Accelerated Work Stealing

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

John Snyder
Duke University
Durham, NC
jsnyder@cs.duke.edu

James Dinan
Intel Corporation
Hudson, MA
james.dinan@intel.com

ABSTRACT

Realizing scalable performance with irregular parallel applications is challenging on large-scale distributed memory clusters. These applications typically require continuous, dynamic load balancing to maintain efficiency. Work stealing is a common approach to dynamic distributed load balancing. However its use in conjunction with advanced network offload capabilities is not well understood. We present a distributed work-stealing system that is amenable to acceleration using the Portals 4 network programming interface. Our work shows that the structures provided by Portals to handle two-sided communication are general-purpose and can accelerate work stealing. We demonstrate the effectiveness of this approach using known benchmarks from computational chemistry and for performing unbalanced tree searches. Results show that Portals accelerated work-stealing can greatly reduce communication overhead, task acquisition time, and termination detection.

CCS CONCEPTS

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

ACM Reference Format:

D. Brian Larkins, John Snyder, and James Dinan. 2019. Accelerated Work Stealing. In *48th International Conference on Parallel Processing (ICPP 2019)*, August 5–8, 2019, Kyoto, Japan. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3337821.3337878>

1 INTRODUCTION

Many parallel problems exhibit irregularity in the tasks created during their execution that often leads to load imbalance. This imbalance is frequently caused by sparsity in the data, irregularity inherent in the problem, or a lack of parallel slackness. Static techniques to decompose and schedule tasks are ill-equipped to deal with the irregularity present in these problems. Solutions are often most naturally expressed with approaches such as recursive spatial decomposition for handling sparsity and irregularity. Decomposing these problems into tasks and dynamically assigning them to available resources permits execution with good scalability and parallel performance.

The use of a distributed dynamic load balancing scheme can address many of the problems of static systems. Randomized work

stealing, popularized by the Cilk [17] programming language, requires idle processors to select targets at random and attempt to steal work from them. This method has been shown to be optimal for large classes of problems and has low space bounds [5, 23].

Remote direct memory access (RDMA) operations (i.e., remote read, write, and atomic operations) are commonly supported by the high-speed fabrics used to construct high performance computing (HPC) systems. This allows for the efficient realization of a Partitioned Global Address Space (PGAS) in systems such as OpenSHMEM [32], Unified Parallel C [40], or the MPI Remote Memory Access (RMA) interface [30]. The PGAS model exposes a global view of memory for all participating processes along with one-sided communication operations for accessing non-local data. Both of these aspects make the PGAS programming model a natural fit for implementing the work stealing operations in a dynamic load balancing system. Within the context of work stealing, RDMA hardware support allows for idle workers to search for and acquire work without interrupting active computation at the target.

Scioto (Shared Collection of Task Objects) [12, 13] is a PGAS-based task parallel programming model, whose work stealing system utilizes RDMA capabilities via the Aggregate Remote Memory Copy Interface (ARMCI) [31]. In this work, we revisit the design of this dynamic load-balancing system and consider improvements made possible by recent work in high performance networking.

Using a traditional PGAS approach, work stealing requires a sequence of one-sided RDMA communications to complete a steal operation. A conventional system must locate and identify work, steal it, and update the state of the work queue on the target. While this has been shown to provide good scalability and occurs without interrupting the target process, this sequence of communication results in high steal latencies. This work explores the use of an alternative communication interface to greatly reduce communication when stealing. Specifically, we consider the use of the Portals 4.2 network programming interface [2], a well-known interconnect programming model that is designed to offload communication processing to the Network Interface Controller (NIC). Portals 4 has been implemented with hardware acceleration within the Bull* BXI* interconnect [11] and Portals 3 was implemented on the Cray* Seastar* interconnect [7].

In contrast with traditional RDMA approaches, our Portals Accelerated Work Stealing system, dubbed PAWS, uses Portals primitives in a novel manner, leading to improved performance and significantly reduced overhead for load balancing operations. A problem with traditional work stealing systems is that determining the location of available tasks and stealing an appropriate amount of them is incumbent on the stealing process. This results in a steal operation consisting of a multi-step transaction (searching, determining half of tasks available, copying work, and updating target state). One of the primary contributions of the PAWS system is to give

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPP 2019, August 5–8, 2019, Kyoto, Japan

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6295-5/19/08...\$15.00

<https://doi.org/10.1145/3337821.3337878>

the target process control over how much work should be stolen once it has been found. By shifting this responsibility, the stealing process only needs to locate and copy work, simplifying the steal transaction down to a single network round-trip.

