Optimizing Work Stealing Communication with Structured Atomic Operations

Hannah Cartier
Rhodes College
Memphis, TN, USA
carhr-21@rhodes.edu

James Dinan
NVIDIA Corporation
Westford, MA, USA
jdinan@nvidia.com

D. Brian Larkins
Rhodes College
Memphis, TN, USA
larkinsb@rhodes.edu

ABSTRACT

Applications that rely on sparse or irregular data are often challenging to scale on modern distributed-memory systems. As a result, these systems typically require continuous load balancing in order to maintain efficiency. *Work stealing* is a common technique to remedy imbalance. In this work we present a strategy for work stealing that reduces the amount of communication required for a steal operation by half. We show that in exchange for a small amount of additional complexity to manage the local queue state we can combine both discovering and claiming work into a single step. Conventionally, work stealing uses a two step process of discovering work and then claiming it. Our system, SWS, provides a mechanism where both processes are performed in a singular communication without the need for multiple synchronization messages. This reduction in communication is possible with the novel application of atomic operations that manipulate a compact representation of task queue metadata. We demonstrate the effectiveness of this strategy using known benchmarks for testing dynamic load balancing systems and for performing unbalanced tree searches. Our results show the reduction in communication reduces task acquisition time and steal time, which in turn improves overall performance on sparse computations.

CCS CONCEPTS

• **Computing methodologies → Parallel computing methodologies**.

KEYWORDS

Work stealing, PGAS, dynamic load balancing, atomic communications

ACM Reference Format:
Hannah Cartier, James Dinan, and D. Brian Larkins. 2021. Optimizing Work Stealing Communication with Structured Atomic Operations. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472522

1 INTRODUCTION

A large number of parallel problems produce irregular computations that can be challenging to execute without resulting in load imbalance. The sources of this imbalance can be due to non-uniform data structures, inherent irregularity in the algorithm, or variation in the degree of available parallelism throughout a computation. Traditional static task decomposition and scheduling approaches do not map well to these classes of problems. Many irregular problems are often most naturally expressed dynamically, such as with recursive spatial decompositions or similar methods to handle inherent sparsity. Dynamic load balancing systems can provide abstractions that work well with irregular problems in both task decomposition and mapping variable workloads onto available compute resources – yielding both good performance and scalability.

Distributed dynamic load balancing systems can be used to remedy many of the deficiencies of static load balancers. The Cilk [14] programming language provided a load balancing system that requires idle processes to steal tasks from target processes chosen at random. Previous work has shown that this method is optimal for many classes of problems and has efficient space bounds [5, 18]

The one-sided read, write, and atomic operations supported by remote direct memory access (RDMA) systems are handled by specialized hardware in the high-speed network fabrics of high performance computing (HPC) systems. This low-level support enables the efficient realization of a Partitioned Global Address Space (PGAS) in systems such as the MPI Remote Memory Access (RMA) interface [24], Unified Parallel C [34], and more recently in systems such as OpenSHMEM [28]. PGAS models provide both a global view of memory and a set of one-sided communication operations for accessing non-local data. Implementing work stealing dynamic load balancing with PGAS allows systems to take advantage of both a global view of memory as well as one-sided access operations. Hardware supported RDMA permits a work stealing system to both search for and acquire work without interrupting active computations at the target.

Scioto (Shared Collection of Task Objects) [10, 11] is a PGAS-based task parallel programming model whose work stealing system utilizes RDMA capabilities via the Aggregate Remote Memory Copy Interface (ARMCI) [25]. In this work, we review the design of this dynamic load-balancing system and consider improvements made possible by recent work in high performance networking.

Conventional work-stealing systems must perform a sequence of one-sided RDMA communications in order to identify available work, claim it, transfer stolen tasks to the local work queue, and record steal completion in the work queue state on the target process. This approach has been shown to have good scalability and happens without requiring the active participation of the target process. However, this multi-step communication sequence causes high steal latencies. This work considers the novel use of atomic operations to greatly reduce the number of communication messages needed for a steal operation. In particular, we consider the OpenSHMEM programming model, a modern framework that provides both a partitioned global address space and a set of communications that operate on it. OpenSHMEM is built on lower-level communications frameworks such as Unified Communication-X (UCX) [33] or libfabric [27] that provide optimized communication primitives which take advantage of available hardware acceleration.

In contrast with conventional approaches, our Structured-atomic Work Stealing system, (*SWS*), adapts the work stealing communication process with atomic operations, yielding improved performance and reduced overhead. Prior work reduced communication in a work stealing framework and relied on features within the Portals 4 network programming interface [2] — a system to prototype next-generation interconnect design. Leveraging Portals messaging abstractions reduced communications for steal transactions to a single network round-trip [21]. The design of this system inspired the development of new techniques for reducing steal communication latency using programming frameworks that were supported by existing interconnect hardware, rather than next-generation Portals hardware. A principal contribution of this work is that it combines the work discovery step and the work claiming step into a single operation, which results in reducing the communication needed for a steal operation by half.

This work is based on the following insights: (1) implementing distributed task queues using OpenSHMEM allows us access to hardware accelerated network communication operations, (2) the information needed for a stealing process to identify and claim work can be represented compactly, (3) a compact representation of key task queue metadata can be operated on with atomic communication operations, and (4) the additional complexity of this representation adds minimal processing to queue metadata upkeep and maintenance.

This work makes the following contributions: We describe the implementation of the SWS dynamic load-balancing system that is constructed using a novel representation for queue metadata that is amenable to atomic communication operations. We also provide an experimental validation of this approach using two representative applications, a bouncing producer/consumer benchmark and the unbalanced tree search (UTS) benchmark. We show that our approach results in half the communications needed to steal tasks, a corresponding reduction in steal time, and a reduction in the time needed to disseminate tasks to worker processes.

2 BACKGROUND

The advent of hardware supported Remote Direct Memory Access (RDMA) operations has enabled higher performance in traditional two-sided message passing programming models. Modern networking hardware with RDMA support also allows for the efficient realization of one-sided PGAS programming models. Recent interconnect hardware has added to the offload capabilities of the NIC with support for sophisticated atomic access operations and receiver-side offloads such as message matching. These features are important in the context of a distributed dynamic load-balancing system since they provide a way of accessing remote memory without involving the CPU on the target process. Disrupting the CPU from processing tasks in order to handle steal requests and update system state would reduce performance and degrade efficiency.

Work stealing systems rely on a *task queue* that is both used by a local process to enqueue and dequeue tasks and also globally visible so that remote processes may inspect the queue and asynchronously steal tasks. Prior research in this domain has demonstrated that these operations fit well within programming models that use one-sided communication [11]. This work focuses on extending this model in order to reduce the communications needed to safely support asynchronous task stealing.

