On the Merits of Distributed Work-stealing on Selective Locality-aware Tasks

Jeeva Paudel  
Computing Sc., University of Alberta  
jeva@cs.ualberta.ca

Olivier Tardieu  
IBM T.J. Watson Research Center  
tardieu@us.ibm.com

José Nelson Amaral  
Computing Sc., University of Alberta  
amaral@cs.ualberta.ca

Abstract—Improving the performance of work-stealing load-balancing algorithms in distributed shared-memory systems is challenging. These algorithms need to overcome high costs of contention among workers, communication and remote data-references between nodes, and their impact on the locality preferences of tasks. Prior research focus on stealing from a victim that best exploits data locality, and on using special deques that minimize the contention between local and remote workers.

This work explores the selection of tasks that are favourable for migration across nodes in a distributed memory cluster, a lesser-explored dimension to distributed work-stealing. The selection of tasks is guided by the application-level task locality rather than hardware memory topology as is the norm in the literature. The prototype for the performance evaluation of these ideas is implemented in X10, a realization of the asynchronous partitioned global address space programming model. This evaluation reveals the applicability of this new approach on several real-world applications chosen from the Cowichan and the Lonestar suites.

On a cluster of 128 processors, the new work-stealing strategy demonstrates a speedup between 12% and 31% over X10’s existing scheduler. Moreover, the new strategy does not degrade the performance of any of the applications studied.

I. INTRODUCTION

Load balancing in distributed shared-memory machines running irregular parallel applications is challenging for two primary reasons: (a) statically determining a distribution of data and scheduling of tasks to ensure a balanced workload throughout the execution of the application is difficult; (b) a locality-aware placement of data and tasks is necessary to reduce the penalty of remote data access. However, locality-aware scheduling may conflict with the competing goal of load-balancing. Mainstream languages and frameworks including Cilk [1], X10 [2], PFunc [3], Java [4], Microsoft Task Parallel Library [5], and Intel Threading Building Blocks [6] extensively employ locality-aware work-stealing to address these challenges in both shared-memory and distributed shared-memory architectures.

Work stealing requires synchronization among competing workers to ensure exclusive access to task queues. Work stealing’s popularity stems from its ability to shift the burden of this synchronization to the thief, i.e. the worker without work. Such an approach minimizes the overhead on busy workers, which are the ones most likely to be performing critical-path computation. Despite this shift of burden on idle workers, improving the performance of work stealing on distributed shared-memory systems is challenging. Work stealing in large distributed systems suffers from several overheads.

(a) contention: Contention may occur if multiple thieves try to steal from the same victim, or if there is a single task in the victim’s deque.

While lock contention can be expensive in shared-memory machines, the cost is considerably higher in distributed systems and severely impacts the scalability of work stealing in distributed-memory contexts.

(b) communication: The cost of communication accrues from both the extra contention for the system bus and the latency of data transfer to a remote node.

(c) remote reference: Tasks bear affinity to the processors to which they are assigned. This affinity results from the non-uniform memory accesses of multiprocessor systems. Thus, accessing a processor’s local memory is much faster than referencing remote memory.

(d) cache pollution: Stealing a task from another processor can reduce the potential to reuse locally cached data resulting in a burst of cache misses while the cache is reloaded.

To minimize these overheads, prior work on locality-aware distributed stealing focussed on: (a) selecting the most favourable victim node for stealing a task using information such as proximity, and the workload of nodes in the cluster; (b) choosing the optimum number of tasks to transfer during a single session of stealing; (c) minimizing the contention among local and remote workers by using specialized queues for local and distributed stealing.

However, the important related question of which tasks should be stolen has received much less attention in the literature. Most existing work-stealing implementations either restrict movement of tasks only within shared-memory abstractions or renounce statically-specified locality preferences. Strategies that do use locality preference derive those preferences from hardware topology.

In distributed stealing, choosing the nearest node or worker in the memory hierarchy is useful, but the victim-node selection policy does not profoundly impact the total cost of stealing a task from a remote node.

The cost of stealing a task depends largely on the characteristics of the task being stolen including its size, the amount of data that it references, the number of sub-tasks it spawns, and the number of local memory accesses it performs. Therefore, this work focuses on deciding which task should be stolen, rather than which worker or which node must be polled first during distributed work-stealing.

Restricting the movement of tasks only within a node to

---

1 A deque is a double-ended queue. Local tasks are inserted and removed to one end of the queue with a First-In-First-Out policy, while tasks are removed from the other end of the deque for stealing operations.

2 The victim-node selection policy has greater impact if the cluster is not fully connected. For instance, in a cluster with ring topology it is a common practice to choose nearest, or adjacent nodes first.
fixate on locality preferences alone reduces the opportunities for parallelism and limits resource utilization. Similarly, permitting all tasks to be stolen across the nodes to improve the global workload distribution may incur significant overhead from loss of locality and the extra costs of remote data access. Thus, the following research questions arises:

Is there a potential for keeping the locality preferences of some tasks intact, while relaxing the locality preferences of others? How to strike a balance between load balance and locality of reference by restricting stealing across nodes?