This work is based on the following insights: (1) implementing distributed task queues using Portals allows us to leverage hardware accelerated network primitives, (2) steal operations can be mapped directly to Portals operations which exploit message ordering guarantees and would require additional round-trip communication using traditional RDMA or PGAS systems, (3) Portals structures place control of the amount of work exposed to the *target* of the steal operation, rather than with the stealing process, and lastly (4) this approach enables the target of steal operations to maintain passivity, which is required to achieve low overheads at the target of a steal operation and low latencies at the initiator.

This work makes the following contributions: We describe the implementation of the PAWS dynamic load-balancing system that is constructed using a novel application of the Portals matching interface and message ordering rules. We also provide an experimental validation of our approach, through two representative scientific applications, the MADNESS multiresolution chemistry framework and the unbalanced tree search (UTS) benchmark. We show that our approach results in a five-fold reduction in communications needed to steal tasks, a reduction in the amount of time necessary to disseminate tasks to workers, and an improved termination detection process that is amenable to offload and reduces termination detection time by a factor of nearly 3x.

2 BACKGROUND

The introduction of NICs with support for one-sided Remote Direct Memory Access (RDMA) operations was shown to provide an efficient and scalable basis for supporting work stealing [13]. Recent work in high performance networking has introduced a number of new receiver-side offloads (e.g., message matching) and remote access operations (e.g., atomics). These features help reduce latency, reduce CPU interrupts, and limit the amount of CPU involvement in message processing [11, 37]. Such innovations in network design are important within a dynamic load-balancing system since any interruption of a host CPU core from processing tasks will negatively impact program performance and parallel efficiency. Recently, programmable, “smart” NICs have also been introduced, along with software APIs that allow for programmable, in-network computing [21, 22]. Programmable NICs can offload arbitrary message processing; however, they have not yet become mainstream in HPC systems. In this work, we take the Portals 4 interface as a representative example of emergent offloads in HPC systems and demonstrate that Portals structures created to accelerate point-to-point communication can be repurposed to accelerate work stealing.

The fundamental data abstraction in a work stealing system is a task queue, in which new tasks are enqueued, local work may be dequeued, and remote processes may asynchronously steal tasks from the queue. Prior work has demonstrated that this structure is well-suited for one-sided communication with offload support [13]. The principal area of investigation in this work is the implementation of this structure and the acceleration of remote access using Portals 4 networking primitives.

There are a number of dimensions to consider when using a load-balancing system in an application with an irregular workload. For example, an application with coarse-grained tasks (~ 100 ms) can dominate the overhead factors in a load-balancing system but may lead to larger imbalances at scale due to the large task size. On the other hand, an application with fine-grained tasks ($\sim 10\mu$ s) will be easier to balance, but at the expense of being more sensitive to steal overheads.

One of the principal problems in a work stealing system is knowing where work is and how much is available when it is found. Conventionally, locating work is done by randomly selecting a target. Once work is located, work stealing systems tend to perform best by stealing half of the available work, so as to reduce communication overhead while leaving enough work for future steals to yield a balanced workload [20]. A challenge with this work-first approach is that the burden of locating work and stealing tasks is incumbent on the stealing process.

2.1 Task Execution Model

We adopt the *task pool* model proposed by Scioto [12]. Task pools permit the programmer to express a parallel computation by decomposing the problem into a set of *tasks*. A task is the fundamental work unit and will be executed by a process participating in the parallel computation. The task pool must be seeded with an initial set of tasks. The pool is then processed until no tasks remain to be executed.

Tasks are represented by a portable *task descriptor*, which contains references to addresses in a companion global address space that are used as function input and output locations. Since tasks primarily rely on data specified in the partitioned global address space they become portable and may be executed by any process participating in the computation.

Tasks are permitted to spawn new subtasks and add them to the task pool. This enables parallelism to be expressed recursively and allows for task ordering 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 $O(T_{depth})$, where T is the task tree.

The PAWS task execution model requires that all tasks enqueued for execution are independent. While dynamic task creation can support parent-child dependencies, any enqueued task must be able to complete without blocking. Tasks are permitted to access data or communicate through data in the global address space, but they may not wait for results produced by concurrently executing tasks. These restrictions permit an execution style that is suited to a wide range of applications but allow PAWS to rely on a relaxed fairness model and avoid migrating partially executed tasks.

Processes consume tasks until their local task pool is exhausted, then switch from processing tasks to seeking the remaining tasks within the system. This behavior requires a distributed termination detection system to identify when all work has been consumed from the task pool on all participating processes.