The problem of load-balancing an application with an irregular workload is complex and at times seemingly contradictory. Consider task granularity as an example: An application with short-lived, fine grained tasks (~10μs) will be easier to balance, but will be more sensitive to overheads in the load balancing system, such as steal latency. Similarly, applications with coarse-grained task sizes (~100 ms) will be more tolerant of larger steal latencies, but may lead to larger imbalance at scale due to longer task duration.

In traditional *work-first* load-balancing systems, local workers process the newest tasks first in their own queues prior to searching elsewhere for work. Once local work is exhausted, the primary problem faced by the process is to identify the location of tasks and determine how much work is available once found. Available work is discovered by selecting a target at random and checking to see if there are tasks that may be stolen. Work stealing systems have been shown to perform best by stealing half of the available work on a target process, striking a tradeoff between reducing steal attempts, while leaving work for other idle processes to discover, giving a balanced workload [17].

2.1 Task Execution Model

The SWS system adheres to the task pool model originally used by the Scioto load balancing system [10]. This model allows the programmer to express a parallel computation by decomposing the problem into a set of *tasks*. Tasks are the fundamental units of work and are executed by processes participating in the parallel computation. Each process maintains its own *task queue*, taken together to form the global *task pool*. The pool is initially seeded with a set of tasks and then processed until there are no more tasks remaining to be executed.

A portable *task descriptor* is used to uniquely identify a task. Task descriptors maintain which program function is to be executed by the task, as well as the parameters or other state needed by the task. The SWS system is built using OpenSHMEM, which provides a global address space based on a symmetric heap. The state necessary for the task function inputs and outputs may be global addresses in the partitioned global address space or any other portable representation that can be used by any process partaking in the parallel execution of the task pool.

Tasks may create new subtasks and add them to the task pool. This enables the recursive expression of parallelism and allows for tasks to be ordered with respect to parent-child data dependencies. Tasks are processed in a LIFO order, which yields a depth-first traversal of the task tree and also bounds the space requirements of the task pool at $O(T_{depth})$, for a task tree, T.

Under the Scioto execution model, all enqueued tasks in the task pool are required to be independent. Parent-child dependencies may be expressed with dynamic task creation, however, any enqueued task in the pool must be able to complete without blocking. Tasks are allowed to communicate and use data stored in the global address space, but they may not wait for results produced by any concurrently executing tasks. These constraints allow SWS to rely on a relaxed fairness model and avoid handling the migration of incomplete tasks while still being suitable for a wide range of applications.

Processes will remove and execute tasks from their local queue until it is exhausted, then move from processing tasks to searching for available work within the system. This mode of operation requires distributed termination detection to determine when all work has been consumed from the task pool and no work is available anywhere in the system.

2.2 Related Work

The techniques presented in this paper build on work optimizing the Unbalanced Tree Search Benchmark (UTS) [12] and the Scioto work stealing infrastructure [10] using PGAS communication primitives. Other work in this area has considered techniques to reduce communication in work stealing through hardware offload operations [21]. With respect to innovations in atomic-based network operations, some vendors have included both hardware and software support for using atomics on structured data [26].

Dynamic load balancing has been extensively studied in the literature. Early work focused on optimized scheduling for task graphs [20] or using graph partitioning to schedule tasks with consideration of data locality [7, 32]. These techniques assume knowledge of total task volume and graph contents prior to balancing and executing tasks.

Cilk [14] popularized work stealing as a means to dynamically load balance fully strict computations. Several other projects have looked at adaptations and implementations of shared memory or distributed shared memory load balancing including NESL and others [4, 23].

Several modern parallel programming frameworks provide support for load-balancing, although typically only within a shared memory domain. Language-level support for dynamic scheduling exists within OpenMP and has been studied within Chapel [13] and X10 [8]. Some parallel programming frameworks, such as Legion [3], also support work stealing within a shared memory environment.

Within the context of distributed memory systems, dynamic load balancing has been widely studied in various contexts [18, 30]. Cilk NOW [6] extended the Cilk model to networks of computers with additional features to deal with adaptive parallelism and fault tolerance. More recent language-level load balancing has been studied within the context of X10 [9, 31].

Optimizations to the work stealing algorithm have included the *help-first* approach to perform locality-aware distributed load-balancing in systems such as SLAW [15, 16] and HotSLAW [35]. Work done with the Habenero environment has focused on adding hierarchy to ensure that target selection during a steal attempt always succeeds in finding work and reduces overall communication traffic [19]. Lifelines [29] have been proposed to improve quiescence detection and eliminate unproductive stealing traffic. The Open Community Runtime (OCR) provides an implementation of the "asynchronous many task" model that is suitable for exascale systems [22]. Work with CHARM++ has also demonstrated that hierarchical load balancing is effective at scaling dynamic load-balancing systems [36]. Private locally accessible task queues have also been studied in conjunction within two-sided communication used for stealing tasks [1].

Our work focuses on accelerating the communication involved in remotely accessing the task queues used by work stealing implementations. Thus, the techniques we present can be used in conjunction with enhancements to the work stealing algorithm or to accelerate existing implementations of work stealing on networks that support the OpenSHMEM parallel programming framework.

3 BASELINE SDC IMPLEMENTATION

We use the Scioto work stealing engine as a baseline and compare this with the SWS approach. In the Scioto framework, each process maintains a double-ended queue (deque), implemented using a circular buffer. This buffer is allocated in a globally-accessible memory region in the PGAS. The task queue is carefully organized such that it provides efficient local queue operations with low overhead as well as allow remote processes to steal available tasks.

The best performing implementation approach in Scioto has been shown to be the "Split Queues, with Deferred Copies, and Aborting Steals" (SDC). The SDC implementation permits steal attempts to abort early in the case that another process has already stolen all tasks, as well as a passive, non-blocking steal completion acknowledgement (deferred copy) [11]. We implemented a version of Scioto using OpenSHMEM for all communication, directly substituting ARMCI calls with the corresponding OpenSHMEM operations. All comparisons to the Scioto baseline implementation use this configuration.

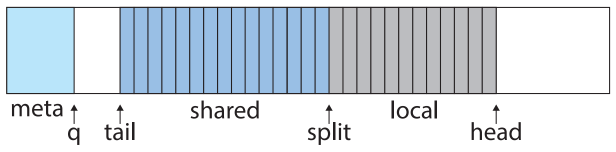


Figure 1: Scioto Task Queue Structure

As shown in Figure 1, Scioto divides the queue into two portions. A local portion which may only be accessed by the owning process, and a shared portion which contains the work available to be stolen by other processes. We interchangeably use the terms *owner* and *target* to reflect the process on which the task queue resides, and is the target of steal operations. A stealing processes may also be referred to as the *initiator* of a steal.