The strategy presented in this paper indicates that identifying tasks that do not incur high communication costs or that do not bear strong affinity to processors is important to improve performance in distributed work stealing. Restricting distributed stealing to tasks that are not locality critical can compensate the underlying costs and result in a performance improvement through an improved load distribution. This new approach outperforms work-stealing implementations that stress only load-balance or only data-locality.

Its use in several applications demonstrates that this new approach is both effective and easy to use. Information about locality-flexible tasks is supplied by the programmer through a simple annotation, and it can be used by the runtime system for distributed work stealing to improve CPU utilization of nodes in a cluster. An experimental evaluation on a cluster of 128 processors results in a speedup in the range of 12% – 31% against a state-of-the-art load balancing approach. The proposed approach to load balancing does not degrade performance on any of the applications used in the experimental evaluation.

II. Selection of Tasks for Distributed Stealing

All tasks are not equally favourable for distributed stealing. Rather, some tasks are more suited to stealing than others. The following task model identifies tasks that reduce the stealing cost. A task qualifies for distributed steal by a thief processor if the task meets any of the following conditions. (a) The processor’s cache is warm for that task because the related data is already in its local memory. Therefore, additive cost does not need to be paid for reloading a cold cache, or to communicate with the remote processor to copy the data. (b) The task is local to the thief processor because it is spawned by some task originally stolen by the thief processor. Thus, no extra cost needs to be paid. (c) The task’s granularity is large enough to overcome the cost of stealing by keeping the thief node busy for sufficient time to prevent repeated and frequent steals. (d) The task encapsulates the data necessary for its computation.

A task that qualifies for distributed steal is a locality-flexible task. A task that bears strong affinity to a processor is a locality-sensitive task. Such a differentiation leads to a two-fold advantage. (a) The run-time needs to manage fewer parallel tasks (locality-flexible ones), thus reducing the bookkeeping and remote-stealing overheads because only selected few tasks are available to be stolen remotely. (b) Tasks that bear strong affinity to a processor, and thus incur large overheads on a stealing operation, are excluded from participating in the stealing process.

Surprisingly, there is no previous work that classify tasks in terms of their sensitivity or flexibility to locality in order to make the flexible ones available for distributed work stealing while restricting the movement of sensitive ones to their shared-memory partition. Thus, a principal contribution of this work is the underpinning of the distributed work-stealing strategy on task properties and their computing needs rather than hardware topology.

The attributes — such as critical path, remote data-access overheads, and task granularities — that characterize the locality-flexibility of a task can be obtained in multiple ways. They can be derived a priori through static analyses, or can be computed on the fly as the program is executing. Identifying tasks that possess the locality-flexible properties may also require semantic and algorithmic details that are often beyond the scope of the static analyses performed by an optimizing compiler. However such information is often readily available to a programmer. In such cases, the information can be supplied by programmers through static code annotations.

Partitioned Global Address Space (PGAS) programming languages, such as X10 and Chapel, already offer abstractions such as places and locales to express locality preferences among a set of tasks. Programmers already need to possess a high-level overview of how computations in an application unfold, how degrees of parallelism change, and how the data access patterns of dynamically spawned tasks change. Therefore, identifying locality-flexible tasks requires only an incremental reasoning about an application because the task properties are typically visible in an application’s algorithm, whose understanding is essential to all programmers.

The key to our approach is conveying which tasks are locality-flexible and which ones are locality-sensitive to the runtime system, regardless of how this classification is obtained. The first implementation of this idea presented in this paper relies on programmer-specified locality hints to identify locality-flexible tasks. This approach requires minimal changes to the X10 programming system.

III. Background

The X10 language specification [13] provides a detailed description of X10. This section introduces only the key concepts that are necessary to understand the rest of the paper.

Activities and places are two main concepts in X10. Every computation in X10 is an asynchronous activity, akin to a lightweight task. A place is an abstraction intended to enable the encoding of affinity between tasks and memory partitions and therefore it also induces the notion of locality. Every activity runs in a place. X10 provides the statement async (p) S to create a new activity at place p to execute S. The activities running in a place may access data located at that place with the efficiency of local access. An access to a remote place may take orders of magnitude longer and is performed using the at (p) S statement. An at statement shifts the control of execution of the current activity from the current place to place p, copies any data that is required by the statements S to p, and, at the end, returns the control of execution to the original place. The necessary data copying is done through runtime system calls inserted by the compiler.

X10’s system of places is designed to make the local and remote accesses obvious. As a result, programmers are aware of the places of their data, and know when they are incurring communication costs. Thus, they can identify which tasks will
encapsulated by different triangles, and the number of re-
throughout the mesh-generation process. The number of points
among the nodes does not guarantee a balanced workload
node is crucial to minimize remote memory references.

A candidate compiler

X10 is a natural fit to the work-stealing model employing
selective locality-aware tasks. First, the ability to create threads
to perform computations at a specified remote node facilitates
the implementation of distributed work stealing. Second, X10’s
semantics that offers a precise view of local and remote data
access enables programmers to identify the locality-flexible
tasks. The semantics also allows control over converting local
data access into remote data access. This conversion may be
needed after a task is stolen by a remote node, and the data
is no longer local to the task. Therefore, we use X10 as a
candidate compiler to implement our ideas and to investigate
the applicability and benefits of our approach.