2.2 Related Work

The techniques presented in this paper build on prior work on optimizing the Unbalanced Tree Search Benchmark (UTS) [14] and

the Scioto [12] work stealing infrastructure using PGAS communication primitives. The use of network offload message matching support has also been explored in the context of distributed data structures and related programming models [26, 27].

Dynamic load balancing has been well studied in the literature. Early work focused on optimizing schedules for task graphs [25] or the use of graph partitioning to schedule tasks with consideration of data locality [8, 39]. These techniques assume knowledge of total task volume and graph contents prior to balancing and executing tasks.

Cilk [17] 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, 29].

Modern parallel programming frameworks provide support for load-balancing, although often only within a shared memory domain. Language-level support for dynamic scheduling exists within OpenMP and has been studied within Chapel [15] and X10 [9]. 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 [23, 35]. 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 [10, 36].

Optimizations to the work stealing algorithm have included the *help-first* approach to perform locality-aware distributed load-balancing in systems such as SLAW [18, 19] or HotSLAW [41]. 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 [24]. Lifelines [34] 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 [28]. Private locally accessible task queues have also been studied in conjunction with 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 Portals 4 networking interface.

3 THE PORTALS NETWORKING INTERFACE

The Portals framework provides several low-level data abstractions that are used to implement PAWS, our distributed dynamic load balancing system. PAWS is constructed upon a one-sided communication model, which allows for asynchronous data access while avoiding processor involvement at the target node. Portals provides support for both one-sided (a.k.a. non-matching) and matching communication semantics. Matching communication was designed to support MPI; we describe a novel application of the Portals matching interface abstractions to implement one-sided operations on the

Figure 1: Portals data structures and example put operation using the Portals matching interface.

task queue structure that is the basis of PAWS. By using features in the Portals interface, we can take advantage of network hardware offload [7, 11] to a greater extent than if PAWS were implemented using conventional PGAS techniques.

Portals specifies a *network interface* (NI) as a per-process abstraction of a physical network interface on the target process. The interface is defined at creation to be either a *matching* or a *non-matching* interface, depending on the intended use for the interface (message-passing or one-sided). PAWS relies on the Portals matching interface.

Portals defines two primary roles for communication operations. The *initiator* is the process that issues Portals put, get, and atomic operation requests and the *target* is the process receiving requests. The initiator defines a *memory descriptor* (MD), which describes a region of memory to be used with locally-initiated Portals operations, as well as additional structures used to track communication events for operations on the given region.

Target processing is defined in terms of a *portal table*. An index into the portal table is specified on every Portals communication operation and is used to separate and classify different channels between endpoints. Every *portal table entry* (PTE) maintains a list of memory regions that are available for communication operations as well as an event queue to track asynchronous notifications for ongoing communication.

A PTE using the matching interface maintains a list of matchlist entries (ME). Each ME specifies a set of matching criteria, including a unique *match bits* field, and a memory region associated with the entry. If all of the match criteria are satisfied, the communication operation commences working on the corresponding memory region. For example, a process may post an `MPI_Recv()` with a specific tag that maps to the match bits field. Later, Portals would receive a communication request for an `MPI_Send()` with the same match bits set, which would be matched to the posted ME. If there are multiple MEs with identical matching criteria, Portals selects the ME that was posted onto the matchlist first.

Figure 2: Task Pool Queue Structures

Portals provides *event queues* and *event counters* to track events relating to initiator- and target-side Portals operations. Full events contain detailed information about the event and communication operation involved, whereas counting events are lighter weight and capture only the success or failure of a given operation.

On the initiator, an event queue may be associated with an MD, whereas on the target, an event queue may be associated with each PTE. Counters can also be used to track both initiator and target events. Unlike target-side event queues, a counter may be associated with each ME in the matchlist. PAWS typically uses counters to watch for successful completion at the initiator and full events to track steal completions on the target. More detailed descriptions of event and counter use within PAWS are provided below.

To put everything together, Figure 1 shows the relationships between the target-side data structures and a sample `Pt1Put()` operation, used in typical MPI communication. In this example, the initiator process has called `Pt1Put` with the match bits set to 74. When the target process receives the request, it is dispatched through the initiator-specified PT index. Next, the match list is searched until a 74 is found or the end of the list is reached. Portals also supports mask bits that can be set on an ME, allowing the receiver to ignore some or all of the match bits. Once a matching ME has been located, the `Pt1Put` data is copied into the corresponding memory region. The operation may cause a PUT event to be appended to the event queue or the ME success counter to be incremented. An acknowledgement is sent back to the initiator, possibly generating another full event or counting event according to the MD that the initiator supplied to the `Pt1Put` operation. Portals matching lists also allow data to be retrieved, rather than sent, using `Pt1Get` operations. We use this unique capability, in combination with Portals structures to provide target-side control, in the implementation of PAWS.

