi-issue pipelined core, private level-1 (32 *KB*) data and instruction caches, and a 256 *KB* slice of the shared last-level cache (totaling 18 *MB* on chip). The tiles are interconnected using 2D mesh networks that support hardware capability to implement in-hardware messages between cores, as well as directory-based cache coherence for shared memory operations. Tilera's Tiled Multicore Components (TMC) pthreads library is used to port the RELD, OBIM, PMOD, and HAPS CPS designs. OBIM and PMOD are tuned for processors that have a small number of cores per socket. To configure OBIM on Tilera multicore, the 72 tiles are logically distributed with a variable number of tiles per socket, and best performing configuration of 9-sockets with 8-tiles each is used for evaluation.

Four task-parallel graph algorithms, SSSP, BFS, MST, and Pagerank are chosen from PMOD [9] for evaluation. SSSP uses delta-stepping algorithm to compute the shortest distance from a source vertex to every vertex in a graph. BFS is a special case of SSSP, where the weight of each edge is 1. MST uses Boruvka's algorithm to find a minimum spanning tree consisting of all the vertices with minimum total edge weight. Pagerank uses a push-pull algorithm, where each node's rank is calculated by computing over its incoming edges, and then propagating the changes to outgoing neighbors. As in PMOD, real world graph inputs are picked for evaluation, i.e., USA road network, DNA electrophoresis (CAGE), and two social network graphs, live-journal (lj) and web-google (wg).

## 5 Evaluation

Fig. 3 shows the performance evaluation of OMIB's optimized PMOD, RELD, and HAPS, normalized to optimized sequential implementations. PMOD shows performance scaling over RELD for all benchmarks since RELD suffers from significant communication costs and work-inefficiency. HAPS successfully adapts the task distribution factor (TDF) and distributes tasks or containers among cores to optimize priority drift, resulting in a work efficient and communication aware CPS design. Overall, HAPS outperforms OBIM by 1.9× and PMOD by 1.6×. Similar to PMOD paper, MST is observed to show low performance scaling.

Fig. 4 introduces the proposed HAPS optimizations and normalizes their performance to the PMOD baseline. HAPS – *HWm only* adds hardware messaging capability to the RELD baseline. The benefits here arise from mitigating the overheads of synchronizations in RELD due to atomic operations on PQs. However, it uses a static 100 percent TDF, and thus suffers from work-inefficiency due to unmanaged priority drift among cores. The HAPS – *HWm + ATDF* utilizes the adaptive TDF heuristic proposed in Section 3.2 to obtain work-efficient execution. The USA graph results in high priority drifts as TDF increases, as also seen in Fig. 2. This results in poor work-efficiency at 100 percent TDF, which significantly improves when a near-optimal TDF is used. However, for CAGE graph the priority drift remains low as TDF varies, thus performance benefits mainly stem from reduced communication costs associated with the right selection of TDF at runtime. Overall, HAPS – *HWm + ATDF* outperforms HAPS – *HWm only* by 1.6× and PMOD by 1.35×, but it still has high CPS overheads as it stores individual tasks in a PQ. If tasks belonging to the same priority are bundled and managed as a container of tasks, the PQ overheads can be dramatically reduced. Moreover, the communication overheads of individual tasks can be mitigated. The HAPS – *HWm + ATDF + AC* always creates task containers instead of individual tasks. This configuration only improves performance when multiple tasks with close priority are generated, otherwise the overheads of container formation result in overheads. For example, in SSSP-USA the graph mostly generates tasks with random priorities. Hence, it always incurs the computational overheads of containers. HAPS – *HWm + ATDF + SC* adopts the proposed heuristic for selective clustering from Section 3.3, where containers are formed at runtime when multiple tasks with same priority are generated. Otherwise, individual tasks are inserted in the PQ.

