Brief Announcement: Work Stealing through Partial Asynchronous Delegation

Jiawei Wang
Huawei Dresden Research Center
Technische Universität Dresden
Germany
jiawei.wang@huawei.com

Yutao Liu
Huawei Dresden Research Center
Germany
liuyutao2@huawei.com

Ming Fu*
Huawei Central Software Institute
China
ming.fu@huawei.com

Hermann Härtig
Technische Universität Dresden
Germany
hermann.haertig@tu-dresden.de

Haibo Chen
Huawei Central Software Institute
Shanghai Jiao Tong University
China
haibochen@sjtu.edu.cn

ABSTRACT

Work stealing is a well-established technique in multi-core systems that aims to improve load balancing and task scheduling efficiency. Each processing unit maintains its own task queue, and when idle, it steals tasks from other units. Traditional work-stealing approaches often face performance bottlenecks due to costly synchronization primitives and contention arising from concurrent access by both the queue owner and thieves. The state-of-the-art solution addresses these issues through coarse-grained synchronization; however, it restricts stealing in specific scenarios, thereby limiting parallelism.

We introduce PadWS, a partial and asynchronous delegated work-stealing strategy. PadWS employs a block-based design in which, under common cases, the queue owner and thieves work on separate blocks, reducing metadata contention. Delegation is partially enabled only for the block in which the owner is located, allowing thieves to steal from that block—an approach that deviates from the current block-based method. Additionally, our delegation strategy is asynchronous, which removes the need for thieves to spin-wait after sending a request.

CCS CONCEPTS

• **Computing methodologies → Concurrent algorithms**.

KEYWORDS

parallel processing; scheduling; work stealing; delegation

ACM Reference Format:
Jiawei Wang, Yutao Liu, Ming Fu, Hermann Härtig, and Haibo Chen. 2024. Brief Announcement: Work Stealing through Partial Asynchronous Delegation. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3626183.3660261

*Corresponding author.

SPAA '24, June 17–21, 2024, Nantes, France

ACM ISBN 979-8-4007-0416-1/24/06.
https://doi.org/10.1145/3626183.3660261

1 INTRODUCTION

Work stealing [3] is a widely employed technique in various multi-core systems. In this approach, each processing unit (e.g., thread or core) has its own queue and serves as the owner, responsible for pushing and popping tasks from it. When a processing unit completes its own tasks and becomes idle, it takes on the role of a thief, attempting to steal tasks from other queues.

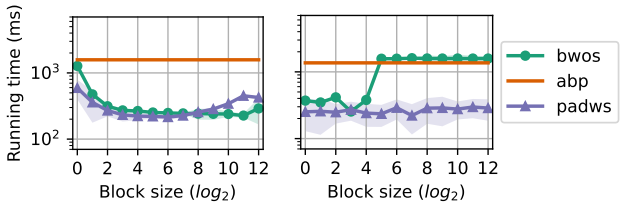
1.1 ABP Work Stealing

The ABP work-stealing algorithm [2] is widely recognized and extensively used in various industrial contexts. However, it requires a significant number of synchronization primitives (marked as **P1**), primarily memory barriers [7, 9], to ensure the correctness of concurrent operations performed by the queue owner and thieves. This requirement can create a bottleneck, impacting the overall end-to-end performance, especially when dealing with small tasks [5, 11]. For instance, in garbage collection components.

1.2 Block-based Work Stealing

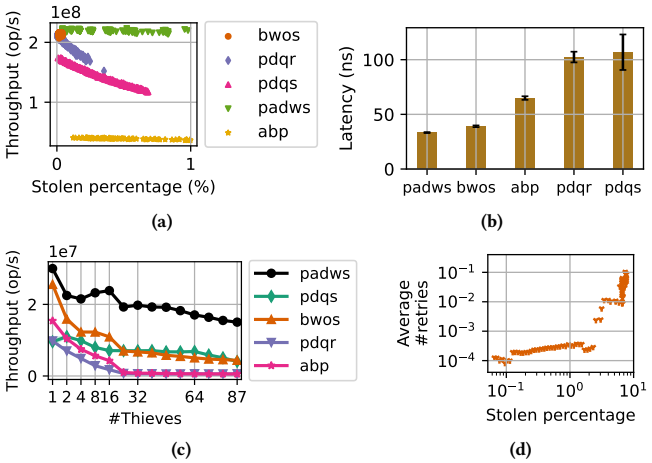
Numerous research studies [1, 4, 5, 8, 11] have focused on optimizing or proposing alternatives to reduce memory barriers in work stealing algorithms. One notable work is the block-based work stealing (BWoS) [11], which partitions the queue's data and metadata into multiple blocks. Unlike the fine-grained synchronization in ABP, where the owner and thieves synchronize for every task, BWoS uses a coarse-grained synchronization strategy. Thieves are prevented from stealing tasks from the block where the owner is located, granting sole access to the owner and eliminating the need for synchronization with thieves on that specific block. Synchronization is only required when operations cross between blocks. This design allows BWoS to achieve significant performance improvements for various scenarios.