4 IMPLEMENTATION OF PAWS

We take the Scioto work stealing engine as a baseline in this work and compare this approach with one accelerated using Portals 4¹. In Scioto, each process maintained a double-ended queue (deque), implemented using a circular buffer allocated in a globally-accessible memory region. The task queue must be carefully organized to

provide efficient local access and also allow remote processes to steal available work.

Scioto splits the task queue into two portions, a private portion that may be accessed by the local process without the need for locking and a shared portion of the queue that contains work that is available for stealing by other processes. New tasks are pushed onto the head of the queue's local portion and remote processes may steal tasks from the tail of the shared portion. New tasks are typically enqueued locally. Spawning tasks on remote processes is permitted with some additional overhead.

A process may perform a *release* or *acquire* operation to move the location of the split between the local and shared portions. The *release* operation moves the split point into the local portion, thereby exposing more work in the shared portion. Making more tasks available for the local process, *acquire* is a complementary operation and moves the split into the shared portion. 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.

We first implemented a version of Scioto where all ARMCI calls are replaced with Portals one-sided operations. This baseline implementation is able to use several different stealing strategies with respect to work splitting. Earlier work with Scioto showed the best performing variant of Scioto was 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) [13]. All comparisons to the Scioto baseline implementation use this configuration.

4.1 Baseline Implementation

The baseline implementation is a direct port of the ARMCI-based implementation of Scioto as described in [13] to Portals 4. Mutex locks are implemented using the Portals atomic swap operation, `Pt1Swap()`. One-sided put and get operations are implemented with `Pt1Put()` and `Pt1Get()` calls that perform the same function as their ARMCI counterparts.

The entire task queue is allocated and associated with a single Portals matchlist element, as shown in Figure 2a. The task queue maintains the indices for the tail and split points, a count of the

¹<http://github.com/brianlarkins/paws>

Figure 3: Baseline and PAWS Native Steal Communications

number of local tasks, as well as a mutex 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 section describes the implementation of the principal operations of the baseline load-balancing system:

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

Dequeuing 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 any locking.

Release Operation. The release operation is used whenever the shared portion of a task queue is empty and there is work in the local portion. The original Scioto implementation utilized a queue metadata structure that is mirrored in our baseline model, which allows the release to be completed without synchronization. If a steal is attempted while a release is occurring it simply appears that less work is available in the shared portion of 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 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, the baseline performs the communications operations shown in Figure 3, 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)

4.2 Portals Implementation

Our conception of a native Portals implementation of the task queue in PAWS also relies on a split circular buffer for tasks with both shared and local portions. In contrast to the baseline implementation, multiple Portals matchlist entries are configured to correspond to blocks of tasks within the queue, thereby handing off steal operation processing to the network interface. Each matchlist entry corresponds to a geometrically decreasing number of tasks. This configuration allows the *target* to arrange the sequence of stealable task blocks, with each subsequent block corresponding to half of the remaining work in the shared portion of the queue.

Upon startup, a Portal table entry is allocated for each task queue. Initially, a single matchlist entry is appended to the overflow matchlist, marking the end of the shared portion of the queue (EOQ) and signifying that no work is available to steal.

As shown in Figure 2b, the task queue is implemented with a contiguous deque structure, but the Portals matchlist entries each correspond to a block of tasks in the shared portion instead of the entire queue. Since much of the communication processing is handled by Portals, an additional list, the *matchpair list*, is maintained by PAWS and tracks an application-level view of the Portals internal state.

With Portals maintaining the matchlist internally and managing steal operations, the queue state within PAWS must be kept up to date with respect to asynchronous Portals operations. This is handled with the matchpair list and by registering a Portals event queue for all of the matchlist entries. The matchpair list allows PAWS to make changes to the Portals matchlist, for example to re-acquire work from the shared portion of the queue. When a steal operation completes, it generates an event, which is added to the queue. PAWS processes these events to keep the tail of the queue up to date and to ensure space is available for new tasks as they are added.

Enqueueing and dequeuing local tasks. This is unchanged from the baseline 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 a task queue is empty and there are tasks available in the local portion. During a release, the queue split is moved so that half of the tasks are now in the shared portion. Starting from the tail, half of the newly shared tasks are grouped together and associated with a matchpair and matchlist entry. The ME is appended to the priority list, and the process is repeated with

half of the remaining tasks until the end of the shared portion of the queue is reached, similar to the state shown in Figure 2b.