Local queue operations are lightweight and do not require the use of a lock. Remote processes are permitted to steal any tasks located between the tail and the split point, beginning with those closest to the tail. Upon spawning new tasks, they are typically

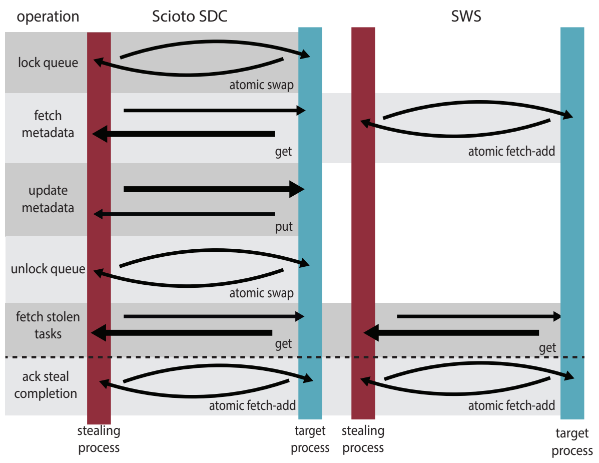


Figure 2: SDC and SWS Native Steal Communications

enqueued locally, however a process may spawn tasks onto remote queues, although with more overhead due to communication.

A local process will move the split point in order to either expose more tasks or reacquire them when it runs out of work. Moving the split point between the local and shared portion of the queue can happen when the process owning the queue performs a *release* or *acquire* operation. The release operation moves the split point closer to the head of the queue, thereby exposing more tasks to the shared portion. The complementary *acquire* operation does the opposite, pushing the split point into the shared portion, hiding tasks from remote processes and making more work available locally. These operations are called periodically when the runtime system discovers that one of the shared or local portions of the queue is empty but the other contains tasks.

3.1 Implementation

The task queue maintains the indices for the tail and split point, a count of the number of local tasks, as well as a lock used when remote processes access the shared portion. These metadata structures are stored in the globally exposed memory region so that they may be queried and modified by other processes.

The following descriptions consider the implementation of the principal operations for the baseline SDC load-balancing system. Each of these operations is described in more detail in [10, 11, 21].

Enqueueing Tasks. New tasks are enqueued at the head of the local portion. The system ensures that there is sufficient space in the queue and then copies the tasks into the local part of the queue, possibly wrapping around the circular buffer. This entire operation is local and may be performed without locking.

Dequeueing Local Tasks. Removing tasks from the head of the queue is a local-only memory copy operation and may be performed without the need for locking.

Release Operation. A release operation occurs when there is work remaining in the local portion of the queue, but the shared portion is empty. Scioto was originally implemented to allow release operations to occur without synchronization. Since a release happens when the shared queue is empty, a steal attempt executing prior to the release will see the empty queue and abort the steal attempt. To change the amount of work visible in the queue, the release operation only needs to update the split point, which can be done atomically, thereby avoiding the need for the owning process to lock the queue.

Acquire Operation. The acquire operation occurs when the local portion of a task queue is empty. If there is available work in the shared portion, the split point is updated to effectively move half of the available work into the local portion of the queue. Since the index of the split point is used by remote processes when stealing work, this operation requires the queue to be locked during the update.

Stealing Tasks. Using the Scioto-derived SDC model, stealing tasks requires synchronization in order to safely check and modify the task queue on a remote process. In total, a steal operation requires six communication operations to complete. Five of these are blocking operations that must complete in order to preserve safety. The final communication (deferred copy) is passive and may be issued without waiting for completion.

To steal work from a remote target, SDC performs the communications operations shown in Figure 2, which correspond to the following steps in a steal operation:

(1) Acquire a lock on the remote queue (atomic)
(2) Fetch the tail and split points of the remote queue to determine the amount of shared work available. (get)
(3) Update tail index on remote queue (put)
(4) Release lock on remote queue (atomic)
(5) Copy stolen tasks from remote queue (get)
(6) Update steal completion status (non-blocking atomic)

The SDC baseline implements all locks with spinlocks. These queue locks are typically uncontended, however when work is sparse in the system (dispersing work initially, or aggressive searching late in the computation), multiple processes may attempt steals simultaneously. The use of application-level spinlocks permit the initiating process to steal work without committing to acquiring the lock and allows periodic polling of the target queue metadata to check if there is any work remaining. This allows stealing processes to *early abort* a steal attempt if there is no work to be found.

Periodically, the runtime must update the local queue to account for the asynchronous completion status messages. This *progress* operation updates the local tail of the queue past all stolen tasks that have signaled completion in order to reclaim space in the queue. This operation does not require locking the queue.

4 SWS IMPLEMENTATION

Stealing processes using the baseline SDC implementation are required to first search for work by examining the queue metadata of a potential target and then modifying it in order to claim the work. As can be seen in Figure 2, this requires locking, read, and write operations to perform these steps. If sufficient information to both discover and claim work can be represented in a form that is suitable for atomic communication operations, then the four communications necessary in SDC to discover and claim work can be reduced to a single step.

Atomic operations in OpenSHMEM can work on values up to 64-bits. The design of SWS relies on using a single 64-bit value (*stealval*) to represent multiple pieces of information, as shown in Figure 3. This design breaks a 64-bit value into four components, of which only the high 24-bits are modified by stealing (initiator) processes and the low 40-bits of data which are only modified by the task queue owner. The owner data is broken into a 1-bit *valid* flag, a 19-bit unsigned field that represents the initial allotment of tasks in the shared portion of the queue and a 20-bit unsigned value that represents the location of the queue tail:

Tail Index: The *tail index* field represents the index of the tail entry on the symmetric heap of the target process.

Initial Tasks: The *initial tasks* field represents the total number of tasks placed into the shared portion of the queue. This is used to calculate the specific block of tasks to copy during a steal operation.

Valid: This flag is used to signal to stealing processes that the target process has disabled steal attempts or is updating the split point.

Steals Attempted: This field represents the number of attempted steal operations (*asteals*) using the steal-half mechanism and is used to determine the volume of the next steal. If the number of attempted steals is greater than the number of steals possible given the initial tasks, this indicates there is no more work available for stealing in that queue.

