Hybrid Parallel Task Placement in X10

Jeeva Paudel
Computing Sc., University of Alberta
jeeva@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

This paper presents a hybrid parallel task-placement strategy that combines work stealing and work dealing to improve workload distribution across nodes in distributed shared-memory machines. Existing work-dealing-based load balancers suffer from large performance penalties resulting from excessive task migration and from excessive communication among the nodes to determine the target node for a migrated task. This work employs a simple heuristic to determine the load status of a node and also to detect a good target for migration of tasks.

Experimental evaluations on applications chosen from the Cowichan and Lonestar suites demonstrate a speedup, with the proposed approach, in the range of 2% to 16% on a cluster of 128 cores over the state-of-the-art work-stealing scheduler.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features - Concurrent Programming Structures; D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Design, Performance, Algorithms

Keywords Place flexible, Async, Parallel Programming, X10

1. Introduction

Load balancing in distributed shared-memory machines running irregular parallel applications is challenging for two major reasons: (a) statically determining the optimal distribution of data and scheduling of tasks that ensures a balanced workload throughout the execution of the application is difficult; (b) the penalty of remote data access necessitates locality-aware placement of data and tasks. However, locality-aware scheduling may conflict with the competing goal of load-balancing. Mainstream languages and frameworks including Cilk [9], X10 [7], PFunc [11], Java [14], Microsoft Task Parallel Library [15], and Intel Threading Building Blocks [17] extensively employ locality-aware work-stealing [1, 2, 4, 5, 10] to address these challenges in both shared-memory and distributed shared-memory architectures.

Work stealing requires synchronization among competing workers through 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 its success, work-stealing is known to incur performance penalties arising both from contention among workers and from the need to copy data and results between the thief and the victim.

Another popular approach to load balancing is *work dealing*, where tasks are distributed to different nodes, as they are generated, based on a pre-determined work-distribution policy. The challenges in work dealing are to determine when to migrate tasks between nodes, to find the best node to launch the tasks, to select the most favourable tasks for migration, and to do so with minimum overhead.

Existing work-dealing approaches collect detailed information about the system state and the behaviour of the workload to check if there is a load imbalance in the system and to pick the target node for migration of tasks [3, 8, 13]. Collecting such information may require system-wide state exploration and excessive communication between the nodes, which may negate the benefits of an improved work-load distribution. Further, work dealing may disrupt the locality preferences of tasks because they will be launched at the underloaded nodes rather than the nodes to which the tasks bear affinity. Thus, the effectiveness of work dealing depends on efficiently identifying load imbalance in the system and carefully choosing tasks for migration to the underloaded nodes.

The 2013 ACM SIGPLAN X10 Workshop (X10'2013)
June 20, Seattle, Washington, USA.
Copyright © 2013 ACM 978-1-4503-2157-0/13/06. . . $15.00

2. Overview

Programming languages based on the Asynchronous Partitioned Global Address Space (APGAS) memory model [19] are a natural fit to the work-dealing model of load balancing. They offer programmers abstractions to specify where a dynamically spawned task must be launched and also, offer mechanisms to create light-weight threads at the desired node to execute the task asynchronously. Such an ability to create threads to perform computations at a specified remote node facilitates the implementation of work dealing.

X10 is a realization of the APGAS model on top of a Java-like core sequential language and is gaining widespread popularity as a high-performance programming language. X10 primarily relies on work stealing within its shared-memory abstraction for load balancing. Existing implementations of the X10 compiler do not yet employ load balancing in distributed-memory settings. Thus, this paper investigates the following questions regarding work-dealing in the context of the X10 programming system.

1. Is there a more flexible and a cheaper mechanism to detect load imbalances in the system without collecting data about system load?
2. Which tasks can successfully offset the underlying costs of migration across nodes?
3. Can work dealing complement X10's work stealing to alleviate load imbalances and improve performance?

We address each of these questions in detail below.

First, the primary requirement for load balancing is that all processors remain busy as long as there is an unexecuted task in the application. It is not necessary to determine the exact length of task queues in each node to initiate load balancing. Improved load balance can be achieved based on an approximation of which nodes have surplus work and which ones are starved and can process arriving task. Thus, the approach presented in this paper relies on a simple heuristic to detect load imbalance in the system and to initiate the work-dealing load balancer. Such an heuristic does not require any exchange of state information among the nodes to decide whether to transfer a task. The heuristic states that if a worker in a node fails to steal successfully in n attempts, then it is starved for work and is searching for tasks to execute. In the prototype implementation n is the number of workers per node. Thus, any newly spawned task in a loaded node can be launched at that node. A node that has not yet experienced n failed steals is considered loaded.

Second, excessive task migration can incur excessive overhead on an application's performance through loss of locality, and associated costs of migration. Thus, it is important to employ work-dealing only on a subset of tasks, hereafter referred to as *place-flexible* tasks, that can successfully offset the underlying costs of task migration, remote data access, and data movement. For example, dealing out short-running tasks that perform little computation, such as a simple counter increment or decrement, only incurs the cost of migrating the task and returning the result of the computation to its originally destined node, but fails to keep the target node sufficiently busy to manifest any substantial improvement in resource utilization. Such tasks will be referred to as *place-sensitive* tasks in the rest of the paper. A new source-code annotation, @AnyPlaceTask, enables programmers to statically identify the *place-flexible* tasks. The runtime scheduler restricts migration to only the tasks identified by the programmer.