Each matchlist entry is configured with the same match value and refers to the memory region in the queue that contains the corresponding tasks. We take advantage of ordering guarantees needed for matching receives within a two-sided communication model, such as MPI. Portals ensures that incoming requests are paired with posted matchlist entries in the order in which they were placed onto the list. This ensures that a steal attempt always results in matching the largest block of tasks available in the shared portion of the queue. Each entry is configured to generate a Portals event when tasks are stolen from the ME.

Portals also permits the memory regions for matchlist entries to be specified with I/O vectors. This allows a block of tasks that wraps around the ends of the circular queue to be accessed with a single communication operation.

Acquire Operation. With the native Portals implementation, when work is available in the shared portion and the local portion is empty, the acquire operation not only needs to update the split point as in the baseline version but must also update the matchlist entries to reflect the amount of work remaining in the updated shared portion.

Initially, we mirrored the lock-based approach of the original implementation. The PTE that corresponds to the task pool is disabled to prevent steals from occurring during the acquire. Next, the matchpairs list is traversed, unlinking all of the active matchlist entries. Once all matchlist entries have been removed, the PTE is re-enabled and the updated shared portion of the queue is traversed, creating matchlist entries in a similar fashion to the release operation. Since the Portals implementation must update both the matchlist and matchpairs list, a locking approach to the acquire operation requires more processing than the baseline implementation.

To remediate this overhead, Algorithm 1 provides a mechanism to reset the queue state when acquiring tasks from the shared portion without the need for disabling or otherwise locking the PTE. Starting from the split point and working towards the tail of the queue, we traverse the matchpairs list and attempt to unlink the corresponding matchlist entry. Since steal requests always match the head of the matchlist, we work from the split point (i.e. the tail of the matchlist) towards the tail of the queue (the head of the matchlist) checking for entries that are in use. If the unlink operation fails, we know a steal is pending, and we must wait for the steal event in order to keep the tail of the queue consistent with the actual queue state. Steals made after all matchlist entries have been removed will result in a match against the *end-of-queue* marker, designating an empty remote task queue.

Once we have handled all pending steals, we drain the event queue of any other remaining events and then remove all of the matchpair entries. The event queue is drained in a non-blocking manner at this point guaranteeing that a) all MEs have been unlinked, b) busy entries have been cleared by blocking waits, c) all outstanding events have processed and the PTE is in a quiescent state. We are now free to process the new shared queue, appending MEs and new matchpair entries according to the steal-half scheme.

Stealing Tasks. In contrast to the multiple communications required in the baseline version, a steal operation can now be completed with only a single one-sided get operation as shown in Fig. 3.

Algorithm 1 Lock-free reset algorithm for *Acquire* operation

```

mp ← matchpairs.tail()
while mp do                                     ▷ attempt to unlink all MEs
  if unlink_ME(mp) ≠ success then
    busy ← busy ∪ {mp}
  end if
  mp ← mp.prev
end while

while busy ≠ ∅ do                               ▷ remove matchpair for each busy event
  ev ← EQWait()                                ▷ block for pending steal events
  if ev.mp ∈ busy then
    busy ← busy - {ev.mp}
  end if
  process_event(ev)
end while

handle_events()                                ▷ drain event queue

while ¬matchpairs.empty() do                    ▷ empty matchpair list
  matchpairs.pop_tail()
end while

```

The PtlGet() call requests a maximum of $N/4$ tasks, for a task queue of size N , which is half of the shared portion of the task queue. The actual number of tasks received can be recovered from a completion event or from within the task block if steals are setup to generate counting events.

If a steal is attempted and there is no work on the remote target, the get will match the persistent end-of-queue (EOQ) marker. If the stealing process matches the EOQ marker, there is no work remaining in the shared portion on the target process. A new target is selected, and the steal is re-attempted.

Completion Tracking. Matchlist entries are configured to allow Portals to manage the amount of consumed tasks that have been transferred due to steals. If a steal attempt is made for only k of N available tasks on an ME, then Portals automatically adjusts the matchlist entry to refer to the remaining $N - k$ tasks. Once all tasks associated with a matchlist entry have been consumed, the entry is automatically unlinked from the priority list by Portals. This allows PAWS to support both Steal-1 or Steal- N stealing mechanisms. The default mechanism is for idle workers to steal half of available work, which operates as discussed above.

Periodically, PAWS will perform a non-blocking read on the event queue for the local PTE to process Portals generated unlink events. These are correlated to entries on the matchpairs list, so the tail pointer reflects the actual tail of the task queue. This process is lightweight and does not need to be run frequently. The event handling function is run periodically or when trying to enqueue work locally and the task queue appears to be full.