The PMOD baseline is modified with hardware messages to accelerate synchronizations [4], as shown in Fig. 5. Here, instead of using atomic operations, the shared *Global Map* data structure is updated by serializing all it's requests using in-hardware messages. The results show that the use of in-hardware messages improve performance by ∼2 percent over the baseline PMOD that uses atomic operations. However, as discussed earlier, HAPS significantly benefits from using in-hardware messages for task transfers. To justify this hardware overhead, Fig. 5 shows the performance evaluation by replacing in-hardware messages in HAPS with a shared-memory software-based messaging scheme [3], as well as using traditional atomic operations. Atomic locks on priority queues show the least performance since they block both the sender and remote cores from making progress, while the priority queue balances itself during enqueue and dequeue operations. Both hardware and software mes-

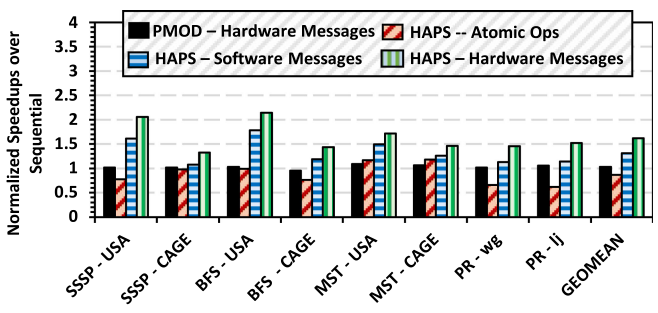


Fig. 5. Performance evaluation of HAPS with different task transfer mechanisms. PMOD baseline that uses atomic operations is also compared with PMOD using hardware messages.

saging-based schemes perform better than atomic locks since they do not stall the cores, and result in faster propagation of high-quality tasks across cores. However, hardware messages perform 1.25× better than the software counterpart. It is noteworthy that software messaging based HAPS also outperforms the PMOD baseline by 1.3×.

## 6 Conclusion

This paper proposes a CPS architecture that ensures work-efficient execution of task parallel algorithms on shared-memory multicores. Current CPS designs follow rigid strategies for selecting high priority tasks without monitoring and adapting to optimize work-efficiency and communication overheads. HAPS proposes in-hardware and dynamically adaptive task distributions that minimize priority drift among cores. Consequently, it optimizes work-efficiency and communication costs, and outperforms state-of-the-art CPS PMOD by 1.6×, and OBIM by 1.9× for the evaluated benchmarks.

## Acknowledgments

This work was supported in part by the National Science Foundation under Grant CNS-1718481. This research was also supported in part by the Semiconductor Research Corporation (SRC).

## References

[1] M. Ahmad, H. Dogan, J. A. Joao, and O. Khan, "In-hardware moving compute to data model to accelerate thread synchronization on large multicores," *IEEE Micro*, vol. 40, no. 1, pp. 83–92, Jan./Feb. 2020.

[2] L. Dhulipala, G. Blelloch, and J. Shun, " Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proc. ACM Symp. Parallelism Algorithms Archit.*, 2017, pp. 293–304.

[3] 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, no. 1, pp. 4:1–4:27, Feb. 2019.

[4] 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 *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 254–264.

[5] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, " A scalable archit. for ordered parallelism," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2015, pp. 228–241.

[6] D. Nguyen, A. Lenharth, and K. Pingali, " A lightweight infrastructure for graph analytics," in *Proc. ACM Symp. Operating Syst. Princ.*, 2013, pp. 456–471.

[7] H. Rihani, P. Sanders, and R. Dementiev, " Multiqueues: Simple relaxed concurrent priority queues," in *Proc. ACM Symp. Parallelism Algorithms Archit.*, 2015, pp. 80–82.

[8] D. Sanchez, R. M. Yoo, and C. Kozyrakis, " Flexible architectural support for fine-grain scheduling," in *Proc. 15th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2010, pp. 311–322.

[9] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, " Understanding priority-based scheduling of graph algorithms on a shared-memory platform," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, pp. 46:1–46:14.

[10] D. Zhang, X. Ma, M. Thomson, and D. Chiou, "Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching," *ACM SIGPLAN Notice*, vol. 53, no. 2, pp. 593–607, Mar. 2018.