Third, X10's scheduler maps tasks to nodes based on the node-affinity information available for each task. Any imbalance within a node is managed by X10's work-stealing scheduler that operates only within a node. The work-dealing scheduler presented in this paper complements the work-stealing scheduler to improve the work-load distribution across the nodes globally.

The key result of this paper is that using a simple heuristic to identify load imbalances in a system and then combining work-dealing and work-stealing approaches for load balancing improves the application performance relative to a system that uses no work dealing. An experimental evaluation on a cluster of 128 processors using applications from the Lonestar [12] and Cowichan [16] suites results in a speedup in the range of 2% to 16% over the X10's existing scheduler. Although the new approach shows only marginal speedup, it does not degrade performance on any of the applications indicating its applicability to a wide range of applications.

3. Background

X10's language specification [18], and the X10 programming guide [6] provide a detailed description of the X10 programming system and its language semantics. This section describes only the key concepts necessary to understand the rest of the paper.

Activities and *places* are two unique concepts in X10. Every computation in X10 is an asynchronous activity, akin to a lightweight task. An X10 place is a repository for related data and activities, corresponding loosely to a processor. Every activity runs in a place, and X10 provides the statement `async (p) S` to create a new activity at place `p` to execute `S`.

Places induce the notion of locality. The activities running in a place may access data located at that place with the efficiency of on-chip access. 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 in `S` to `p`, and, at the end, returns the control of execution to the original place. The necessary data copying is done through the runtime system calls inserted by the compiler.

Atomicity in X10 is obtained through the `atomic` S statement.

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 incur high costs when migrating them to a different processor and hence, qualify only tasks that involve smaller communication and remote data-access costs as locality-flexible.

A place can have multiple workers, each maintaining its private deque. To address runtime load imbalances between different workers, X10 employs a help-first approach to work stealing that operates only within a place. Available X10 releases do not yet support stealing across places.

4. Motivation

In X10, all code runs as part of an activity. When an X10 program starts, the main method is invoked in an activity called the root activity. Currently, the statement that creates an asynchronous activity either specifies one place where the activity will execute, or the current place is chosen by default if no place is specified. Once assigned to a place, the activity is not allowed to change its site at any point in its execution lifetime. In many cases, this constraint on place-specification over-specifies the program execution behaviour because it may be desirable to specify that an activity may be executed in one of several available sites.

This section provides an example to motivate the idea of many-to-one mapping between asynchronous events and places. The example illustrates that there are multiple places where an `async` activity may be launched.

Consider the quick-sort problem, a classic example of recursive parallel divide-and-conquer algorithm, which is a useful micro-benchmark for work-stealing schedulers. Fig. 1 shows its X10 implementation, where elements on either side of the pivot are sorted in parallel by recursive calls to the `async` statement in line 26 [1]. The asynchronous activity that executes the recursive `qsort` method is enqueued into the deque of the current worker because the new activity is created at the current place by default. Alternately, the newly created activity can also be assigned to a worker in any other place. For instance, if the current place has many tasks to execute, while places mapped to other nodes are idle, it is desirable to assign the new activity to one of those idle places. In such a situation, the scheduler must be able to assign newly arriving asynchronous activities to such places.

Line 27 in Figure 1 shows an alternate place where the activity can be launched: `here` refers to the current place, and `here.next()` refers to the next place in the set of available places. The new annotation @AnyPlaceTask allows programmers to identify tasks that can be launched in one of multiple available places. The runtime mechanism then performs work dealing only for these identified tasks.

Note that the costs of executing `async`, `async(here)`, and `async(here.next())` are different. The `async` construct works directly on the place-resident data and does not need to create their copies. However, the execution of `async(here)` creates copies of all the values used in S to the local place, i.e., `here`, even though there is no actual change of place, and the original values already

[1] The quicksort example sorts copies of the array because of the copying semantics of the `at` statement. For the sake of brevity, the example code does not account for bringing back the sorted data to the root place.

```
public class QSort {
    private static def partition(data:Array[int](1),
                                  left:int,
                                  right:int) {
        var i:int = left;
        var j:int = right;
        var tmp:int;
        var pivot:int = data((left + right) / 2);
        while (i <= j) {
            while (data(i) < pivot) i++;
            while (data(j) > pivot) j--;
            if (i <= j) {
                tmp = data(i);
                data(i) = data(j);
                data(j) = tmp;
                i++; j--;
            }
        }
        return i;
    }
    static def qsort(data:Array[int](1),
                     left:int, right:int) {
        index:int = partition(data, left, right);
        finish {
            if (left < index - 1)
                async (here)
                    // async (here.next())
                    // @AnyPlaceTask async (here)
                    qsort(data,left, index - 1);
            if (index < right)
                qsort(data, index, right);
        }
    }
    public static def main(Array[String]) {
        val N = 1000;
        val r = new x10.util.Random();
        val data = new Array[int](N,
                                   (int)=>
                                   r.nextInt(9999));
        qsort(data, 0, N-1);
        prettyPrint(data);
    }
}
```

Figure 1: Quicksort program with multiple valid destination for async activities.

exist in the place. Thus, the cost of running `async(here)` is higher than that of running `async`.