X10 uses ahead-of-time information about the destination
for launching a task because selecting the destination at run-
time would be expensive and delay the execution of tasks even
after their creation. Thus, using static analysis or annotating
the code to inform the runtime about which tasks may be stolen
is more appropriate in the context of X10. Using profiling or
runtime analysis to make this decision would be expensive.

IV. EXAMPLES

Two examples will illustrate the identification of tasks that
bear locality-flexibility.

A. Delaunay Mesh Generation

A Delaunay mesh generator creates a mesh of Delaunay
triangles from a given set of points. The mesh generation
algorithm starts by initializing a work-list with the points to
be processed, and by initializing the mesh with a single large
triangle encompassing all the points. A triangulation procedure
then picks a new point from the work-list, determines the
triangle containing the point, creates a cavity by splitting
the triangle into three new triangles that share the point and
performs re-triangulation until all the new triangles are valid
Delaunay triangles. Initially, there is only one large triangle.
However, as the triangulation process unfolds, the mesh gets
populated with several new triangles. The points in these newly
formed triangles can be processed in parallel because the final
mesh generated is the same regardless of the order in which the
points are processed. Thus, the intermediate triangles can be
distributed among different nodes for parallel and distributed
processing. The triangles allocated in each node create other
Delaunay triangles using the points they enclose. Therefore,
allocating triangles and their encapsulated points at the same
node is crucial to minimize remote memory references.

An even distribution of the initially generated triangles
among the nodes does not guarantee a balanced workload
throughout the mesh-generation process. The number of points
encapsulated by different triangles, and the number of re-
triangulations needed for processing different points could be
different. As a result, some nodes might have to perform more
triangulations than others. Therefore, a natural question arises:
If a node is idle after processing all the points in its work-
list, then would it be beneficial, performance-wise, to steal a
triangle with unprocessed points from other nodes?

The triangulation procedure is a task with the following
properties: (i) it encapsulates all the data necessary for its
computation, eliminating the need for repeated remote memory
references; (ii) copying of the triangle and its point from the
victim to the thief is necessary only once; (iii) all the new
triangles created by the thief have local access to other points
in the triangle because they are already copied to the thief’s
local memory; and (iv) it is coarse enough to keep the idle node
busy for sufficient time because it can spawn several new tasks
to process its points in parallel and utilize multiple cores and
processors in a node. Therefore, such a task is locality-flexible
and can be stolen by a remote node to improve parallelism if
there is any idle node searching for work.

B. Turing Ring

The Turing ring problem simulates the interaction between
predators and preys in a ring of cells. The algorithm (shown in
Figure 1) initializes each cell with a number of predators and
preys and their cell IDs, and evenly distributes the ring across
nodes. In each iteration, the algorithm updates the predator
and prey populations accounting for their death, birth and
migration. Migration can change the workload in cells by as
much as two orders of magnitude in a single iteration resulting
in load imbalance. There are two types of tasks that can be
executed remotely to adjust the load imbalance.

First, if the task that updates predator or prey populations
(lines 5 or 6) is stolen, information about all predators/preys
in the cell including their birth, death and migration rates
must be copied to the thief to compute the new population.
The new population must then be copied back to the victim
cell because the updated population is required to perform re-
distribution of bodies. Second, if the outer task that performs
all operations in a cell, including population update and
migration of predators and preys, is stolen then the entire cell
needs to be copied to the thief node. However, once the cell is
copied, there is no need to copy the results back to the victim
cell because all other operations (lines 5 – 8) on the cell
are now local to the thief. Hence, no more remote memory
references will be needed. Further, the stolen task makes work
available for other co-located workers in the thief node. Thus,
the outer async that processes an entire cell is a locality-
flexible task that is suitable for remote stealing.

A central idea of this paper is that the programmer can
annotate such tasks so that the runtime scheduler can treat their

Fig. 1: Pseudo code for Turing Ring.
mapping to a place in a special manner, making them available for distributed stealing operations. As long as there is no need for any load-balancing operation, such tasks will execute in the place specified in their `async’s` definition. However, if all workers in the specified place are busy, while a worker in any other place is looking for work, then the `async` will be available for that remote worker to steal.

V. DESIGN AND ALGORITHM

Many processor architectures must rely on software to implement work-stealing because they lack support at the hardware level [14]. Steal operations implemented in software can severely interrupt the execution of local workers. A steal operation on a remote deque requires synchronization among multiple workers, both those local to the thief node and those from remote nodes, for exclusive access to the victim’s deque. As a result, a local worker might end up waiting for thousands of cycles while a remote process repeatedly tries to access the deque or while several remote processes manipulate its deque. Such a synchronization requirement reduces the concurrency potential of an application and also adversely affects the performance of the local process. Thus, the time to execute a given set of tasks depends not only on the tasks being executed, but also on the number of interruptions due to steal requests from both local and remote workers.

