Load Balancing Prioritized Tasks via Work-Stealing

Shams Imam and Vivek Sarkar

Department of Computer Science, Rice University
{shams,vsarkar}@rice.edu

Abstract. Work-stealing schedulers focus on minimizing overhead in task scheduling. Consequently, they avoid features, such as task priorities, which can add overhead to the implementation. Thus in such schedulers, low priority tasks may be scheduled earlier, delaying the execution of higher priority tasks and possibly increasing overall execution time. In this paper, we develop a decentralized work-stealing scheduler that dynamically schedules fixed-priority tasks in a non-preemptive manner. We adhere, as closely as possible, to the priority order while scheduling tasks by accepting some overhead to preserve order. Our approach uses non-blocking operations, is workload independent, and we achieve performance even in the presence of fine-grained tasks. Experimental results show that the Java implementation of our scheduler performs favorably compared to other schedulers (priority and non-priority) available in the Java standard library.

Keywords: Work-Stealing, Multi-level Queue, Priority Levels, Priority Scheduling, Load Balancing, Task-Parallel Programming

1 Introduction

Load balancing is an important component in improving the performance of parallel applications as it distributes the workload over all processors. Work-stealing algorithms [1] have been gaining popularity as the technology of choice for load-balancing of parallel tasks in multicores, especially for irregular and dynamic computations. Applications such as tree or graph search problems can benefit from attempting to execute tasks in a specific order. Assigning priorities to tasks can be a method to influence the execution ordering in the scheduling of tasks [18]. In fact, benchmarks such as branch-and-bound and single-source shortest path show that prioritization of tasks can reduce the total amount of work required compared to standard work-stealing execution order [17]. Soft real-time applications with time constraints can also use priorities to promote the execution of tasks, violations of these can allow the application to continue to operate, but with a degraded quality of results. Mainstream work-stealing schedulers do not support user-defined priorities in tasks and may schedule less important tasks earlier. This scheduling leads to increased execution time or degraded quality.

The goal of priority scheduling is to assign tasks to processors in a way that optimizes overall performance metrics such as the total execution time [11]. Sequential implementations of priority scheduling are simple as they can use priority queue data structures, however, scalable and efficient parallel implementations can be comparatively more complex. Parallel implementations of priority scheduling tend to use shared concurrent priority queues. However, synchronization overheads cause such queues to not necessarily make efficient schedulers [11]. Other work-stealing priority schedulers guarantee priorities only in local scheduling via the use of priority queues per worker thread [17] and perform steals from victims once the worker becomes idle. Thus, these approaches do not adhere closely to global priorities while scheduling tasks trading off accuracy for reduced overhead. However, priority scheduling is primarily used to reduce the total amount of work done by an application. Using the priority order can curtail computation time by avoiding further exploration of a solution space or by causing the successful termination of the entire computation. Deviations from priority order may cause the application to end up doing more work.

In this paper, we develop a decentralized work-stealing scheduler that dynamically schedules fixed-priority tasks in a non-preemptive manner. We adapt a multi-level queue scheduling algorithm [13] where the tasks can be classified into priority classes and assign a separate container for each priority class. Our algorithm uses non-blocking operations and minimizes the number of compare-and-swap operations that each local worker thread performs. Furthermore, our workload independent approach extracts performance even in the presence of fine-grained tasks. Our approach relies on the unusual approach of performing steals even if the worker thread is not idle to adhere close to the priority order while scheduling. This strategy ensures that worker threads, in our scheduler, are executing tasks from the highest priority class. Thus, we minimize instances of priority inversion where low priority tasks are scheduled for execution even if higher priority tasks are available in the distributed work queue.

In summary, the contributions of this paper are as follows:

- We introduce our decentralized non-blocking algorithm for a work-stealing scheduler that respects global priorities.
- We present a lock-free implementation of our scheduler written using the standard Java library (JDK) and three work-stealing pool implementations.
- An empirical evaluation that shows our scheduler variants perform competitive to existing priority-based and non-priority-based schedulers available in the JDK.

2 Background

In the task parallel model, an application is usually decomposed into several independent and/or interdependent sets of cooperating parallel tasks. The tasks are stored in task pools, and worker threads are employed by the task scheduler to process the tasks. Scheduling deals with the problem of deciding which of the tasks from the work pool are allocated worker threads for execution. Efficient

task scheduling improves resource utilization by automatically load-balancing tasks across worker threads, thereby enhancing the overall performance of the computation.