Further, the cost of running `async(here.next())` is higher than that of running `async(here)` because it involves inter-node communication to create the copies at the remote node `here.next()`. The place-flexible approach applies to `async(here)` not `async`, *i.e,* the work dealing scheduler is meant to migrate tasks that work on copies of data rather than place-resident data. Such tasks are precisely the ones that have poor locality, and hence, are annotated as *place-flexible*.

5. Selection of *place-flexible* Tasks

The basis of this work is the observation that existing X10 multi-place computations over-constrain the assignment of tasks to places. In particular, currently all tasks specify exactly the place at which they must run. This paper introduces *place-flexible* tasks that are not so constrained and are thus available for a scheduler to use in correcting load imbalances across places. Note that all tasks are not equally favourable for migration across places. Rather, some tasks are more suited to migration than others. The following task model identifies tasks that reduce the cost of migration.

A task is *place-flexible* and qualifies for migration to a target processor if any of the following conditions is met.

(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 target processor (i.e. because it is a subtask of some task originally migrated to that processor). Thus, no extra cost needs to be paid.

(c) The task's granularity is large enough to overcome the cost of migration by keeping the target node busy for sufficient time to prevent repeated and frequent migrations.

(d) The task encapsulates the data necessary for its computation.

The proposed strategy is agnostic about the source of the *place-flexible* and *place-sensitive* labels associated to the tasks. For example they could be generated by an advanced optimizing compiler, or they could be generated by the runtime based on previous executions of the same program or the same task. The labels may also be specified by a programmer when the locality preferences of tasks are not available through static or dynamic program analyses, but are readily available to the programmer from the algorithmic details and the semantic knowledge of the application.

In Asynchronous Partitioned Global Address Space (APGAS) programming languages, such as X10, the programmer already reasons about the affinity between data and tasks to express locality preferences using *places*. 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 place-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.

5.1 An Example

This section describes the Turing ring application used in our experimental evaluation and discusses how to identify which `async` to annotate as place-flexible indicating its suitability for migration to a remote place.

```
/* wl is a distributed ring of cells */
Worklist wl ← DistArray<Cell>
wl.initialize();
finish {
   for each Cell c in wl async (c.place) {
        finish {
            c.updatePredatorPop();
            async (here) c.updatePreyPop();
        }
        BodiesToMigrate mBodies = c.updateCellIDs();
        wl.update(mBodies);
   }
}
```

Figure 2: Pseudo code for Turing Ring.

The Turing ring problem simulates the interaction between predator and prey populations in a ring of cells. The pseudo code (shown in Figure 2) initializes each `cell` with a number of predators and preys and their `cellIDs`, 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. Note that different cells in the worklist and also the predator and prey populations within each cell can be processed in parallel. Thus, there are two types of tasks that can potentially be dealt-out to a remote idle node to adjust the load imbalance.

First, if the task that updates predator or prey populations (lines 5 or 6) is dealt out to a remote idle node instead of launching the task in its designated node[2], information about all predators/preys in the cell including their birth, death and migration rates must be copied to the target node to compute the new population. The new population must then be copied back to the designated node 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 dealt out then the entire cell needs to be copied to the target node. However, once the cell is copied, there is no need to copy the results back to the designated node after every update of the predator and prey populations because all other operations (lines 5 – 11) on the cell are now local to the target node, and can continue execution using the local copies of data [3]. Further, the dealt-out task makes work available for other co-located workers in the target node. Thus, the outer async that processes an entire cell is a locality-flexible task that is suitable for migration. Such an async can be annotated as place-flexible.

6. Design and Implementation

The new scheduling algorithm presented in this paper has two main goals: (1) to employ work dealing only on programmer-specified *place-flexible* tasks; and (2) to employ locality-guided scheduling for all other tasks and to rely on the work stealer to adjust load imbalances within a node. Such an approach eliminates the need for observing the load status of the computing cluster before each task is mapped to a node. Moreover, this approach prevents migration of tasks that may not offset the underlying costs.

To combine work dealing with the existing work-stealing scheduler, X10's existing task mapping approach needs to be modified. The algorithm maintains two major data structures for this purpose:

(a) `initiateWD`: a globally accessible variable that is used to indicate to the scheduler when to initiate the work-dealing policy to task mapping. `initiateWD` is initially set to `false` to indicate that work dealing is not necessary during the initial stages of an application's execution. All *place-sensitive* tasks are precluded from migration, and are directly mapped to their designated place. Thus, the scheduler needs to examine the status of `initiateWD` to decide if task migration is necessary only upon encountering a *place-flexible* task.