A. Multiple Deques

The key to scaling work-stealing implementations to large distributed-memory systems is to use the data structure that minimizes locking on the critical path. Mapping both locality-sensitive and locality-flexible tasks to per-worker deques can lead to frequent interruptions to the operation of local workers because of frequent steal requests from remote nodes. The solution is to maintain separate deques for locality-flexible and locality-sensitive tasks. Using multiple deques or specialized deques to reduce contention is a common practice and has been investigated before [8]. Seung et al. use a global task queue that provides public and private views of tasks to reduce contention. Tasks are stolen only from the public region of the global queue, which requires serialized access using locks. Access to the private portion of the task queue do not require locking. Acar et al. use private deques that permit only non-concurrent private operations on each worker’s deque, and rely on explicit communication for load balancing [11]. Earlier, Acar et al. implement a non-blocking locality-aware work-stealing algorithm using deques and mailboxes [12]. Upon creation, a thread is pushed into both the deque and the tail of the mailbox, which is a FIFO queue, of the process to which the thread bears affinity. A process first tries to obtain work from its mailbox before attempting a steal from the deqeu.

Figure 2 shows the block diagram of a place with different deques used by the new algorithm presented in this paper to map locality-flexible and locality-sensitive tasks. Each place has a shared deque and multiple private deques corresponding to each worker thread within a shared memory partition. Tasks that bear affinity to a processor are mapped to its private deques, while locality-flexible tasks are mapped to the shared deque. Under this scheme, each worker can operate on its private deque without locking. Locking is only required when accessing the shared deque, which is the one accessed by remote steal requests. The LIPO access policy of a worker’s private deque leads the local worker to execute the most recently created task and thus offers a higher chance of exploiting cache locality.

The shared deque, in contrast, is manipulated in a first-in-first-out (FIFO) manner to ensure that any steal operation, whether local or remote, receives the oldest task in the deque. Older tasks potentially contain the largest amount of work in the task graph leading thieves to remain busy for a longer period of time and reducing the overhead of frequent steals.

B. Algorithm

X10’s runtime scheduler’s initial task-mapping strategy must be modified to facilitate distributed stealing of locality-flexible tasks and to ensure that locality-sensitive tasks execute in programmer-specified places.

1) Task Mapping: Any `async` activity with high affinity to a program-specified place is directly mapped to a worker’s private deqeu within that place.

Typically, the algorithm maps locality-flexible tasks to shared deques. However, such an approach can be counterproductive when an application consists of significantly more locality-flexible tasks than locality-sensitive tasks. In such a scenario, mapping all locality-flexible tasks to shared deques may leave workers within a place under-utilized or even idle for lack of work originated in locality-sensitive tasks. Even worse, a remote worker from place $p1$ may steal a task from place $p2$ even when the workers in $p2$ are idle or under-utilized.

A place is idle if it has no running activities, i.e., all the workers are either suspended, stopped, or are searching for work. The workers searching for pending activities are continuously probing the network, even if there is no user code in the network, until they receive a termination signal from the root place. A place is under-utilized if it has room for additional parallel computation. X10 programs start with a fixed number of static threads and then spawn dynamic threads. Until an upper bound on the total number of threads allowed in a place is reached, additional dynamic threads can be created to increase parallelism.

Mapping locality-flexible tasks to an idle, or under-utilized place’s shared deqeu not only limits potential parallelism, but also incurs unnecessary contention overhead on workers by forcing them to steal work either from their own shared deque or from a remote place’s shared deque. Thus, on idle or under-utilized places, the new task-mapping algorithm maps a locality-flexible task to a private deque. On fully-utilized places a locality-flexible task is mapped to the shared deque. Mapping tasks in this manner has two benefits: (i) it prioritizes the
utilization of all available cores in a processor before attempting to utilize other remote processors; (ii) it eliminates the cost of unwarranted steal operations because mapping a task, regardless of its locality property, directly to an idle worker eliminates the need for that worker to contend with other co-located idle workers to steal from the local shared deque. The algorithm (lines 1 - 8, Algorithm 1) uses a combination of data-located idle workers to steal from the local shared deque. The elimination of the need for that worker to contend with other co-located idle workers to steal from the local shared deque is studied. An empirical evaluation that indicated that stealing multiple tasks did not result in performance improvement for the applications studied.

2) Stealing: Load imbalances within a place are managed first by the local work-stealing scheduler. If the local work-stealer fails to find an activity to execute for lack of surplus work among the peer workers or the shared deque within the place, then the scheduler starts distributed stealing attempts.

Lines 9 - 29 in Algorithm 1 show the control logic for local and distributed work-stealing. Under distributed stealing, a thief thread holds a lock on the remote shared deque, retrieves an activity, creates a closure using the stolen activity, marks this closure for remote execution, and creates a new async activity at its home place to execute the closure. In case of a failed distributed steal, the thief first probes the network to see if any remote task has spawned tasks at its home place before continuing to explore other places. This approach eliminates unnecessary steal operations and also prevents polluting other place’s caches. The thief continues this process until it finds work or until it has explored all available places.