Task State		Description
Available	**(A)**	shared tasks — unclaimed and available for stealing
Claimed	**(C)**	claimed tasks — steal operation is still in-progress
Finished	**(F)**	stolen tasks — steal completion notification has been sent by the stealing process to target
Invalid	**(I)**	invalid tasks

Table 1: Shared Task States

Tasks in the shared portion of the queue may be in one of the four states listed in Table 1. Initially tasks shared by a release operation are marked as available. Tasks reserved by a stealing process remain in the claimed state until the initiator actively signals steal

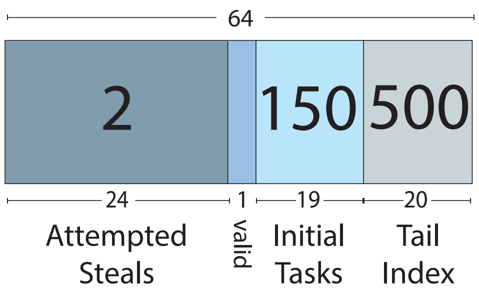


Figure 3: SWS Steal Structure (`stealval`)

completion, when they become marked as finished. Any portion of the queue that contains neither local tasks or shared tasks (available, claimed, or finished) is invalid.

Example: The queue represented by the 64-bit metadata value in the Figure 3 has the following meaning:

(1) The tail starts at index 500. Any steal operations must account for wrapping when determining the block of tasks to be copied. Since task queues are of symmetric size, wrapping steals can be determined locally, without communication.
(2) The target process initially placed 150 tasks into the shared portion of the queue, which are all labeled as *available.* Since we always steal half of the remaining available work, this corresponds to this sequence of 9 steals: {75,37,19,9,5,2,1,1,1}.
(3) The current number of steals attempted is two, which means that the next steal would consist of 19 tasks and that the preceding two blocks of tasks have already been claimed (of 75 and 37 tasks, respectively). With the atomic increment, these 19 tasks become *claimed.* This block of tasks begins at the index at *tail + completed* (or 500 + 75 + 37 = 612) assuming that the queue size is larger than 631 (612 + 19) tasks, otherwise we perform a wrapped steal.
(4) After the steal is completed, the initiator atomically updates a shared array on the target, a *completion array*, with the number of tasks stolen (19), which denotes that the steal is*finished.* The stolen tasks are enqueued locally and able to be processed by the initiating process.

4.1 Implementation

Our implementation[1] of SWS also relies on a split circular buffer for tasks, with both a shared and local portion. In addition, SWS maintains two primary pieces of metadata, the 64-bit atomic *stealval*, and a shared array for tracking the asynchronous completion of steal operations.

Similar to description of the baseline SDC operations, we discuss the implementation of queue operations below:

Enqueueing and dequeing local tasks. This is unchanged from the SDC implementation. Both operations are lightweight, occur without locking and require no interaction with the shared portion of the task queue.

Release Operation. The release operation is invoked when the shared portion of the queue is empty and there are available tasks in the local portion. During a release, the queue split is updated to move half of the available tasks to the shared portion. The *stealval* is atomically reset to reflect the new number of stealable tasks and current tail value of the queue.

Acquire Operation. The acquire operation moves tasks from the shared part of the queue into the local portion. In addition to updating the split point, the *stealval* is updated to reflect the new number of available tasks. Since the queue is not locked in SWS, it is possible for a remote process to begin a steal operation after we have read our queue value, leading to an inaccurate view of the queue state. To prevent this, upon starting an acquire operation,

[1]http://github.com/brianlarkins/saws

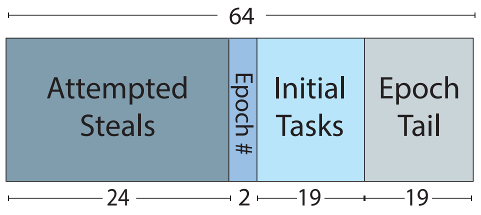


Figure 4: Updated SWS Steal Structure

stealing is temporarily disabled by marking the valid bit in the *stealval* as invalid.

An acquire operation may occur after shared tasks have been claimed, but while tasks are still in the process of being transferred. In our initial implementation, the owning process must wait until all in-progress *claimed* steals become *finished* before updating the *stealval*. Otherwise we lose the means to safely reclaiming queue space. We consider optimizations to this below.

The amount of claimed work is directly known by the number of steals attempted at the time of the acquire and is inspected with a local atomic operation. The total number of stolen tasks is maintained by examining the count of *finished* entries in the *completion array*. If these two counts do not match, there are in-progress steals.

Stealing Tasks. In contrast to the multiple communications required in the SDC version, a steal operation can now be completed with a total of three one-sided communications, as shown in Fig. 2. Of these three steps, only the first two communications block progress — the steal completion notification does not need to complete in order to return to task processing. A stealing process first uses an atomic fetch/add operation to increment the number of attempted steals (*asteals*) part of the *stealval* on the remote target. Once *asteals* has been incremented on the target process no other process may claim the same block of tasks.

Knowing the initial number of tasks available and the number of previously attempted steals allows the stealing process to determine the amount of work that it should steal. If the number of steal attempts is greater than log_2 of the initial tasks, then no work remains, otherwise the amount of work to steal is obtained by dividing the amount of initial work by two for each prior attempt. The displacement from the tail index can now be calculated, skipping previously claimed work.

The stealing process then initiates a blocking one-sided get communication, accounting for the possibility that tasks may wrap around the circular queue. Once the tasks have been copied, the stealing process sends a non-blocking atomic put onto the completion array, to signal to the target that the steal has completed. Similar to the SDC implementation, this last communication is passive and may be issued without waiting for completion.

4.2 Completion Epochs

When work is sparse in the system, the frequency of split-point update operations increases. To avoid the need for waiting for in-flight steals to complete during acquire operations, we amend the *stealval* structure to correspond to the structure shown in Figure 4.

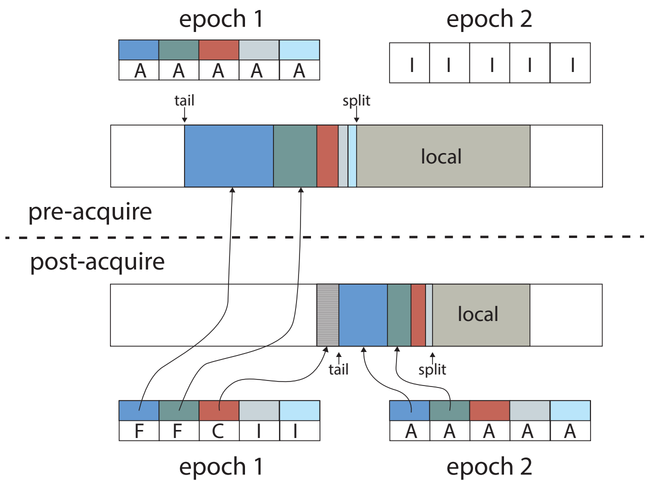


Figure 5: Acquire Behavior with Completion Epochs

Under this scheme, claiming steals happens atomically, but steal-completion may be asynchronous and indeed may overlap with local queue manipulation mechanisms. This is enabled by versioning steal phases into epochs, each of which maintains a separate completion state. An epoch index of anything greater than *MAX_EPOCHS* signifies that the queue is locked by the target process and cannot be stolen from.