4.3 Termination Detection

Since more work may exist in the system when all local tasks have been completed, processes must actively determine when all processes are idle globally and no work remains. There are many techniques for handling termination detection, from centralized counter-based schemes to fully distributed techniques.

In PAWS, we adapt the original voting-tree scheme, where a binary tree is mapped onto the process space, based on the approach proposed in [16]. When a process exhausts the task queue, the process combines the termination votes of its children and propagates the termination vote to its parent. If a process has been the target of

Figure 4: Performance and speedup of triggered vs. conventional termination detection.

a steal operation, a re-vote is called for. If the vote reaches the root without a re-vote being called, the root process signals termination down the tree, otherwise a new round of voting is started.

The PAWS implementation relies on Portals *triggered* operations to manage communication up and down the tree. Leaf processes in the tree initiate the upward pass of voting with explicit atomic operations, and the root process initiates the downward broadcast once all votes have been cast.

At the beginning of each voting round, triggered atomic operations are set for all interior nodes of the tree. Trigger thresholds are set to fire when all children have cast their vote and the current process is ready to vote for termination. This allows for the majority of the communication operations involved in termination detection to be handled transparently by the Portals runtime, without needing to explicitly check for progress and completion with the original asynchronous termination detection scheme.

Figure 4 compares the performance of the termination detection algorithm using the original non-blocking communication operations, compared against using Portals triggered operations on the experimental cluster described in Section 5. This microbenchmark attempts termination detection on an empty task pool until all processes agree to terminate. Using triggered operations results in a 2-3x performance improvement in termination detection. The impact of this improvement on overall program performance will depend on the execution time of the task pool. With shorter execution times (i.e. at scale) or with an application that repeatedly executes task queues, faster termination detection will contribute more to higher program performance.

5 EXPERIMENTAL EVALUATION

Performance results were gathered using an implementation of PAWS running on the Portals reference implementation [33].

The experiments were performed on the Comet cluster system at the San Diego Supercomputer Center with access provided through the Extreme Science and Engineering Discovery Environment (XSEDE) program. Comet contains nodes with several configurations; these experiments were conducted on a portion of the 1944 nodes comprised of Dell* PowerEdge* C6320 servers each with dual 12-core Intel® Xeon® E5-2680 processors and 128 gigabytes of memory. Nodes are connected using a Mellanox* FDR

56Gb/s InfiniBand* fabric. Comet permits jobs requesting up to 72 compute nodes (1728 cores).

5.1 Performance Baseline

One of the primary challenges in deriving performance results for this work is that the Portals 4 reference implementation was developed as an exploratory framework for modeling advanced network processing primitives that are amenable to hardware acceleration. As a result, Portals message processing performed by the reference implementation utilizes the host processor in a Portals-level companion thread and incurs higher overheads than would be expected with a hardware accelerated implementation. Thus, the performance of communication performed through the Portals 4 runtime library can be significantly lower than the performance of the underlying network when used natively. When measuring latency sensitive metrics, we dedicate an additional core per process to Portals communication processing, as described in Section 5.3.2.

In Figure 5, we establish a performance baseline by comparing the relative performance of steal operations between the baseline implementation and the Portals-native PAWS implementation with both small tasks (24 bytes) and larger tasks (192 bytes). When the volume of stolen tasks is small, we see a five-fold reduction in steal times. As the volume increases, the steal operation becomes dominated by the communication of the tasks themselves and the additional communications in the baseline version contribute less to the overall steal time.

5.2 Benchmark Applications

To further evaluate the effectiveness of the PAWS runtime system, we have chosen to run experiments with two benchmark applications: a multiresolution analysis kernel used in high-performance computational chemistry codes and an unbalanced tree search benchmark. The differences in workloads between the two applications is shown in Table 1.

5.2.1 MADNESS Tree Creation. The MADNESS [38, 42] numerical computation framework uses adaptive multiresolution methods in multiwavelet bases to solve computational chemistry problems at scale. The wavelet functions are represented using multidimensional spatial representation trees with scaling basis or wavelet

Figure 5: Steal operation times for baseline and steal-half.

basis coefficients stored at each node. A MADNESS run begins with the creation of this function tree by projecting a user-supplied analytic function onto its numerical representation, using adaptive spatial decomposition. The spatial trees are sparse structures, with larger subtrees corresponding to higher-frequency components of the supplied function.

Initially, the function is projected onto a fixed three-dimensional volume. In areas where the accuracy of the representation is insufficient, the volume is subdivided and each octant is processed recursively until the desired level of accuracy is reached. The size and shape of the task tree is dependent on the user-supplied analytic function, making MADNESS a good fit for a dynamic load balancing system. Tasks correspond to the projection of a single node in the oct-tree. Experiments were all run with a refinement threshold of 10^{-9} .