3) Stealing Multiple Tasks at Once: Prior research has shown that the number of tasks stolen during a single steal-attempt can significantly impact the performance of the work-stealing algorithms. Olivier et al. use StealHalf policy in which thieves steal one-half of the victim’s deque, thereby reducing the total number of steals [10].

An empirical evaluation of the applications in this study indicated that good performance is achieved for both structured and bursty/unstructured task graphs when performing distributed stealing in chunk sizes of 2. With the new algorithm described in this paper a request for stealing only happens when: (i) all the other co-located workers are either busy executing their own activity and do not posses surplus work to be stolen, or are themselves attempting to steal work; and, (ii) the local shared deque is empty. Thus, stealing tasks in chunks not only offers work for the thief thread, but also prevents the need for other peer workers in the thief’s place to perform distributed steal by making work available locally. Within a place, single tasks are stolen locally based on an empirical evaluation that indicated that stealing multiple tasks did not result in performance improvement for the applications studied.

VI. IMPLEMENTATION

This section describes the compiler and runtime support needed to implement the task-mapping facility for locality-sensitive and locality-flexible tasks, and to implement the cross-place work-stealing facility.

A. Compiler Support

We use @AnyPlaceTask annotation to specify locality-flexible tasks as: @AnyPlaceTask async(p) S.

B. Runtime Support

The X10 runtime converts the APGAS constructs into calls into a private API called the X10 Runtime in X10 (XRX). XRX supports both Java and C++ code generation. The new locality-aware stealing algorithm was implemented in the XRX to enable the new runtime task-mapping and work-stealing mechanisms to be plugged into both the C++ and Java backend compilers and run-times for X10.

The scheduler creates an object at each place to maintain information that helps to identify idle or lightly-loaded places. There is no need to synchronize access to this object from different places because the object is local to each place and it is only changed by its co-located workers. The scheduler accesses these objects using PlaceLocalHandle

Assignments to active must be atomic because there are two different sections in the X10 runtime code that can modify the status of a place. The first is after an activity is assigned to the place. Second, active must be set to false, to indicate that the place is idle, in a place pi after n successive failed attempts

\[\text{A PlaceLocalHandle is a unique identifier that resolves to a unique local piece of storage at each Place.}\]
to steal, where \( n \) is the number of worker threads per place. If a place has several tasks to be processed, an idle worker is likely to succeed in stealing work from one of the other \( n-1 \) workers in its place in \( n \) steal attempts. Otherwise, it is highly likely that the other workers are busy processing a single lengthy job or that they themselves are out of work. In any case, the place is a candidate for mapping of any newly arriving asynchronous task directly into a worker’s private deque.

VII. EXPERIMENTAL SETUP

a) Platform: Performance measurements use a blade server with 16 nodes, each featuring two 2 GHz Quad-Core AMD Opteron processors, with 8 GB of RAM and 20 GB of swap space, running CentOS GNU/Linux version 6.

b) Runtime: The nodes in the cluster are connected by an infiniband network with a bandwidth of 10 Gbit/s and use MVAPICH2 library for communication. The experimental runs set X10_NTHREADS=8 to create eight worker threads per place and vary the number of places from 1 to 16 so that the number of threads is the same as the total number of cores.

c) Compiler: The x10c++ compiler version 2.2 is used for all measurements and the command line arguments –O -NO_CHECKS are passed to the compiler to enable optimizations and disable array bounds, null pointer, and place checking. The NO_CHECKS option is used for performance assessment only after a program compiles successfully in the experimental platform.

d) Benchmarks: We chose applications with unstructured data and irregular parallelism for experimental evaluation because they present significant opportunities for load balancing. These include the following applications, from the Cowichan suite [15]: (a) Quicksort sorts an array of 100M elements using quick-sort (b) Turing ring solves a set of coupled differential equations modelling system dynamics using 1M bodies (c) k-Means implements a k-means clustering algorithm resulting in four clusters and using 1000 iterations (d) n-Body simulates the forces acting on a system of 220K bodies using the Barnes-Hut algorithm.

The experimental evaluations also use the following three applications from the Lonestar suite [16], that we ported from the Galois framework to the X10 language: (e) Agglomerative clustering performs clustering of 2M points by building a hierarchical tree in a bottom-up manner (f) Delaunay mesh generation (DMG) deals with 2D Delaunay triangular mesh generation using 80,000 points (g) Delaunay mesh refiner (DMR) refines a Delaunay mesh of 550K triangles such that no angle in the mesh is less than 30 degrees.

The steals-to-task ratios for the benchmarks are low — between 1e-04 and 1e-05 as shown in Figure 3. However, the number of steal operations is significant indicating that these benchmarks are suited for the evaluation of the new algorithm.

VIII. EXPERIMENTAL EVALUATION

A central idea in this work is the restriction of distributed work-stealing to locality-aware tasks. The key to the selection of tasks is their granularity and their sensitivity to locality. Thus, this experimental evaluation addresses the following questions: (1) How does thread and node count impact the speedup obtained using DistWS? (2) How does the amount of work stolen by a request affect application performance using DistWS? (3) Is the distinction between locality-sensitive and locality-flexible tasks really necessary?