(b) `targetNode`: an array of booleans that is used to keep track of places that are underloaded and have workers that are continuously searching for surplus work. The `targetNode` is a a distributed array with one cell in each place, such that the array indices correspond to the `id` of each place. Initially, this array is initialized to `false`.

In the X10 runtime, a worker thread continuously loops polling for work from the bottom of its local deque of pending activities through calls to the `poll()` method enclosed within the `loop()` method. If a worker's deque is empty, the `poll()` method returns a null activity, which prompts the worker to scan other workers and the network for pending activities. If there are no incoming activities from the network, the scheduler attacks a random victim thread for work[4]. When the steal attempt through a call to the `steal()` method succeeds, the worker runs the activity and follows a similar principle for additional work until the network sends a message of termination. If the thief fails to successfully steal from this victim, it continues to randomly choose other victims to retrieve work.

The new approach presented in this paper tracks X10's stealing operations and initiates the work-dealing algorithm if any worker in a place fails to successfully steal work after n consecutive attempts. In the prototype implementation n is the number of workers per node. The idea is to exploit the principle that if a thief fails to retrieve work from its co-located workers in n attempts, then it is most likely that most of the workers in the node are either idle, and thus also searching for surplus work, or do not have any extra work to be stolen. Thus, it would be beneficial, in terms of load distribution, to map a dynamically spawned *place-flexible* task to this node.

6.1 Algorithm

The proposed algorithm uses a two-level strategy for task placement. It uses the X10's existing work-stealing scheduler to map all *place-sensitive* tasks in a locality-guided manner at each place, and complements that with a work-dealing scheduler to migrate *place-flexible* tasks between places for load balancing. The control logic guiding these strategies is shown in algorithm 1.

Migrating a *place-flexible* task `async(p)` to a remote place can be counter-productive when the task's designated place, *i.e.*, place `p`, is underloaded for lack for work for one or more of its workers. Therefore, the scheduler maps the *place-flexible* task `async(p)` to its designated place `p` if its underloaded. Using this strategy, the work-dealing algorithm prioritizes utilization of cores over utilization of nodes.

If the place is not underloaded, indicated by a "false" in `targetNode(p.id)`, then the scheduler must determine if there is any load imbalance in the system and also select the target place for migrating the *place-flexible* task.

A common approach to make these decisions would be to compare the number of tasks assigned, the number of tasks pending or the expected time of completion of the tasks in different places. However, such an approach requires several runtime explorations and computations across all places. Our algorithm uniquely learns from the stealing operations to decide when to initiate work dealing. Thus, preventing the need for expensive book-keeping to identify underloaded nodes in a cluster.

The work-dealing scheduler checks the variable `initiateWD` to determine any load imbalance in the system. An idle worker in a place sets `initiateWD` to true if it fails to steal successfully even after n consecutive attempts. Multiple workers in multiple nodes may be attempting to steal in their home places. The `targetNode` array keeps track of all such places that may be idle.

The scheduler scans the `targetNode` array to find the first underloaded place, say `q`, in the cluster, and deals out work to that place. In preparation for sending a task to the chosen underloaded place `q`, the scheduler first sets `targetNode(q.id)` to false to indicate that the place will no longer be underloaded because it is about to receive a task for execution. To decide whether work dealing needs to be employed on subsequently arriving $\texttt{async}_i\texttt{(p)}$ tasks, the `targetNode` array is scanned to see if there are other places searching for surplus work and the `initiateWD` variable is set accordingly. Then, using the $\texttt{async}_i\texttt{(p)}$'s body, a closure is created. The closure is marked for remote execution and finally an async activity is created at the chosen place `q` to run the closure.

[2] We use the term "designated node" to refer to the node (or, more accurately place) that is specified in an `async`'s definition.

[3] Note that statement 11 performs update on the worklist `wl`. However, after a task is dealt out, all the computations occur using the local copies of data. To reflect updates to the worklist, the copy of the result needs to be copied back to the dealt-out task's designated place.

[4] Only threads that are in the same address partition can be targeted for work stealing.

Algorithm 1: Control logic for work dealing.

```
Input: A sequence of place-sensitive and place-flexible tasks
       ⟨async_1(p), async_2(p), ..., async_n(p), ⟩.
Output: Locality-aware mapping of tasks async_i(p), and work
        dealing.

// Update data structures to indicate underloaded
   nodes to the scheduler and trigger work dealing
foreach place $p_i$ in the Program do
    if local steal fails $n$ times consecutively then
        atomic initiateWD ← true
        atomic targetNode(i) ← true
    end
end

// Initiate work dealing iff the async (p) is
   place-flexible, the place p is not underloaded,
   and there is an underloaded node as indicated by
   initiateWD
foreach async_i (p) in the Program do
    if (async_i (p) ==place-flexible and !targetNode(p.id) and
    initiateWD) then
        for id in targetNode \ targetNode(p.id) do
            if (atomic targetNode(id)) then
                atomic targetNode(id) ← false
                initiateWD ← targetNode.reduce(Or)
                create closure using body of async_i(p) activity
                prepare closure for remote execution
                create an async to run the closure at place(id)
                break;
            end
        end
    end
    // In all other cases, perform the usual mapping
    else
        perform locality-guided mapping of async_i(p)
    end
end
```

The book keeping operations are performed by a worker in the idle place, thus minimizing the interruption on the operation of the busy places.