This requires updates to acquire, as well as a change to steal completion. When calling acquire, the queue is disabled and the completion array states for the new epoch must be initialized before re-enabling steals.

When stealing tasks, completion is updated to use the epoch number in the *stealval* when writing to a completion array on the target once the stealing process has completed copying the stolen tasks.

The acquire operation tends to happen relatively infrequently, so it is responsible for maintaining queue state given the possibility of incomplete steals across multiple epochs. For example, the tail must be advanced past completed tasks to ensure that there is free space in the circular queue to allow additional tasks to be enqueued. Since steals may be incomplete over several different epochs, all completion arrays are traversed to account for the longest sequence of fully completed steals.

Every acquire starts a new completion epoch. If there are outstanding steals in all completion arrays, then acquire must poll until at least one epoch has fully completed. In our experience, the use of two completion epochs was sufficient to avoid polling.

As with the SDC implementation, the runtime must periodically update the local queue outside of acquire/release operations to account for passive steal completion updates (*progress*). In SWS, reclaiming space in the task queue requires also looping over all outstanding completion epochs and determining the largest sequence of completed steals starting at the oldest epoch's tail. The owner tracks the current tail (last available or claimed task) distinct from the tail value advertised in *asteals*.

We can visualize this behavior in Figure 5. In this figure, we consider the state of the system immediately prior to an acquire call with outstanding steal completions above the dotted line. Several task blocks are in the shared portion of the queue with their steal completion status marked as **A** or *available*.

The lower half of the diagram shows the state after an acquire operation. We can see that the completion array for the first epoch has two steals marked as *finalized* (**F**), indicating completion. One block is marked to signify that it has been *claimed* (**C**). The acquire operation is not permitted to reclaim that block of queue space until that the tasks have been marked as finalized, at which point unclaimed task blocks become *invalid* (**I**). The acquire has updated the *stealval* to reflect the new tail, available task blocks, and has set the completion array to epoch 2.

4.3 Steal Damping

A constraint with using a more compact representation for queue metadata is that it is possible to overflow the *asteals* value in the *stealval* after 16.7 million (2^{24}) steal attempts. To address this possibility, SWS can perform *steal damping*, where a stealing process falls back to a less-aggressive steal mode after discovering that a specific process has no work.

Initially, all targets are said to be in *full-mode*, where steal attempts follow the three-step algorithm described above. Upon determining that a target has run out of work and the *asteals* value exceeds a threshold value, it is marked as being in *empty-mode*.

When stealing from a target in empty-mode, we first check for tasks using a read-only atomic fetch. If the target still has no work, the steal attempt is aborted and a new target is selected. If an empty-mode target has acquired more work, we return it to full-mode status and retry the steal attempt with an atomic fetch/add.

In practice, we found that the size of the *asteals* value never approached the overflow limit. The maximum number of initial tasks is capped at $2^{24} - P$ to ensure that overflows may not corrupt other fields in *stealval*. Enabling steal dampening did not incur any significant performance penalty over non-damped runs for our experimental analysis without overflow conditions.

5 EXPERIMENTAL EVALUATION

Performance was evaluated by comparing SWS against the traditional baseline SDC algorithm in the Scioto framework.

All experiments were performed on the Lotus compute cluster at Rhodes College. The cluster has a total of 2,112 cores on 44 compute nodes of 48 cores each. Each node is configured with two AMD EPYC 7352 24 core CPUs operating at 2.3GHz and 256 gigabytes of memory. Nodes are connected with a Mellanox EDR 100Gb/s InfiniBand fabric, using ConnectX-6 InfiniBand host channel adapters (running at EDR rate).

The software configuration consists of CentOS 7 Linux as the host OS. Both the SDC and SWS implementations use the Sandia OpenSHMEM (SOS) runtime library for PGAS and communication. The SOS implementation is based on the main git development branch[2]. SOS was configured to run using the Unified Communication-X (UCX) runtime, release version 1.10 with multithreading enabled.

[2]SOS commit `0e5fd82aef1683a045c85e0a50939aed80cc21c2`

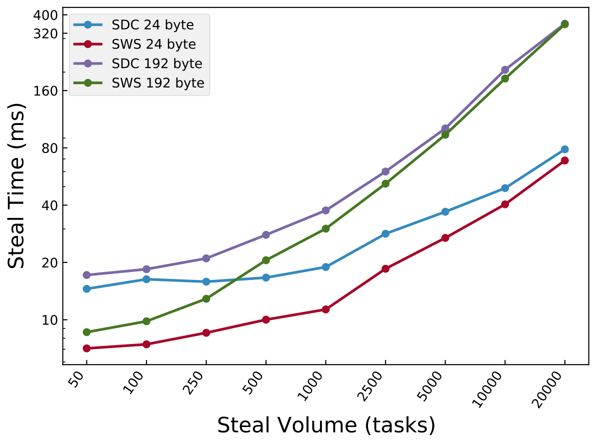


Figure 6: Steal operation times for SDC and SWS.

5.1 Performance Baseline

In Figure 6, we establish a performance baseline by comparing the relative performance of steal operations between the baseline SDC implementation and our SWS implementation with both small tasks (24 bytes) and larger tasks (192 bytes). When the volume of stolen tasks is small, we see that SWS steal times are approximately half of SDC. As the volume increases, the steal operation becomes dominated by the communication of the tasks themselves and the latency from additional communications in the baseline version contribute less to the overall steal time.

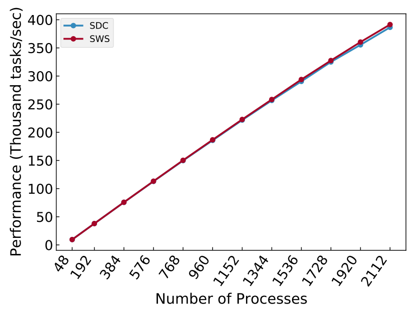
5.2 Benchmark Applications

To further evaluate the effectiveness of the SWS runtime system, we have chosen to run experiments using two benchmark applications: a "bouncing producer-consumer" benchmark and an unbalanced tree search benchmark. The differences in workloads between the applications is shown in Table 2.