1. How does thread and node count impact speedup?

For each application, Figure 3 shows the speedups obtained using the X10’s scheduler (X10WS) and our distributed work-stealer (DistWS). The speedups are relative to the sequential execution time (shown in Figure 4) obtained by running the sequential implementations of the applications. The numbers report the average of ten executions.

The applications exhibit two common trends. First, execution over a single node (using 1 to 8 workers) results in slowdown in comparison to X10WS because there are no opportunities for cross-node steals. However, DistWS incurs extra costs to maintain separate deques, schedule locality-flexible and locality-sensitive tasks separately, and explore the runtime load-status to decide on a deque for the locality-flexible tasks. Using multiple nodes provides opportunities for cross-node steals and better node utilization, resulting in performance improvement over X10WS.

Second, DistWS exhibits larger impact for higher number of workers because they process, create or destroy work items — such as triangles, clusters and points — at a faster and a varied rate resulting in an imbalanced workload across different nodes. Up to 16 workers, the maximum difference in speedup between X10WS and DistWS is only around 7%. Beyond 16 workers, the margin of speedup is larger: for DMG the margin is 13% (at 64 workers); for DMR, it is 11% (at 128 workers); and for n-body, it is 12% (at 128 workers). The best performance improvement over X10WS for these applications are respectively: 31%, 27%, and 19%.

2. How does the amount of work stolen by a request...
Fig. 5: Speedups over Sequential Execution Time using X10WS and DistWS. The taller bars are better. The black bars are consistently taller than the grey bars (for more than 1 node) indicating the performance benefits of DistWS.

### TABLE I: Task granularities (in ms).

<table>
<thead>
<tr>
<th>Application</th>
<th>Qsort</th>
<th>Turing</th>
<th>k-Means</th>
<th>Agglomer.</th>
<th>DMG</th>
<th>DMR</th>
<th>n-Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1.1</td>
<td>1.86</td>
<td>3.83</td>
<td>5.29</td>
<td>7.32</td>
<td>8.93</td>
<td>6.23</td>
</tr>
</tbody>
</table>

affect application performance using DistWS? The amount of work stolen by a successful request is determined by the granularity of the tasks stolen, measured as the time that it takes for a single thread to execute that task, and by the number and granularity of tasks spawned by the stolen tasks. Task granularity affects the performance of DistWS by influencing how long a thief node will perform useful work before requiring another steal. Fine-grained tasks offer better scheduling freedom, but they do not contain enough work to keep a thief node’s multiple workers busy for long enough to preclude frequent steal attempts. A separate experimental study used smaller applications, namely: merge sort, skyline matrix multiplication, Monte-Carlo estimation of π, matrix chain multiplication, and random access with task granularities of 0.12 ms, 0.93 ms, 0.005 ms, 0.09 ms and 0.006 ms, respectively. These granularities are substantially smaller than in the applications used in the experimental evaluation (shown in Table I). The DistWS algorithm performed worse on these smaller applications, thus supporting the claim that only tasks that perform significant computation are suitable candidates for distributed work stealing.

3. Is the distinction between locality-sensitive and locality-flexible tasks really necessary? DistWS-NS is a non-selective version of DistWS that allows any task to be stolen. A naïve way to implement DistWS-NS would be to annotate all tasks in the application with @AnyPlaceTask and then map all tasks to the shared deque in their assigned place. However, such an approach would incur high overhead because even local workers will need to retrieve tasks from the shared deque by competing with both co-located and remote workers, instead of directly retrieving tasks from their private deques. Therefore, for a fair comparison, DistWS-NS maintains private deques, but maps tasks among the private and shared deques in a round robin fashion, so that there are opportunities for both local and remote execution of tasks.

As seen in Figure 6, DistWS performs considerably better than DistWS-NS. The reasons are threefold: First, X10’s place abstraction helps maximize locality and data reuse across tasks by executing a set of tasks on cores that share one or more levels of cache. Stealing a random task from a remote node disrupts the memory access pattern of tasks in the victim, and, in the worst case, may require a transfer of the whole content of the victim’s cache to the thief processor. Consequently, the L1 data cache miss rates are higher for DistWS-NS compared to that of DistWS, as shown in Table II.

### TABLE II: L1 Data Cache Miss Rates(in %) for 128 workers.

<table>
<thead>
<tr>
<th>Applications</th>
<th>X10WS</th>
<th>DistWS-NS</th>
<th>DistWS</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickSort</td>
<td>1.7</td>
<td>4.1</td>
<td>2.2</td>
</tr>
<tr>
<td>Turing Ring</td>
<td>1.9</td>
<td>3.5</td>
<td>2.3</td>
</tr>
<tr>
<td>k-Means</td>
<td>2.1</td>
<td>5.6</td>
<td>3.0</td>
</tr>
<tr>
<td>Agglomer</td>
<td>6.0</td>
<td>10.9</td>
<td>7.1</td>
</tr>
<tr>
<td>DMG</td>
<td>41.1</td>
<td>46.3</td>
<td>42.3</td>
</tr>
<tr>
<td>DMR</td>
<td>31.0</td>
<td>37.7</td>
<td>33.6</td>
</tr>
<tr>
<td>n-Body</td>
<td>14.0</td>
<td>21.0</td>
<td>16.0</td>
</tr>
</tbody>
</table>

Second, DistWS strives to migrate only tasks that require minimum data transfer between the victim and thief nodes or the tasks that do not require frequent access to the remote data. DistWS-NS, in contrast, chooses tasks for migration without such considerations. As a result, DistWS-NS transmits a significantly larger amount of data across the nodes than DistWS, as shown in Table III.
TABLE III: Number of messages transmitted across nodes (for 128 workers).