7. Experimental Setup

This section describes the hardware, the compiler and the benchmarks used in the experimental evaluations.

Platform Performance measurements use a blade server with 16 nodes, each featuring two 2 GHz Quad-Core AMD Opteron processors (model 2350), with 8 GB of RAM and 20 GB of swap space, running CentOS GNU/Linux version 6.0. No other applications compete with the benchmarks for resource usage during execution, except the system logger and the secure shell.

Compiler The `x10c++` compiler version 2.1 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 the programs compile and run successfully in the experimental platform.

Benchmarks This experimental evaluation uses the benchmarks from the Cowichan suite and the Lonestar suite that are shown in Table 1. The Lonestar applications are written using the Galois framework. They were ported to X10 language by preserving the algorithms and the parallelization strategies as much as possible. Each benchmark is then restructured to fit the language changes along the lines of the multi-terminus async construct. The changes preserve semantics such that the applications continue to produce the same results as with the original programs and the unmodified X10 programming system.

Table 1: Benchmarks, their inputs, async granularity, and sequential execution time.

Benchmark	Description	Async Size (ms)	Seq. Time (s)
Quicksort	sorts an array of 1000 elements using quick-sort	0.18	0.25
Turing Ring	solves a set of coupled differential equations modelling system dynamics using 1000 elements	1.86	123
k-Means	implements a k-means clustering algorithm resulting in four clusters and using 50 iterations	383	7.7
Agglom. clustering	clusters 2M points based on similarity by building a binary tree in a bottom-up manner	529	189
DMG	constructs Delaunay mesh from 80K points	899	195
DMR	refines a Delaunay mesh of 550K triangles such that no angle in the mesh is less than 30 degrees	623	100
n-Body problem	simulates the forces acting on a system of 220K bodies using the Barnes-Hut algorithm	732	150

Each benchmark is run 10 times to account for variances, such as work-stealing in the X10 runtime, and scheduling policies in the operating system. The 95% confidence intervals are very small, hence not shown in the performance charts.

Binding of Places 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 available (*i.e.*, 16*8 = 16*2*4).

8. Experimental Evaluation

The experimental evaluations compare the performance of three different schedulers: (i) X10WS, which is the X10's existing scheduler employing work-stealing within a single address partition (ii) WD that combines X10WS with our work-dealing scheduler (iii) Eager-WD, which is similar to WD but does not wait until a load-imbalance to occur in the system to initiate work dealing. Eager-WD proactively maps each *place-flexible* task in a load-balance-aware manner.

8.1 Speedup

Figure 3 shows the speedups obtained for the applications using the X10's scheduler (X10WS) and our work-dealing-bases scheduler (WD). The speedups are relative to the sequential execution time (shown in Table 1) obtained by running the sequential implementations of the applications. The applications exhibit a larger impact of WD at higher thread and node count. Up to 16 workers, the speedup obtained with WD over X10WS is within 2%. Beyond 16 workers, the speedup obtained is larger: 16% for DMG, 13% for DMR, 9% for n-Body, 10% for Agglom, 8% for k-Means, around 2% for Quicksort and Turing Ring.

8.2 Async Granularities

The lower speedups seen in Quicksort and Turing Ring applications can be ascribed to the lightweight and short-running nature of their