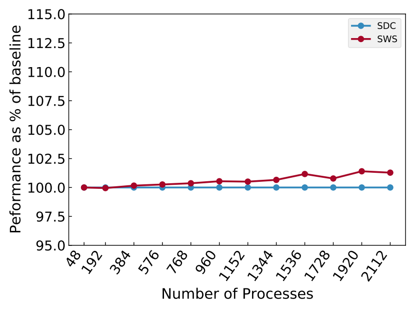
5.2.1 Bouncing Producer-Consumer (BPC). The Bouncing Producer Consumer (BPC) benchmark [11] is designed to challenge a load balancing system's ability to locate and disperse work. BPC produces two types of tasks, producer tasks and consumer tasks, with 1 producer task spawned for *n* consumer tasks. Each producer task creates an additional producer task along with *n* more consumer tasks, until a set depth is reached. In the "bouncing" mode, the producer task is always located at the tail of the queue, so it is first to be stolen. The result of this is that any producer task may bounce between processes several times before finally being executed on another processor.

Each queue was configured such that each producer task produced 8,192 consumer tasks, and has a depth of 500. Consumer tasks take 5ms to complete and producer tasks take 1ms.

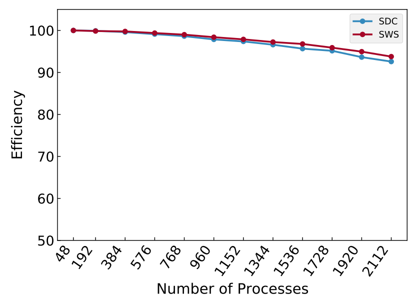
5.2.2 Unbalanced Tree Search (UTS). The Unbalanced Tree Search benchmark (UTS) [12] is representative of an exhaustive state space exploration or combinatorial search problem. The benchmark measures the performance of a parallel search over a deterministic but highly unbalanced tree. The tree is constructed using a random



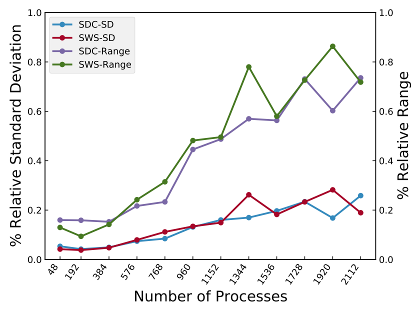
(a) Performance of BPC.



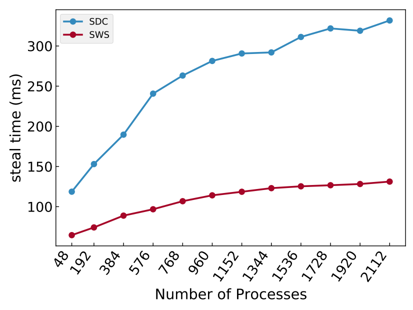
(b) Relative BPC run-time performance improvement with SWS over SDC.



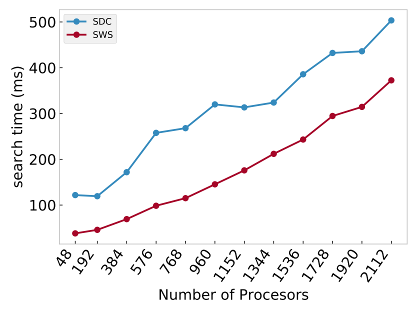
(c) Parallel efficiency of BPC relative to ideal execution.



(d) Variation in BPC runs.



(e) Steal operation times for BPC.



(f) Search times for BPC.

Figure 7: Performance of the Bouncing Producer-Consumer Benchmark with both SDC and SWS steal-half task queues.

stream generated using the SHA-1 secure hash algorithm. Nodes in the tree are represented using a 20-byte hash digest – children are located by composing the digest of the parent node and the identifier of the child. This approach leads to a high degree of variability in terms of subtree size for a given node. Each tree node corresponds to a task, with a wide range of potential work (searching) for a given node. The UTS benchmark was configured to perform an exhaustive search on an unbalanced tree containing 270,751,679,750 nodes (T1WL), with a depth of 18.

5.3 Performance Evaluation

The design of SWS leads us to expect performance gains in several areas: reduced latency for steal operations, lower overhead from the load-balancer due to less time spent searching for work, and improved responsiveness (throughput) at the target. All results are with steal damping and completion epochs. Timing was done with TSC-based timers that were calibrated every run. Whole program timers start after initialization and stop after global termination is detected. Since processes continue to search for work until it is globally exhausted, these times represent the maximum runtime of any process. All times are averaged over 10 runs at each process count.

Benchmark	Total Tasks	Avg. Task Time	Task Size
BPC	2,457,901	5 ms	32 bytes
UTS	270,751,679,750	0.00011 ms	48 bytes

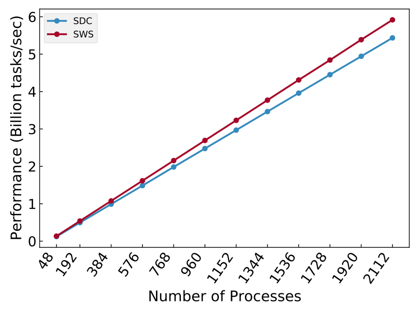
Table 2: Benchmarking workload characteristics.

5.3.1 Bouncing Producer Consumer (BPC). In Figure 7 we can see the performance of the bouncing producer consumer benchmark with the SDC and SWS implementations. In terms of overall performance, measured by runtime, we see only modest improvement.

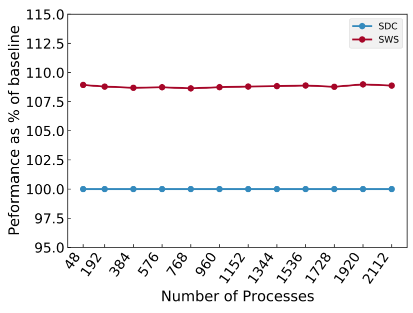
Looking at both Fig. 7a and Fig. 7b, we can see that both SDC and SWS perform similarly at small process counts, with SWS improving as the workload scales out. Since BPC tasks are relatively coarse-grained, computation dominates execution time, giving similar performance. Discovering work at scale requires more steal attempts, resulting in the benchmark being more sensitive to communication overhead. In Fig. 7c, we see that both systems perform well at scale, with SWS maintaining a slight edge in efficiency.