5.2.2 Unbalanced Tree Search (UTS). The Unbalanced Tree Search benchmark (UTS) [14] 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 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. Within PAWS, each 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.

Benchmark	Total Tasks	Avg. Task Time	Task Size
Madness	422,720	84.13 ms	32 bytes
UTS	270,751,679,750	0.00011 ms	48 bytes

Table 1: Benchmarking workload characteristics.

5.3 Performance Evaluation

The design of PAWS leads us to expect performance gains in several areas: faster task dissemination (or *dispersion*), reduced latency for steal operations, improved responsiveness (throughput) at the target, and faster termination.

Figure 6: Performance of MADNESS benchmark with both baseline and PAWS steal-half task queues.

Figure 7: Task dispersion latency with the MADNESS benchmark.

5.3.1 MADNESS Tree Creation. In Figure 6 we can see the performance of the MADNESS tree creation benchmark with the baseline and PAWS steal-half implementations. In terms of overall performance as measured by total runtime, there is a minimal performance improvement.

MADNESS tasks are relatively heavyweight (a tensor contraction is performed by each task), causing computation to dominate execution time. Both versions show good performance, with the PAWS implementation beating the baseline at larger processor counts, where performance becomes more sensitive to load-imbalance. At larger core counts, we observe an increase in throughput with PAWS since less work per process is available in the system. Finding work at scale requires more steal attempts, making performance more sensitive to communication overhead. Failed steal attempts require only one communication in PAWS, but three in the baseline version (steps 1, 2, 4 in Fig. 3) with the early steal aborts used in SDC. The communication optimizations employed by PAWS can contribute to the scalability of an application even with computation dominated workloads.

Looking closer, we can see evidence of these improvements in the load-balancing system. In Figure 7, we examine the *dispersion time*, i.e. the average time elapsed before each process acquires its first

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

Figure 9: Impact of dedicating cores to Portals progress threads for the UTS benchmark.

task. The lower steal attempt costs and improved responsiveness at the target allow processes to find work faster than the baseline system.

5.3.2 Unbalanced Tree Search (UTS). The performance characteristics of the UTS benchmark application are shown in Figure 8. The PAWS steal-half implementation shows a clear performance gain over the baseline in terms of task throughput, with a 5-15% improvement in overall program runtime shown in Figure 8b.

The UTS benchmark was designed to be a challenging workload for dynamic load balancing. Thus, this workload best highlights the advantages of PAWS. With its highly irregular workload, very short task duration, and large number of spawned tasks, reductions in steal overhead and improvements in task dispersion have a substantial impact on overall performance. In Figure 8c we show the parallel efficiency as the ratio of measured execution time versus the ideal execution time (sequential execution time divided by number of processes). This figure shows that, because of the communication improvements provided by PAWS, it is able to maintain high parallel efficiency even at the scaling limit of this workload.

Lastly, we look at the evaluation of software-simulated offload hardware with respect to program performance. The Portals reference implementation relies on a progress thread to perform the

offload tasks of the networking system. In Figure 9, we show the performance of the UTS benchmark using all available cores on the node – without any reserved for Portals progress threads (*none*), against dedicating a core per-process for Portals progress (*twox*). With dedicated cores for progress threads, we see higher throughput, but no substantive change in performance profile – i.e. context-switching contention impacts both work-stealing implementations equally. When measuring latency sensitive performance metrics (get latency, dispersion, and termination detection), we ran experiments with two cores per process to eliminate noise and contention effects (Figs. 4, 5, and 7).

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 network offload interface can be used to improve a dynamic work stealing system.

Our implementation reduces the communication needed for steal operations by a factor of five, leading to lower dispersion and steal overheads. Further, our approach allows the target process to specify the tasks available for stealing, rather than requiring the stealing process to issue multiple communications to determine the appropriate volume of tasks to steal.

We show using two scientific computing applications, UTS and MADNESS, that this approach is effective. We also show significant speedup in the termination detection process when using offload friendly triggered communication operations. These contributions show that offload-aware work stealing systems can lead to higher performance. More broadly, it shows how a programmable networking layer is capable of both enabling and accelerating parallel programming models important to the high performance computing community.