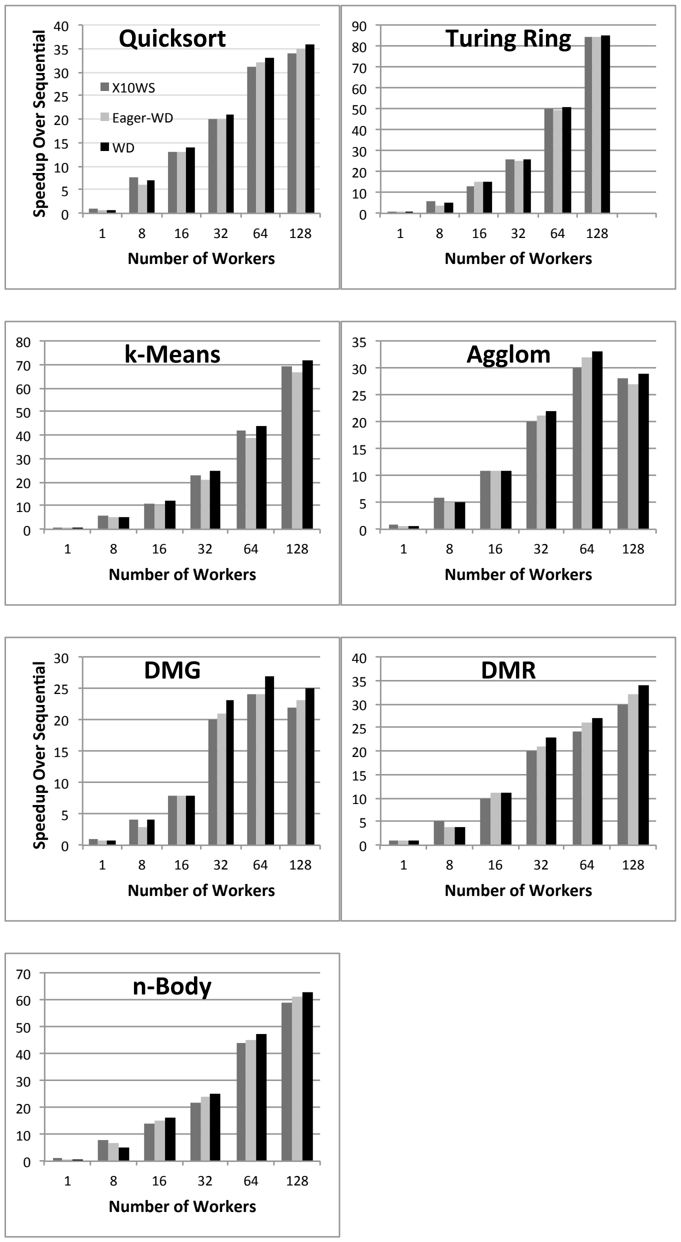

Figure 3: Speedups over Sequential Execution Time using X10WS, Eager-WD and WD. The taller bars are better.

async tasks. Their async granularities are much smaller compared to the other applications, as shown in Table 1. Such tasks suffer from the cost of migration and from the cost of returning the result of the computation to their originally destined node. However, the migrated tasks fail to keep the target node occupied for a sufficient amount of time to exhibit any substantial improvement in resource utilization.

8.3 Proactive and Reactive Work Dealing

The WD scheduler does not start its work-dealing strategy until the runtime experiences a load imbalance. The next experimental evaluation investigates whether proactively performing work-dealing-based load balancing can improve performance. To do so, it initiates WD on all *place-flexible* tasks instead of waiting for steal attempts to fail repeatedly. While WD uses the knowledge of stealing to identify a lightly loaded or an idle node, the Eager-WD algorithm needs to perform extra work in identifying such nodes. Eager-WD must first scan all available nodes to identify the idle nodes so that it can send work from *place-flexible* tasks to such nodes. If all nodes are busy, Eager-WD must compare the amount of surplus activities in each node to identify the node that is underloaded. These status checks and runtime explorations critically affect the performance of Eager-WD. Thus, it performs poorly compared to WD. Eager-WD shows similar or slightly better speedups (less than 4%) compared to X10WS.

8.4 Node Utilization

The work dealing algorithm improves workload distribution across nodes leading to an improved CPU utilization on each node. Figure 4 shows the average CPU utilization of eight cores in each node. As expected, each node exhibits increased utilization with WD in comparison to X10WS. This increase in CPU utilization can be ascribed to the extra work of task migration performed by the nodes.

WD also yields a more uniform node utilization compared to X10WS. With X10WS, the standard deviations of average node-utilization for Quicksort, Turing Ring, Agglomerative clustering, k-Means clustering, DMG, DMR and n-Body problem are 18.68, 8.23, 6.59, 8.7, 13.59, 12.45, and 8.28 respectively. With WD, the standard deviations are 5.32, 4.12, 3.09, 3.59, 5.10, 2.80, and 4.716 respectively. The lower standard deviations of average node-utilizations when using WD point to an improved load-distribution achieved with WD.

Eager-WD checks the load status of all nodes before mapping a *place-flexible* task. Thus, it performs more work compared to X10WS, resulting in an increased CPU utilization over X10WS. However, Eager-WD still exhibits a non-uniform CPU utilization because a large number of task migrations incur loss of locality-preferred mapping of tasks to nodes. As a result, Eager-WD must perform several remote data accesses corresponding to such tasks. Hence, some nodes show heavy utilization while the others show lighter utilization. All applications show similar behaviour with Eager-WD. For the sake of clarity of graphs, we show the CPU utilization resulting from Eager-WD for Quicksort application only and omit the graph for other applications.

8.5 Task Migrations

Table 2 shows the total number of tasks migrated by the work-dealing algorithm during the execution of the programs using 2 to 16 nodes[5]. The table also shows the migration ratios for the programs. Migration ratio is the ratio of the total number of tasks migrated to the total number of tasks spawned during the execution of the program. The low migration ratios (between 1e-05 and 1e-06) of the programs means a relatively small number of tasks were dealt out during distributed load-balancing. Thus, the resulting improvement in execution times are also modest.

In a separate experiment, we investigated the performance impact of making all tasks place flexible and thus, amenable to migration across the nodes. Both Eager-WD and WD strategies performed worse in terms of execution time compared to the approach where only selective tasks are identified as place flexible. Hence, we do not report the performance numbers here.

9. Limitation

In X10, final variables are globally accessible, and can be directly accessed by an `async` in any place. However, access to non-final

[5] The number of tasks dealt out in each benchmark is zero when using only one node.