Load balancing is based on the idea of migration of excess load from heavily loaded workers to lightly loaded ones. Given perfect information, a static scheduling algorithm attempts to produce an optimal assignment of tasks to workers that ideally balances their loads. Such information, however, may be unavailable in irregular computations that generate non-uniform tasks. Such applications cannot rely on static load balancing and have to defer to dynamic schemes that redistribute the workload at runtime.

While dynamic load balancing is complex, its benefits outweigh its complexity. The main challenge of the scheduler is to deal with the dynamic load imbalance with minimal overhead while executing tasks on multiple workers. Conventional scheduling policies, such as work-sharing, are normally centralized and global in scope. The overhead of global synchronization that must be performed to maintain a consistent state limits the scalability of such schedulers. One of the simplest, yet best-performing, dynamic load balancing algorithms for shared-memory architectures is work-stealing.

2.1 Work-Stealing Schedulers

The work-stealing algorithm is an effective decentralized technique for scheduling parallel computations. The key observation is that there is no need to migrate tasks between the workers for load balancing if all threads have enough work. What makes work-stealing successful is that it employs a reactive asynchronous strategy [16]. When a worker runs out of local work, it (randomly) chooses a victim thread and asynchronously tries to steal some work from it. The attempt to load balance is receiver-initiated as the thief actively attempts to obtain available work. The asynchronous nature allows a thief to get some work without any involvement from the victim thread that may be busy processing user tasks. Thus, the idle workers eventually unburden the busy workers and load balance is achieved.

A key component of work-stealing is the use of double-ended queues by each worker thread [2,10]. Workers treat their own deques as a stack, pushing and popping tasks from the bottom, but treat the deque of another busy worker as a queue, stealing tasks only from the top, whenever they have no local tasks to execute. Worker threads process their own deques in a LIFO order processing local tasks, as long as they are available. Consequently, it may be the case that most tasks are consumed locally, and relatively few steals are required to address a load imbalance [9].

When a worker becomes idle, it transitions into a thief searching for available work from active workers. The thief attempts to steal tasks from its victim using FIFO order, i.e. from the opposite end from which the victim is working on its deque. Since the victim and thief operate on opposite ends of the deque, efficient algorithms can be implemented for the deque that minimize the need for synchronization [6]. On a successful steal, a thief pushes the stolen task onto its

local deque, returns to the worker state, and continues to process its local tasks. If unsuccessful, the thief randomly searches for another victim and continues steal attempts until successful. This procedure repeats until all workers have exhausted their tasks and termination is detected.

Work-stealing schedulers focus on minimizing overhead in task creation and scheduling [1]. As a result, they avoid features, such as task priorities, which can add overhead to the implementation.

3 Priority Work-Stealing Algorithm

In this section, we provide the details of our work-stealing algorithm for scheduling tasks under global priority. We briefly describe the pool data structure and our overall technique before describing the algorithm.

3.1 Operations on Pool data structure

We generalize the data structure used to store tasks in a work-stealing algorithm as a *pool* – a container of ready tasks. Like the concurrent deque described by Arora et al. [1] in their work-stealing algorithm, the pool data structure is owned by a worker thread and supports the following methods:

- `push` operation: this method is executed only by the owner thread and always succeeds in storing a task into the pool.
- `pop` operation: this method is executed only by the owner thread and may return a special `empty` value if the pool is empty.
- `steal` operation: this method is executed only by the thief thread and method may return a special `empty` value if the pool is empty.

The pools support concurrent method invocations and may be implemented using any concurrent data structure. Concurrency can only occur between one invocation of `push` or `pop` in the owner thread and one or more executions of `steal` from thief threads. Each `push`ed task is extracted exactly once either by a call to `pop` or by a call to `steal`.

Note that we do not restrict the owner of the pool to use it as a stack (`push`ing and `pop`ping at the bottom) and victim to `steal` only from the top of the pool. Thus, pool implementations can also internally support priorities and can choose the appropriate task to return during the `pop` or `steal` operation. Our priority work-stealing scheduler (Section 3.3) which uses priority levels can be paired with any appropriate pool implementation to use specific execution orders. Thus depending on the application, a queue, a stack, or some other container data type can be used as the pool.

3.2 Work-Stealing with Global Priorities

We assume that tasks are assigned fixed priorities and our work-stealing algorithm schedules the highest priority task available globally in a non-preemptive

manner. In non-preemptive priority scheduling, when a task starts execution, it executes to completion even if a higher priority task arrives at the ready pool. Since we expect fine-grained parallel programs; the iterations of the scheduler loop are expected to be frequent. Thus, a minor delay in the execution of a higher priority task is not a concern, and we do not expect major priority inversion issues despite the use of a non-preemptive scheduler.