When considering small performance gains, we must consider the variability between independent runs. In Fig. 7d, we look at the relative standard deviation and range (max time - min time) as a percentage of the averaged total run time. We see that the relative standard deviation is flat and consistently under 0.1% for all runs. Similarly, the relative range shows that the difference between the fastest and slowest runs is also very small, mostly under 0.2%.

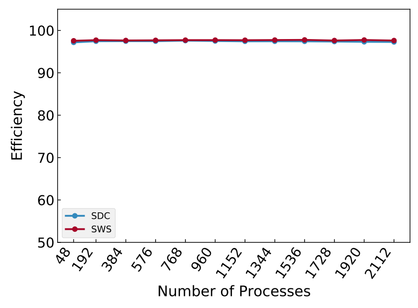
Both Figures 7e and 7f consider load-balancing efficiency. SDC and SWS have different models of discovering and stealing work – we treat *steal time* as time spent performing successful steal operations and *search time* as time spent looking for work. Failed steal attempts are treated as searches and successful attempts as steals. If we consider steal operations in BPC, we can see that the



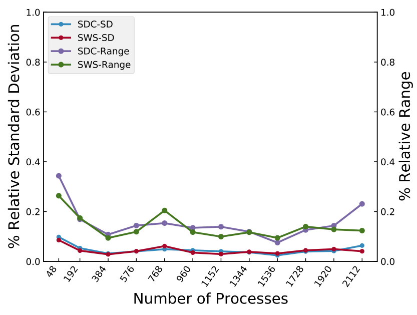
(a) Performance of UTS.



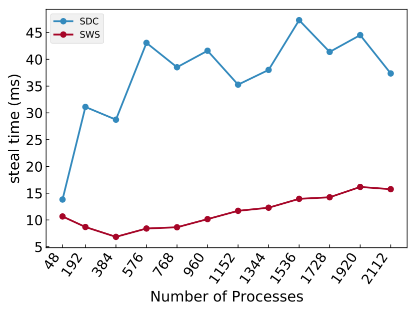
(b) Relative UTS run-time performance improvement with SWS over SDC.



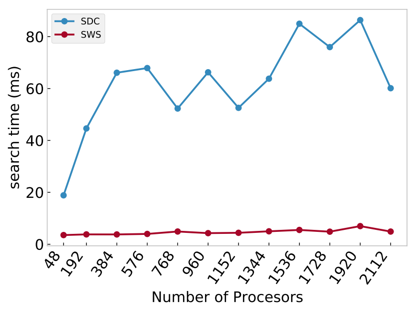
(c) Parallel efficiency of UTS relative to ideal execution.



(d) Variation in UTS runs.



(e) Steal operation times for UTS.



(f) Search times for UTS.

Figure 8: Performance of the Unbalanced Tree Search Benchmark with both SDC and SWS steal-half task queues.

optimized communication in SWS contributes to scalability even with compute-dominated dynamic workloads. In Fig. 7e, we see that the time spent stealing work remains relatively flat, in contrast to the growth trend of SDC as the process count increases.

5.3.2 *Unbalanced Tree Search (UTS).*

The UTS benchmark has a highly irregular workload consisting of a large number of very short duration tasks. This benchmark is designed to be a challenging workload for dynamic load balancers. Due to the large number of tasks spawned and their small lifetimes, UTS is much more sensitive to the performance of both attempted and successful stealing communication.

The performance characteristics of the UTS benchmark application are shown in Figure 8. The SWS implementation shows a clear performance gain over SDC in terms of task throughput, as shown in Fig. 8a. This corresponds to a roughly 9% improvement in overall program runtime as shown in Fig. 8b when comparing SWS relative to SDC. In Fig. 8c, we see that both systems perform well at scale, with SWS maintaining a slight edge in parallel efficiency.

Looking at the variability of runs for UTS in Figure 8d, we see similar results to BPC, with the standard deviation relative to average runtime under 0.1% and the range (max-min) relative to average runtime is also small, typically under 0.2%, indicating that the performance improvement in SWS is significant relative to run variability.

Figs. 8e and 8f again consider the efficiency of the load-balancing system. When considering the total time spent stealing work in UTS, in Fig. 8e, we see that SWS outperforms SDC with lower steal overhead. This graph shows an improvement in steal times by a factor of 3-4, reflecting the lower number of communications.

With work discovery in the system, we consider both time spent searching for and stealing work. The additional communications needed for the baseline SDC implementation to determine a lack of work incurs higher overhead when compared with the single communication test used by SWS. The impact of this can be seen in Fig. 8f, where SWS shows very low and flat trending search times across all core counts, as compared with the higher and upward trend seen with the SDC version. Searching for work is more efficient in SWS, not only due to fewer communications, but also the lower volume of communication needed to discover work (i.e. a single 64-bit word vs. the queue metadata structure).

6 CONCLUSION

Work stealing is a known, effective method of load balancing tasks in a distributed system. Current high-performance computing trends are leading to increased programmability and acceleration in the network. In this paper, we have shown how a compact task queue representation enables the use of atomic operations to improve a dynamic work stealing system.

Our implementation reduces the communication needed for steals by half and has significantly better properties when a target is contended. This eliminates the need for locking a critical section and leads to lower steal overheads and a reduction in time spent looking for work. Further, our approach combines both the work discovery steps and work claiming steps into a single operation, leading to a reduction in both communication count and volume.

We show that our approach is effective by evaluating two representative benchmarks, BPC and UTS. We also provide techniques for improving asynchrony in queue management operations such as acquire and release. The introduction of completion epochs allows the queue owner to update the split point without needing to lock the queue and poll until pending steals complete. More broadly, we show how carefully partitioning structured data fields into an atomic-friendly format can lead to both reduced communication and higher performance.

ACKNOWLEDGMENTS

Initial development of SWS was completed using the cluster systems at the San Diego Supercomputer Center, with access provided through the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575. This work was also supported by the National Science Foundation under award 2018758.

REFERENCES

[1] Umut A. Acar, Arthur Chargueraud, 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* (Shenzhen, China) *(PPoPP '13)*. ACM, New York, NY, USA, 219–228. https://doi.org/10.1145/2442516.2442538

[2] Brian W. Barrett, Ron Brightwell, Ryan E. Grant, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Arthur B. Maccabe, and Trammell Hudson. 2017. *The Portals 4.1 Network Programming Interface.* Technical Report SAND2017-3825. Sandia National Laboratories.

[3] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 66, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389086

[4] Guy E. Blelloch and John Greiner. 1996. A Provable Time and Space Efflcient Implementation of NESL. In *Proc. 1st ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP)*. Philadelphia, Pennsylvania, 213–225.