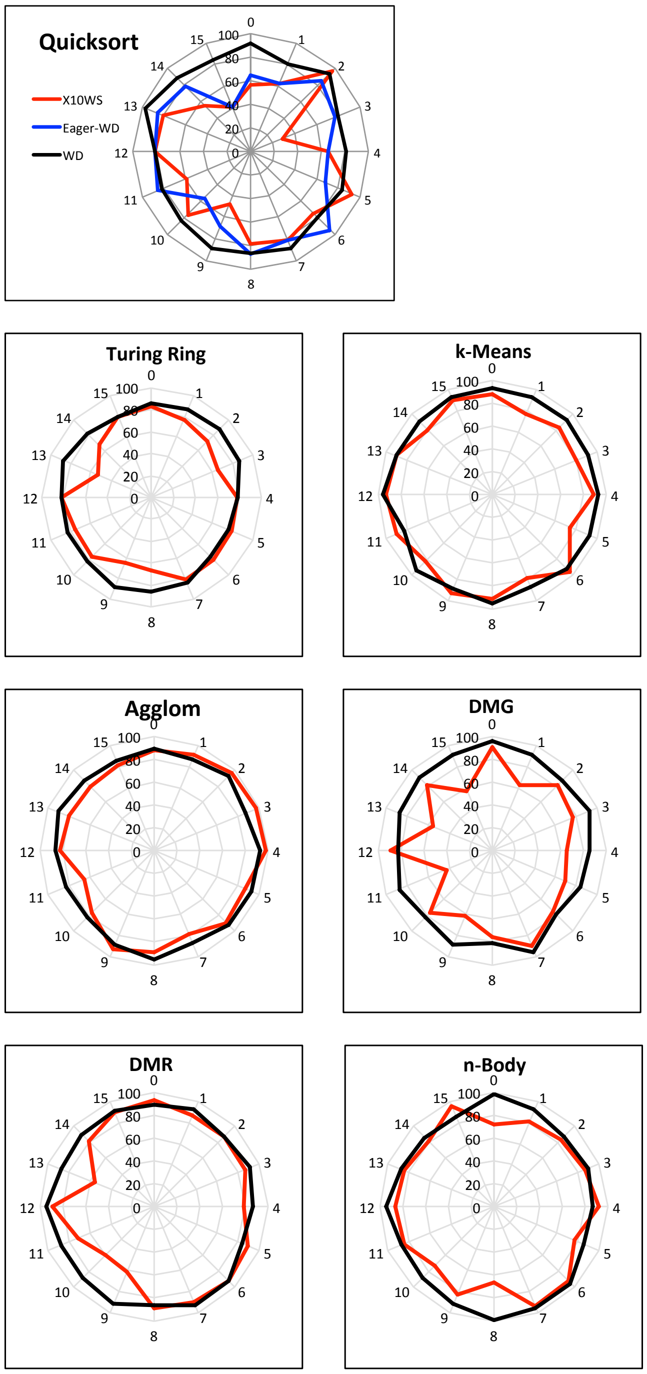

Figure 4: CPU utilization of nodes. The axis along the radius represents average CPU utilization in % and the axis along the circumference represents the nodes. Circular lines show more uniform CPU utilization.

fields is permitted only for objects residing at the same place as the asyncs. For instance, the code in Figure 5 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 5 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 it is dealt out by the work-dealing scheduler, 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.

Table 2: Number of tasks dealt out to a remote node during work dealing. Each node employs eight worker threads.

Benchmarks	**# of Task Migrations**				**Migration Ratio**
	2	4	8	16	(for 16 nodes)
Quicksort	82	191	399	683	1.12E-06
Turing Ring	97	210	427	739	2.03E-06
k-Means	106	216	479	762	1.36E-05
Agglom	99	183	429	735	1.42E-05
DMG	193	330	775	1020	1.83E-05
DMR	114	346	753	937	1.75E-05
n-Body	168	245	849	1154	1.47E-05

```
1  val arrayReg: Region = (1..100);
2  val arrayDist: Dist = Dist.makeBlock(arrayReg);
3  val distArray: DistArray[Int] = DistArray.make[Int]
4      (arrayDist, ((i):Point) => 0);
5  final val incr = incrVal;
6  finish {
7      for (p in arrayReg) {
8          val p1 = arrayDist(p); val p2 = p1.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 "
14                                 + p.toString() + " "
15                                 + distArray(p));
16             // + at(p1) distArray(p)); }}}
```

Figure 5: An example to illustrate the use of explicit `at` statements to make data-access local after a task is stolen by a remote node.

10. Related Work

An efficient work-dealing-based load balancer must strive to minimize the costs associated with its migration policy. A migration policy must perform expensive operations to decide when to perform migration, which tasks to migrate and which target nodes to choose for launching the tasks. Prior works use different kinds of information to make these decisions. Eager *et al.* collect system state information to determine the destination node at which a task selected for dealing must be launched [8]. Lau *et al.* use the processing speeds of the sender and receiver nodes, as well as their relative workloads to determine the appropriate number of tasks that can be transferred during each session of work dealing [13]. Bertozzi *et al.* uses a standard migration initiation mechanism based on the number of processes allocated to nodes [3].