Our algorithm is inspired from the Multi-Level Queue Scheduling (MLQS) algorithm [13] where tasks are partitioned into priority levels. Like MLQS, our scheme maintains a number of distinct pools, each assigned a different priority level. More than one task may be in a given pool and thus have the same priority. However, like work-stealing algorithms, the tasks need to be distributed across worker threads. Each worker thread maintains a local pool for each priority level; we avoid a fully shared task pool as it would require expensive synchronization for every access. Thus, the tasks of any given priority are potentially distributed across the pools in the different worker threads.

In MLQS, higher priority queues must be empty before tasks from lower priority queues are allowed to execute. Similarly, in our scheme higher priority pools from all worker threads must be empty before a worker thread executes a local task from a lower priority pool. This strategy requires the maintenance of a global data structure to track the availability of tasks in the different priority levels across all the workers. It also implies that worker threads will need to perform a steal even if they are not idle, i.e. their pools may not be empty.

The assumption with standard work-stealing algorithms is that stealing accesses are rare as they only occur when the local pool is empty. One strategy that works well under this assumption is the choice of a random victim. Since steals are concurrent operations, they are accompanied by synchronization overheads. To maintain scalability, a work-stealing algorithm needs to be careful with victim selection to minimize the number of failed steal attempts where the thief discovers that the victim has no work available. When performing a steal operation, a thief must also determine how much work to steal.

Our goal is to adhere, as closely as possible, to the priority order while scheduling tasks tolerating added overheads to preserve accuracy. We can neither predict the dynamic priorities of newly spawned tasks nor expect the highest priority tasks to be evenly balanced across workers. Hence, we expect steal attempts to be relatively more frequent than traditional work-stealing approaches and steal only one task from a carefully chosen victim. This avoids *ping-pong* effects where a task, being one of the highest priority ready tasks, moves back and forth between worker threads. This also minimizes failed steal attempts required by thieves to locate and steal work when highest priority tasks are available on a single worker thread's pool.

Starvation: Enforcing priorities means that high-priority processes will always be favored over low-priority ones, causing starvation for low priority tasks. One common method of ameliorating this situation is aging in which we gradually increment the priority of waiting tasks, ensuring that they will all eventually

execute. As our benchmarks did not exhibit starvation, we do not address this issue further in this paper.

3.3 Priority Work-Stealing Algorithm

Our priority work stealing algorithm does not depend on synchrony for correctness, and it involves lock-free operations[1]. Each worker thread is guaranteed to make a locally optimal decision while making a best-effort at a globally optimal decision in scheduling a task. The local guarantee is achieved as a worker will not schedule a lower-priority task if there exists a higher-priority task in its pool(s). The lack of guarantee in global optimality is because the global data structure maintains a weakly consistent state of the availability of tasks in the different priority levels.

```
1 class WorkerThread {
2   def pushTask(priority, task) {
3     val level = priorityLevel(priority) // compute the level
4     myPools[level].push(task) // owner thread pushes local task
5     globalState.set(level, true) // update global state after populating pool
6   }
7   def run() {
8     while !stopped()
9       val task = findTask()
10      if task != EMPTY then task.run()
11        else yield()
12  }
13  private def findTask() {
14    // search local and global pools, attempting steals
15    var level = globalState.nextAvailableLevel(0)
16    // Not shown: search lower level pools locally and...
17    // return if task found updating global data structure
18    while level < priorityLevels()
19      // first search for a local task to execute
20      val localTask = myPools[level].pop()
21      if localTask != EMPTY then return localTask
22      // choose victim carefully and try to steal from there
23      for victimThread in threadClaimsTaskWithPriority(level)
24        val stolenTask = victimThread.steal(level)
25        if stolenTask != EMPTY then return stolenTask
26      // update global state, try and search again
27      globalState.set(level, false)
28      level = globalState.nextAvailableLevel(level)
29    // found no task to execute
30    return EMPTY
31 } }
```

Fig. 1. Simplified version of the non-blocking work-stealing algorithm that adheres, as closely as possible, to the global priorities of tasks. All worker threads execute the same scheduling loop. The heart of the algorithm is in `findTask()` which determines which task is scheduled next.