[5] Robert D. Blumofe and Charles Leiserson. 1994. Scheduling multithreaded computations by work stealing. In *Proc. 35th Symposium on Foundations of Computer Science (FOCS)*. 356–368.

[6] Robert D. Blumofe and Philip A. Lisiecki. 1997. Adaptive and reliable parallel computing on networks of workstations. In *Proc. USENIX Annual Technical Conference (ATEC)* (Anaheim, California). 10–10.

[7] Ümit V. Çatalyürek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and Lee Ann Riesen. 2007. Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations. In *Proc. 21st Intl. Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1–11. http://dx.doi.org/10.1109/IPDPS.2007.370258

[8] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. Conf. on Object Oriented Prog. Systems, Languages, and Applications (OOPSLA)*. 519–538.

[9] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorty, Doug Lea, Vijay Saraswat, and Tong Wen. 2008. Solving Irregular Graph Problems Using Adaptive Work-Stealing. In *Proc. 37th Int Conf. on Parallel Processing (ICPP)*. Portland, OR.

[10] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan. 2008. Scioto: A Framework for Global-View Task Parallelism. In *Proc. 37th Intl. Conf. on Parallel Processing (ICPP)*. 586–593.

[11] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable Work Stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland, Oregon) *(SC '09)*. ACM, New York, NY, USA, Article 53, 11 pages. https://doi.org/10.1145/1654059.1654113

[12] James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P. Sadayappan, and Chau-Wen Tseng. 2008. A message passing benchmark for unbalanced applications. *J. Simulation Modelling Practice and Theory* 16, 9 (2008), 1177 – 1189. https://doi.org/DOI:10.1016/j.simpat.2008.06.004

[13] Noah Evans, Stephen L. Olivier, Richard Barrett, and George Stelle. 2017. Scheduling Chapel Tasks with Qthreads on Manycore: A Tale of Two Schedulers. In *Proc. 7th Intl. Workshop on Runtime and Operating Systems for Supercomputers* (Washingon, DC, USA) *(ROSS '17)*. ACM, New York, NY, USA, 4:1–4:8. https://doi.org/10.1145/3095770.3095774

[14] M. Frigo, C. E. Leiserson, and K. H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. Conf. on Prog. Language Design and Implementation (PLDI)*. ACM SIGPLAN, 212–223.

[15] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. 2009. Work-First and Help-First Scheduling Policies for Terminally Strict Parallel Programs. In *Proc. 23rd Intl. Parallel and Distributed Processing Symposium (IPDPS)*.

[16] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. 2010. SLAW: A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Bangalore, India) *(PPoPP '10)*. ACM, New York, NY, USA, 341–342. https://doi.org/10.1145/1693453.1693504

[17] Danny Hendler and Nir Shavit. 2002. Non-blocking steal-half work queues. In *Proc. of the 21st Symposium on Principles of Distributed Computing* (Monterey, California) *(PODC '02)*. ACM, New York, NY, USA, 280–289. https://doi.org/10.1145/571825.571876

[18] V. Kumar, A. Y. Grama, and N. R. Vempaty. 1994. Scalable Load Balancing Techniques for Parallel Computers. *J. Parallel Distrib. Comput.* 22, 1 (1994), 60–79. https://doi.org/10.1006/jpdc.1994.1070

[19] Vivek Kumar, Karthik Murthy, Vivek Sarkar, and Yili Zheng. 2016. Optimized Distributed Work-Stealing. 74–77. https://doi.org/10.1109/IA3.2016.019

[20] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *Comput. Surveys* 31, 4 (1999), 406–471. https://doi.org/10.1145/344588.344618

[21] D. Brian Larkins, John Snyder, and James Dinan. 2019. Accelerated Work Stealing. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 75, 10 pages. https://doi.org/10.1145/3337821.3337878

[22] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, , B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo. 2016. The Open Community Runtime: A runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2016.7761580

[23] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. 2009. Idempotent work stealing. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)* (Raleigh, NC, USA). 45–54. http://doi.acm.org/10.1145/1504176.1504186

[24] MPI Forum. 2015. *MPI: A Message-Passing Interface Standard Version 3.1.* Technical Report. University of Tennessee, Knoxville.

[25] J. Nieplocha and B. Carpenter. 1999. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. *Lecture Notes in Computer Science* 1586 (1999), 533–546. citeseer.ist.psu.edu/nieplocha99armci.html

[26] NVIDIA Inc. 2021. MLNX_OFED Documentation Rev 5.3-1.0.0.1. https://docs.mellanox.com/display/MLNXOFEDv461000/Advanced+Transport.

[27] OpenFabrics Alliance 2021. OpenFabrics Interface Application Programming Interface, Version 1.12.1. https://ofiwg.github.io/libfabric/.

[28] OpenSHMEM Specification Committee 2020. OpenSHMEM Application Programming Interface, Version 1.5. http://www.openshmem.org.

[29] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. 2011. Lifeline-based Global Load Balancing. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. ACM, New York, NY, USA, 201–212. https://doi.org/10.1145/1941553.1941582

[30] Amitabh Sinha and Laxmikant V. Kalé. 1993. A Load Balancing Strategy for Prioritized Execution of Tasks. In *Proc. 7th Intl. Parallel Processing Symposium (IPPS)*. 230–237.

[31] Olivier Tardieu, Haichuan Wang, and Haibo Lin. 2012. A Work-Stealing Scheduler for X10's Task Parallelism with Suspension. *ACM SIGPLAN Notices* 47, 267–276. https://doi.org/10.1145/2145816.2145850

[32] Aleksandar Trifunović and William J. Knottenbelt. 2008. Parallel multilevel algorithms for hypergraph partitioning. *J. Parallel Distrib. Comput.* 68, 5 (2008), 563–581. https://doi.org/10.1016/j.jpdc.2007.11.002

[33] Unified Communication Framework Consortium 2021. Unified Communication X Application Programming Interface, Version 1.10. https://www.openucx.org/documentation/.

[34] UPC Consortium. 2013. *UPC Language and Library Specifications, v1.3.* Technical Report LBNL-6623E. Lawrence Berkeley National Lab.

[35] Ke Wang, Xiaobing Zhou, Tonglin Li, Dongfang Zhao, Michael Lang, and Ioan Raicu. 2014. Optimizing Load Balancing and Data-Locality with Data-aware Scheduling. https://doi.org/10.13140/2.1.4577.8880

[36] Gengbin Zheng, Esteban Meneses, Abhinav Bhatele, and Laxmikant V. Kale. 2010. Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops (ICPPW '10)*. IEEE Computer Society, Washington, DC, USA, 436–444. https://doi.org/10.1109/ICPPW.2010.65