These prior implementations of migration policies detect load imbalances in the system using information such as the number of tasks allocated and the average utilization of each node. Collecting such information may require system-wide exploration, communication among several nodes, and expensive synchronization operations. To the best of our knowledge, our idea of relying on work-stealing events to trigger task migration is original.

Work dealing (WD) relies on the heuristic that a large number of failed steal attempts within a node most likely indicates that the node is starved for work and can execute more tasks. Thus, WD reduces the impact of successive and failed steal attempts in a node by initiating work dealing that targets such nodes for task migration.

Saraswat *et al.* propose lifeline-graph-based load balancing to reduce the cost of failed steals [20]. A lifeline graph provides a meta-topology used for work stealing: 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 steal attempts. By remaining in quiescent state after a failed steal attempt and also by informing other nodes in its outgoing edges, lifeline graphs help reduce the impact of failed steals on performance. They initiate lifeline-graph-based load balancing using the knowledge of failed steals, while WD employs work-dealing-based load balancing using similar knowledge. Both approaches take advantage of the fact that the node that performed failed steals is idle and assign it most of the book-keeping operations needed to initiate the lifeline-graph or work-dealing algorithm.

11. Concluding Remarks

The key to load balancing in distributed memory machines is efficient utilization of all cores in a node and also all nodes in the computing cluster. While X10 provides a work-stealing-based load balancer to adjust load imbalances within a node, it does not support load balancing across the nodes. This work complements the existing work-stealing scheduler with a work-dealing scheduler.

Relying on system-wide load exploration or communication between nodes to identify any load imbalance in the system is known to be expensive. Similarly, frequent migration of tasks that incur excessive communication and remote data access costs are known to significantly penalize application performance. Therefore, this work used a simple heuristic based on work-stealing operations in a node to cheaply detect load imbalances. Further, to minimize the cost of task migration, this new work-dealing scheduler presented a statically-identified set of tasks to the runtime as suitable for migration while restricting the movement of other tasks. Using these approaches, the experimental evaluations on a cluster of 128 processors and using applications from the Lonestar and Cowichan suites result in speedups in the range of 2% to 16% over the X10's existing scheduler.

Although the overall speedup achieved with the proposed approach is marginal, none of the applications experience performance degradation. Such a behaviour indicates the applicability of our approach to a wide range of applications.

Acknowledgments

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

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The Data Locality of Work Stealing. In *Proceedings of the twelfth annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 1–12, Bar Harbor, Maine, United States, 2000.

[2] U. A. Acar, A. Chargueraud, and M. Rainey. 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, pages 219–228, Shenzhen, China, 2013. ACM.

[3] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 15–20, 3001 Leuven, Belgium, Belgium, 2006. ISBN 3-9810801-0-6.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling in Multithreaded Computations by Work Stealing. In *Symposium on Foundations of Computer Science*, pages 356–368, Santa Fe, New Mexico, 1994.

[5] R. D. Blumofe and P. A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. In *USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 1997.

[6] J. Brezin, S. Fink, B. Bloom, and C. Swart. An Introduction to Programming with X10. http://dist.codehaus.org/x10/documentation/guide/pguide.pdf, Last access: 10 March, 2013.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '05, pages 519–538, San Diego, CA, USA, 2005. ACM.

[8] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *Software Engineering, IEEE Transactions on*, SE-12(5):662–675, May 1986.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of The Cilk-5 Multithreaded Language. In *Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, 1998.

[10] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. 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, pages 341–342, Bangalore, India, 2010. ACM.

[11] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine. PFunc: Modern Task Parallelism for Modern High Performance Computing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 43:1–43:11, Portland, Oregon, 2009. ACM.

[12] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali. Lonestar: A Suite of Parallel Irregular Programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, Boston, MA, USA, 2009.

[13] S.-M. Lau, Q. Lu, and K.-S. Leung. Adaptive load distribution algorithms for heterogeneous distributed systems with multiple task classes. *Journal of Parallel and Distributed Computing*, 66(2):163 – 180, 2006.

[14] D. Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, San Francisco, California, United States, 2000. ACM.

[15] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 227–242, Orlando, Florida, USA, 2009. ACM.

[16] J. Paudel and J. N. Amaral. Using Cowichan Problems to Investigate the Programmability of X10 Programming System. In *ACM SIGPLAN X10 Workshop*, San Jose, CA, USA, 2011.

[17] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[18] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification. http://x10.codehaus.org/x10/documentation, Last access: 22 March, 2013.

[19] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The Asynchronous Partitioned Global Address Space Model. In *Proceedings of the First Workshop on Advances in Message Passing*, AMP'10, New York, NY, USA, 2010. ACM.

[20] V. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based Global Load Balancing. In *Symposium on Principles and Practice of Parallel Programming*, pages 201–212, San Antonio, TX, USA, 2011.