However, in specific scenarios such as parallel task processing systems [6] employing a fork-join model, configuring each block in BWoS to be too large can result in a prolonged period where an owner, who initially possesses the root task, takes a significant



(a) fat tree workload (b) thin tree workload

Figure 1: *Running times for PadWS, BWoS, and ABP algorithms across various tree traversal workloads, where lower values indicate better performance. In our runtime setup, we assigned 8 queues, each to a dedicated thread. Initially, the root task was placed in the first queue, with all other queues left empty.*



(a) (b) (c) (d)

Figure 2: *(a) Owner's throughput under various stolen percentages. (b) Average latency of steal operations. (c) Total throughput of steal operations with various numbers of thieves. (d) Average number of retries of steal operations under different stolen percentages for* adws. *All experiments were conducted on an x86 machine with 88 hyper-threads.*

amount of time to fill the first block of its queue. During this time, other threads are idle but unable to steal tasks, reducing parallelism to just one core. Reducing the block size to mitigate this issue introduces significant synchronization overhead due to frequent and costly block advancement operations.

To support this assertion, we conducted a benchmark comparing the performance of tree traversals with two different workloads: the fat tree and the thin tree, as shown in Fig. 1. In the fat tree, each node has numerous children but maintains a shallow depth, leading to significant fluctuations in task numbers in the queue during traversal. Consequently, for BWoS with a smaller block size, frequent block advancements occur, resulting in considerably longer running times. Conversely, the thin tree comprises nodes with fewer children but with greater depth, resulting in minimal fluctuations in task numbers. We observed that for BWoS, as the block size increases, the runtime experiences rapid escalation, consistent with the previously claimed limitation. This emphasizes the trade-off in block size selection within BWoS (marked as **P2**). Larger blocks may lead to core under-utilization in specific scenarios, while smaller blocks can increase synchronization overhead. Given that real-world scenarios often involve a mix of these workloads, there is no universally suitable configuration when employing BWoS.

1.3 Delegated Work Stealing

Another alternative for implementing work stealing is delegation [1], wherein the thief takes on the role of a client and sends a message to the queue owner, requesting a task. The owner, in turn, acts as a server and responds to the thief's request by providing a task. Delegation reduces the number of memory barriers for the owner, since barriers are only needed when there are steal requests to transfer tasks to thieves. Nevertheless, state-of-the-art delegation approaches still encounter the following issues.

• The steal operation may occur frequently, resulting in a scenario where the owner and thieves frequently access the same communication variables (marked as **P3**), causing significant impacts on performance due to the contention [11]. To quantify the impact, as shown in Fig. 2a, we conduct a benchmark on the state-of-the-art delegation work stealing algorithms, namely the receiver-initiated and sender-initiated private deques work stealing algorithms [1] (pdqr and pdqs). Initially, we measure the owner's throughput in a sequential setup, where an owner pushes and pops data from its queue without any tasks being stolen. Next, we introduce a thief performing a steal operation and adjust the frequency of these operations to create different steal percentages. As shown in Fig. 2a, the throughput of pdqr and pdqs drops significantly when steal operations occur.

• After sending a request to the owner, the thief enters a spinning state, waiting for a response (marked as **P4**). This spinning can cause additional latency in the steal operation, especially when the owner does not respond promptly (e.g., due to preemption, pushing/popping tasks, or running tasks). To evaluate this, we measured the average latency of steal operations, as shown in Fig. 2b. The results suggest that pdqr and pdqs have the highest latency, even surpassing that of ABP, which employs extensive memory barriers in its steal operation.

• When processes steal requests, the owner is burdened with delegated workloads (e.g., copying tasks, setting/resetting communication variables), instead of allowing them to be executed in parallel by multiple thieves. This centralized and sequential ways of processing steal requests (marked as **P5**) have a considerable impact on thieves scalability. To support this claim, we increase the number of thieves and measure their throughput, as shown in Fig. 2c. The experiment revealed that for pdqr and pdqs, throughput does not scale effectively as the number of thieves increases.

2 OUR CONTRIBUTIONS

We introduce PadWS, a novel partial and asynchronous delegated work-stealing approach to address the performance challenges and scenario limitations found in current state-of-the-art methods. We acknowledge that delegation effectively mitigates the issue of barrier overhead (P1). In subsequent sections, we will present two strategies, namely asynchronous delegation and partial delegation, to address the aforementioned issues found in existing methods.

2.1 Asynchronous Delegation

We propose asynchronous delegation, in which a steal operation doesn't request a task for itself, but for the following steal operation instead. Initially, a steal operation retrieves the task requested by

the previous steal (if available). Then, the thief continues without waiting for a response from the owner (P4). This optimization eliminates nearly all unnecessary waiting time for thieves.