<table>
<thead>
<tr>
<th>Applications</th>
<th>X10WS</th>
<th>DistWS-NS</th>
<th>DistWS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>5,349,730</td>
<td>8,196,604</td>
<td>6,943,568</td>
</tr>
<tr>
<td>Turing Ring</td>
<td>4,192,734</td>
<td>7,895,344</td>
<td>6,424,840</td>
</tr>
<tr>
<td>k-Means</td>
<td>9,540,830</td>
<td>12,375,106</td>
<td>11,800,547</td>
</tr>
<tr>
<td>Agglomer</td>
<td>8,996,422</td>
<td>12,430,790</td>
<td>11,800,547</td>
</tr>
<tr>
<td>DMR</td>
<td>34,143,024</td>
<td>42,689,149</td>
<td>37,923,541</td>
</tr>
<tr>
<td>n-Body</td>
<td>28,582,822</td>
<td>39,880,036</td>
<td>32,892,145</td>
</tr>
<tr>
<td></td>
<td>15,655,429</td>
<td>21,938,135</td>
<td>18,289,203</td>
</tr>
</tbody>
</table>

Fig. 6: Speedups for work stealing algorithms (128 workers).

Third, DistWS leads to a higher, and more uniform, CPU utilization across all nodes in comparison to DistWS-NS. Figure 7 shows that X10WS yields highly disproportionate node utilization. On average, the applications suffer from nearly 35% disparity in node utilizations. The differences arise because of the irregular nature of parallelism and the lack of opportunities for inter-node load balancing.

Compared to X10WS, the node utilizations with DistWS-NS and Dist-WS are higher because of extra workload from stealing and related operations, such as remote data access and data movement across the network. However, the disparities in CPU utilization with DistWS-NS are still high as shown for the Quicksort application. Other applications exhibit similarly large difference in node utilization, indicating an imbalanced workload. For the sake of clarity, the DistWS-NS node utilization for other applications is not shown.

With DistWS, the average variance in node utilization decreases to 13%, indicating that DistWS improves load balance. DistWS also achieves the highest node utilization compared to X10WS and DistWS-NS. However, the increased utilization does not directly translate into performance improvement because the CPU must do extra work for distributed steals and the associated overheads. Thus, most likely the overall performance gain is the result of more uniform node utilizations rather than an increase in the node utilization itself.

In conclusion, DistWS-NS’s failure to distinguish between tasks that can compensate for the costs of critical path, remote communication, and data copying between the nodes means higher L1 D cache misses, a higher number of data transfer between the nodes, and a highly disproportionate utilization of nodes. Consequently, DistWS leads to a significantly better overall performance compared to DistWS-NS.

IX. LIMITATION

In X10, final variables are globally accessible, and can be directly accessed by an async in any place. However, access to non-final fields is permitted only for objects residing at the same place as the asyncs. For instance, the code in Figure 8 creates a distributed array of one hundred elements distributed over all places via a block distribution with each element initialized to 0. The access to distArray within the async statement in lines 11 and 15 of Figure 8 are valid for place p1 because the array elements and the async task are co-located at place p1. If the async activity is launched at a remote place, say p2, after a remote steal, then those accesses will no longer be valid. Fortunately, the X10’s type system checks and identifies such non-local data accesses that may occur as a result of migration. An easy way to ensure that data accesses are local even after migration is to explicitly type-cast the array accesses using the at construct as shown in lines 12 and 16.

X. RELATED WORK

DistWS differs from prior locality-aware distributed work-stealing algorithms in the selection of tasks for distributed stealing, and the locality information used to guide the stealing.

A. Task Selection for Distributed Stealing

The COOL programming model allows programmers to specify locality hints for scheduling tasks and distributing objects [17]. Unfortunately, the work stealer employs randomized work-stealing, thereby, forsaking the programmer-specified locality preferences. The Scioto framework also categorizes tasks based on their affinity to processors and encourages distributed steals of low-affinity tasks [18]. However, it allows stealing of high-affinity tasks as well, which may interfere with programmer-specified locality preferences. Unlike COOL and Scioto, DistWS guarantees that the programmer-specified locality preferences are honoured, unless they are explicitly marked as being flexible for launching in any other place.