The scheduler operates as shown in Figure 1. The `pushTask` operation (lines 2 to 6) finds and populates the local pool for a task with a specified priority. The important operation here is that the global data structure is updated after performing local updates to the pool (line 5). Each worker thread uses a scheduling loop (lines 7 to 12) which tries to find a task to execute and executes it until

[1] The pool implementation may involve locking in the `pop`, `push`, and `steal` operations.

the scheduler is stopped. The heart of the algorithm lies in each iteration of the scheduling loop where each thread attempts to find a task to execute. The worker tries to find the highest available priority level by querying a global data structure (line 15) implemented using atomic variables to store the full/empty state for each level. This global state can be out of sync with the state of the local pools, hence the thread attempts to find a local task with a higher priority (lower level index) and schedule the task if available (code not shown on line 16). If no such local task exists, we enter the loop on line 18. The worker then attempts to find a task from the local pool and returns if it finds such a task (lines 19 to 21). If no such task is found locally, the worker must now become a thief and attempt to find a similar priority task from sibling workers (lines 22 and 25). The worker queries global state to find the workers (potential victims) that claim to have a task with the specified priority level. The worker then iterates through each of these victims and attempts to asynchronously steal a task from them. Since steals are attempted from victims claiming to have tasks at the specified level, the number of failed steals should be relatively low and only fail during high contention for few highest priority tasks. If the steal attempt is successful, the worker executes the stolen task. Otherwise, the worker now realizes that the global state is out of sync as none of the victims could provide a task. The worker updates the global state to signal no tasks of the specified priority are available (line 27). The worker keeps looping until the global state returns an invalid index (line 28) signaling no global tasks are currently available to steal. The `findTask` method returns the special value `EMPTY` signaling no tasks are currently available to execute. The worker yields itself when it is unable to find a task to execute (line 11) and then resumes the scheduling loop.

4 Implementation

We briefly describe the Java-based implementation of our scheduler in this section. Our implementation has no third party dependencies; it relies on classes and data structures available in the Java standard library (JDK). Our implementation and the benchmarks are released open source online on GitHub at `https://github.com/shamsmahmood/priorityworkstealing`.

We have implemented three variants of unbounded pool data structures for use in our scheduler based on:
(a) lock-based implementation of THE protocol in Cilk's deque [6];
(b) wait-free array-based pool based on X10's concurrent deque [3]; and
(c) wait-free linked-list-based pool based on JDK's `ConcurrentLinkedQueue` [4].
Our scheduler is flexible in that it can be configured to use either of these implementations as its pool data structure in the worker threads. Table 1 summarizes the properties our priority work-stealing scheduler compared to other work-stealing schedulers.

Our scheduler's implementation is lock-free and uses the help-first policy for task scheduling as this strategy is favorable when stealing is frequent [7]. Under this policy, spawning a child task `push`es it in the task pool and allows the parent

Table 1. Comparing our work-stealing algorithm with couple other algorithms in the literature.

Scheduler	Pool Type	When to steal from chosen victim	What to steal	What to pop
Work-Stealing without priority	Deque	Local deque is empty; victim chosen randomly	Oldest item by age	Newest Item by age
Work-Stealing local priorities	Priority Queue	Local queue is empty; victim chosen randomly	Highest priority item	Highest priority item
Our Work-Stealing global priorities	Deque	Another worker (victim) has a higher priority level pool which is non-empty	Oldest item in pool by age	Newest item in pool
	Queue			Oldest item in pool

task to continue execution past the spawn operation. The scheduler maintains a fixed number of worker threads which are configured during initialization. All worker threads execute a scheduling loop similar to the one displayed in Figure 1. During steal attempts, the victims are traversed in round-robin order starting at the previous successful victim.

Work-stealing schedulers do not enforce global prioritization since this could compromise scalability of the implementation [17]. A limiting factor for scalability of concurrent data structures is the number of global operations performed concurrently by the worker threads. We use atomic variables available in the JDK to implement the global state to reduce the risk of memory consistency errors and to minimize the cost of overheads from synchronization. We reduce the number of calls made to update the global state for a given priority level by identifying instances when the owning worker realizes a pool has transitioned to empty or non-empty states.

We allow tuning parameters like the number of worker threads and the kind of pool to use. We also allow arbitrary scalars ranges to be used as priority levels, there is no limit imposed on the size of the range. A default priority can be specified for tasks created without an explicit priority property. Tasks with invalid priorities are sanitized to meet the constraints of the allowable range for priorities.

5 Experimental Results

Our benchmarks were run on four eight-core IBM POWER7 processors running at 3.8GHz each. Each node contains 256GB of RAM and the software stack includes IBM Java SDK Version 1.7.0. Each benchmark ran using the same kernel where the user specifies priorities during task creation; only the task scheduler was changed to report the execution times. Each benchmark was configured to run using 32 worker threads; the arithmetic mean of the best fifty execution times (from the hundred and eighty iterations) are reported and error bars represent one standard deviation. Using the best execution time allows us to minimize the effects of JVM warm up, just-in-time compilation, and garbage collection.