In the rare event that the steal operation fails to retrieve the task because the owner hasn't yet prepared it, the thief retries. To determine whether this failure is indeed rare, we conducted an experiment to measure the expected retry time for steal operations, as shown in Fig. 2d. To confirm that the low number of retries is due to asynchronous delegation and not by partial delegation, we created an algorithm that uses asynchronous delegation but omits partial delegation (adws). The results demonstrate that the expected retry time is remarkably low, indicating that asynchronous delegation is effective, as most steal operations do not require retries. Additionally, Fig. 2b shows that PadWS has the lowest average latency for steal operations among all tested algorithms, with approximately 0.32x latency compared to pdqr and pdqs.

2.2 Partial Delegation

Partial delegation encompasses two aspects: segmenting the queue and fragmenting the operation.

2.2.1 Segmenting the Queue. The block-based approach [10, 11] has recently emerged as an effective solution for reducing metadata contention within queues. We adopt this approach to facilitate partial delegation. The queue is divided into blocks [1], where delegation is only applied to a portion of the data located within the owner's block, rather than the entire queue. Once the owner completes pushing/popping tasks within a block, the owner advances to the corresponding next block. Subsequently, the owner ends the delegation for the current block and begins it for the new one.

In typical scenarios where the owner and thieves operate on different blocks, they access metadata from separate blocks, thereby reducing potential contention for shared variables, which could impact the owner's performance (P3). Additionally, in the block where delegation ends, thieves can steal tasks directly without the owner's involvement. This further reduces memory barrier overhead on the owner's side caused by delegation (P1) compared to state-of-the-art delegation methods. Evidence supporting this can be found in Fig. 2a, which shows that increasing the percentage of stolen tasks has a negligible effect on the owner's throughput—similar to the behavior observed in the other block-based approach, BWoS.

Moreover, when the owner and thieves are in the same block, thieves can still acquire tasks via asynchronous delegation. This approach overcomes the limitations faced in BWoS, where thieves were unable to steal from the owner's block (P2). Consequently, as illustrated in Fig. 1, by selecting an appropriate block size that isn't at the extreme ends, PadWS can consistently deliver improved performance, achieving speedups of up to 4.7x compared to ABP.

2.2.2 Fragmenting the Operation. State-of-the-art approaches delegate the entire steal operation to the owner. In contrast, our approach delegates only a portion of it. Specifically, we delegate task ownership acquisition to eliminate the costly sequentially consistent barriers required for mutual exclusion by the owner and thieves, like those in ABP. Upon receiving a steal request, the owner notifies the thief solely of the position from which to steal, rather than the actual task. Afterward, the thief accesses the queue, copies the task, and signals that the copying is complete.

This design offers several significant benefits, addressing P5. First, it reduces the owner's workload, which can improve performance. Second, it allows the owner to respond quickly to subsequent steal operations, reducing the chance of spinning steal operations. Finally, the parallel execution of task copying and the indication of copy completion on the thieves' side enhances the scalability of concurrent steal operations. Consequently, as illustrated in Fig. 2c, PadWS demonstrates the best scalability for steal operations among all tested algorithms. For instance, when there were 87 thieves, PadWS delivered 3.4x performance improvements compared to the second-best one.

[1] We created a heuristic function to determine the block size, defined as $2^{\log_2(\text{capacity})/2}$, which offers a practical performance solution.

3 CONCLUSION

We presents PadWS, a novel design that leverages partial and asynchronous delegation to overcome performance challenges and scenario limitations observed in current state-of-the-art approaches.

REFERENCES

[1] Umut A Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 219–228.

[2] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. 2001. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems* 34, 2 (2001), 115–144.

[3] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.

[4] Rafael Custódio, Hervé Paulino, and Guilherme Rito. 2023. Efficient Synchronization-Light Work Stealing. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*. 39–49.

[5] Michihiro Horie, Hiroshi Horii, Kazunori Ogata, and Tamiya Onodera. 2018. Balanced double queues for GC work-stealing on weak memory models. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*. 109–119.

[6] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems* 33, 6 (2022), 1303–1320. https://doi.org/10.1109/TPDS.2021.3104255

[7] Nian Liu, Binyu Zang, and Haibo Chen. 2020. No Barrier in the Road: A Comprehensive Study and Optimization of ARM Barriers. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 348–361. https://doi.org/10.1145/3332466.3374535

[8] Maged M Michael, Martin T Vechev, and Vijay A Saraswat. 2009. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 45–54.

[9] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 530–545. https://doi.org/10.1145/3445814.3446748

[10] Jiawei Wang, Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, Jitang Lei, Geng Chen, Hermann Härtig, and Haibo Chen. 2022. BBQ: A Block-based Bounded Queue for Exchanging Data and Profiling. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 249–262. https://www.usenix.org/conference/atc22/presentation/wang-jiawei

[11] Jiawei Wang, Bohdan Trach, Ming Fu, Diogo Behrens, Jonathan Schwender, Yutao Liu, Jitang Lei, Viktor Vafeiadis, Hermann Härtig, and Haibo Chen. 2023. BWoS: Formally Verified Block-based Work Stealing for Parallel Processing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 833–850. https://www.usenix.org/conference/osdi23/presentation/wang-jiawei