The Legion programming model enables user-guided placement of tasks [19]. However, its runtime requires that all tasks in a region be stolen if any task in the region is stolen. In

Fig. 8: An example to illustrate the use of at statements to make data-access local after a task is stolen by a remote node.

```java
1. val arrayReg: Region = (1..100); 2. val arrayDist: Dist = Dist.makeBlock(arrayReg); 3. val distArray: DistArray[Int] = DistArray.make[Int] (4. (arrayDist, (p):Point) => 0); 5. final val incr = incrVal; 6. finish { 7. for (p in arrayReg) { 8. val p1 = arrayDist(p); val p2 = p.next(); 9. async (p1) { 10. // async (p2) { 11. val newVal = distArray(p) + x; 12. // val newVal = at(p1) distArray(p) + x; 13. Console.OUT.println("array val at " + p.toString() + " " + distArray(p)); 14. // + at(p1) distArray(p)); 15. 16. }
```

Even though there is no ordering among the nodes in Figure 7 a line graph is more effective to visualize the load variation than a bar graph.
B. Task-level locality vs. memory-hierarchy-based locality

Yan et al. introduce hierarchical place trees to facilitate locality-aware movement of data across different levels of memory in the hardware [9]. However, they do not implement the runtime mechanisms required to support work stealing.

HotSLAW uses hardware topology information to steal from the nearest victim in the memory hierarchy [8]. In contrast, DistWS uses application-level locality to guide work stealing. DistWS uses unbounded private and shared queues to distinguish between locality-sensitive and locality-flexible tasks. HotSLAW also splits a bounded per-thread queue into private and public regions to support distributed stealing, but it does not distinguish tasks as locality-sensitive and locality-flexible. DistWS allows distributed stealing of only those tasks that are locality-flexible. HotSLAW switches between work-first and help-first policies depending on the occupancy of per-worker deques. DistWS uses a help-first policy and its tasking model abides by the locality preferences of tasks rather than the size or occupancy of per-thread queues.

Similar to HotSLAW, SLAW also uses both work-first and help-first work-stealing policies and adopts one of them based on the actual stealing rate [22]. SLAW is intended for the SPMD programming model only, unlike DistWS, which supports MIMD as well. Cong et al. and Guo et al. investigate approaches to combine work-first and help-first stealing policies [23, 24]. Unlike these policies, DistWS is based on a pure help-first scheduling policy and uses application-level locality information instead of hardware memory hierarchy.

Saraswat et al. introduce lifeline-graph-based load-balancing [25], and demonstrate significant performance benefits on the Unbalanced Tree Search (UTS) benchmark [26]. A preliminary investigation indicates that their approach outperforms our DistWS algorithm on UTS, which is expected because they use a dedicated two-step load balancer for UTS. First, it performs random stealing, followed by a more organized lifeline-based load balancing. When a node fails to steal, it quiesces and informs the outgoing edges in the lifeline graph. Work arrives from a lifeline and is pushed by the nodes onto all their active outgoing lifelines. In randomized work-stealing, a missed steal does not help future steals. By remaining in quiescent state after a failed steal and also by informing other nodes in its outgoing edges, lifeline graphs help reduce the impact of missed steals on performance. Unfortunately, the authors do not consider how lifeline graphs could be made locality-aware. When we disable the lifeline-based load balancing, DistWS achieves a 9% speedup over the randomized stealing approach at 128 workers. Even when there is no need to selectively specify tasks for stealing, DistWS does not incur any overhead on the UTS problem, indicating...
the benefits and applicability of DistWS to problems where all tasks are locality-flexible.

Majo and Gross’s permit migration of tasks across nodes to reduce cache contention [27]. DistWS permits migrations of tasks that keep the thief sufficiently busy and that incur small penalty of remote data access. DistWS guides selection of such tasks by analyzing task properties, such as task granularity, and number of remote memory references. However, their work relies on previous runs of the program to identify favourable tasks. Further, unlike DistWS, Majo and Gross do not consider parallel workloads.

XI. Summary

This paper assessed the impact of employing distributed work-stealing on selective locality-aware tasks. It described the implementation of (DistWS) in X10, and provided support for programmers to distinguish between locality-flexible and locality-sensitive tasks. The private deque in a node offers a local view of the collection of locality-sensitive tasks, while the shared deque provides a global view of the collection of locality-flexible tasks. To exploit locality in distributed-memory multicore clusters, an idle worker first attempts to steal from other co-located workers, then from the local shared deque, and finally from the remote shared deque.

DistWS resulted in a speedup between 12% to 31% on applications from the Cowichan and Lonestar suites. The evaluation also revealed that some applications are amenable to higher performance improvement with DistWS than others. Examples include Delaunay mesh refinement, Delaunay mesh generation, n-Body Barnes Hut, Agglomerative clustering and k-Means clustering problems that possess coarser-grained tasks and irregular dynamic parallelism. Even applications that are not best suited to DistWS exhibit a marginal performance improvement. Overall, DistWS does not lead to performance degradation in any of the applications, indicating its applicability to a wide range of applications.

ACKNOWLEDGMENT

This research has been supported and partially funded by Alberta Innovates Technology Futures Scholarship, Natural Sciences and Engineering Research Council of Canada, and IBM Center for Advanced Studies.

REFERENCES