ACKNOWLEDGEMENT

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

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 (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 (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 Efficient 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)*. 10–10.
- [7] R. Brightwell, K. T. Pedretti, K. D. Underwood, and T. Hudson. 2006. SeaStar Interconnect: Balanced Bandwidth for Scalable Performance. *IEEE Micro* 26, 3 (May 2006), 41–57. <https://doi.org/10.1109/MM.2006.65>
- [8] 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>
- [9] 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.
- [10] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, 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.
- [11] S. Derradji, T. Palfer-Sollier, J. P. Panziera, A. Poudes, and F. W. Atos. 2015. The BXI Interconnect Architecture. In *Proc. IEEE 23rd Annual Symposium on High-Performance Interconnects*. 18–25. <https://doi.org/10.1109/HOTI.2015.15>
- [12] 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.
- [13] 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 (SC '09)*. ACM, New York, NY, USA, Article 53, 11 pages. <https://doi.org/10.1145/1654059.1654113>
- [14] 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/10.1016/j.simp.2008.06.004>
- [15] 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 (ROSS '17)*. ACM, New York, NY, USA, 4:1–4:8. <https://doi.org/10.1145/3095770.3095774>
- [16] N. Francez and M. Rodeh. 1982. Achieving Distributed Termination without Freezing. *IEEE Trans. on Software Engineering* SE-8, 3 (May 1982), 287–292.
- [17] 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.
- [18] 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)*.
- [19] 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 (PPoPP '10)*. ACM, New York, NY, USA, 341–342. <https://doi.org/10.1145/1693453.1693504>
- [20] Danny Hendler and Nir Shavit. 2002. Non-blocking steal-half work queues. In *Proc. of the 21st Symposium on Principles of Distributed Computing (PODC '02)*. ACM, New York, NY, USA, 280–289. <https://doi.org/10.1145/571825.571876>
- [21] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E. Grant, and Ron Brightwell. 2017. sPIN: High-performance Streaming Processing In the Network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 59, 16 pages. <https://doi.org/10.1145/3126908.3126970>
- [22] Antoine Kaufmann, Simon Peter, Thomas Anderson, and Arvind Krishnamurthy. 2015. FlexNIC: Rethinking Network DMA. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. <https://www.usenix.org/conference/hotos15/workshop-program/presentation/kaufmann>
- [23] 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>
- [24] Vivek Kumar, Karthik Murthy, Vivek Sarkar, and Yili Zheng. 2016. Optimized Distributed Work-Stealing. 74–77. <https://doi.org/10.1109/LA3.2016.019>
- [25] 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>
- [26] D. Brian Larkins and James Dinan. 2016. Extending a Message Passing Runtime to Support Partitioned, Global Logical Address Spaces. In *Proceedings of the First Workshop on Optimization of Communication in HPC (COM-HPC '16)*. IEEE Press, Piscataway, NJ, USA, 11–16. <https://doi.org/10.1109/COM-HPC.2016.7>
- [27] D. Brian Larkins, John Snyder, and James Dinan. 2018. Efficient Runtime Support for a Partitioned Global Logical Address Space. In *Parallel Processing (ICPP), 2018 47th International Conference on*. <https://doi.org/10.1145/3225058.3225092>
- [28] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganey, 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>
- [29] 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)*. 45–54. <http://doi.acm.org/10.1145/1504176.1504186>
- [30] MPI Forum. 2015. *MPI: A Message-Passing Interface Standard Version 3.1*. Technical Report. University of Tennessee, Knoxville.
- [31] 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
- [32] OpenSHMEM Specification Committee. 2016. OpenSHMEM Application Programming Interface, Version 1.3. <http://www.openshmem.org>.
- [33] Portals 4. 2018. Portals 4 Reference Implementation. Online: <https://github.com/Portals4/portals4>.
- [34] 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 (PPoPP '11)*. ACM, New York, NY, USA, 201–212. <https://doi.org/10.1145/1941553.1941582>
- [35] 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.
- [36] 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>
- [37] Mellanox Technologies. 2014. *MellanoX 2.1 Messaging Library User Manual*. Technical Report.
- [38] W Scott Thornton, Nicholas Vence, and Robert Harrison. 2009. Introducing the MADNESS numerical framework for petascale computing. In *Proc. Cray Users Group (CUG '09)*.
- [39] 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>
- [40] UPC Consortium. 2013. *UPC Language and Library Specifications, v1.3*. Technical Report LBNL-6623E. Lawrence Berkeley National Lab.
- [41] 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>
- [42] Takeshi Yanai, George I. Fann, Zhengting Gan, Robert J. Harrison, and Gregory Beylkin. 2004. Multiresolution quantum chemistry in multiwavelet bases: Analytic derivatives for Hartree-Fock and density functional theory. *J. Chemical Physics* 121, 7 (2004), 2866–2876. <https://doi.org/10.1063/1.1768161>

Intel and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to <http://www.intel.com/performance>.

★ Other names and brands may be claimed as the property of others.