We evaluated several different priority schedulers in the Java platform. We present empirical evaluation of our implementation using Cilk-like deques (PWSTCD), X10-like deques (PWSTXD), and JDK's (PWSTJQ) priority queues compared against: (a) JDK's work-stealing `ForkJoinPool` scheduler (FRKJPL) [10]

that does not support priorities; (b) custom implementation of a work-stealing scheduler using local priority queues that steals only when local queues are empty (WSTLPQ). (c) JDK's work-sharing `ThreadPool` scheduler that does not support priorities (THRDPL); (d) JDK's `ThreadPool` scheduler with a thread-safe priority queue using `synchronized` statements (SYNCPQ); and (e) JDK's `ThreadPool` scheduler using a concurrent queue (`PriorityBlockingQueue`) from the JDK (PBLKQ). All the priority schedulers were configured to run with ten levels of priorities unless otherwise specified.

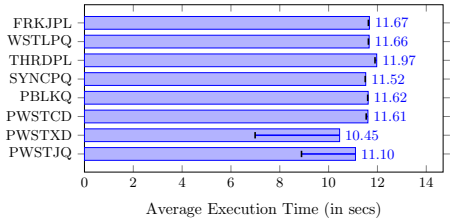

Fig. 2. Trapezoid - 800 thousand tasks to compute an area approximation.

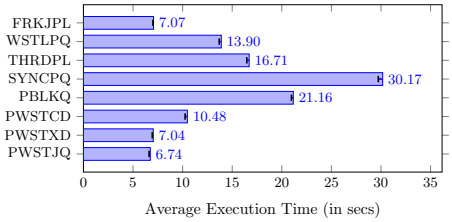

Fig. 3. Fibonacci - computing the 34th term using recursive formula.

Micro-benchmarks The first two micro-benchmarks measure overheads in the scheduler implementation, the rate at which tasks are processed. Almost all variants perform similarly in the `Trapezoid` benchmark (Figure 2), showing that the scheduler implementations for all the variants are of equivalent quality. In the `Fibonacci` benchmark, each parent task spawns two additional tasks with random priorities. Thus, this benchmark measures the overheads from supporting priorities. As expected the SYNCPQ and PBLKQ variants perform the worst due to overheads from a centralized queue implementation over the equivalent non-priority version (THRDPL) as seen in Figure 3. The WSTLPQ, despite having a decentralized pool, also shows noticeable overheads compared to a non-priority work-stealing scheduler using a similar victim selection strategy (FRKJPL). Our scheduler implementation with queues and deques (PWSTJQ, PWSTCD, and PWSTXD) performs close to FRKJPL.

Quality of Priority Scheduler Benchmarks Next benchmark used is a variant of the `JGK ForkJoin` benchmark (Figure 4) where equal numbers of tasks with random priorities are created upfront on each worker. This synthetic benchmark mimics applications that use priorities to ensure quality of results. The FRKJPL and THRDPL do not support priorities and scheduled tasks with an average priority of 4.00 as expected. WSTLPQ, with a local priority queue, schedules its local task to completion ignoring global priorities. Our decentralized implementations (PWSTJQ, PWSTCD, and PWSTXD) perform as well as a centralized global priority queue variants (SYNCPQ and PBLKQ) which schedule the tasks in perfect priority order. Next is the Unbalanced Tree Search (UTS) benchmark designed to help evaluate systems that require dynamic load

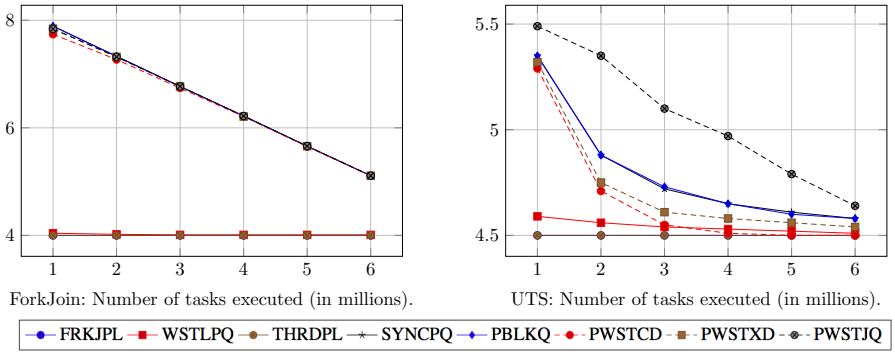

Fig. 4. Y-axis represents average priority of tasks executed by scheduler, higher is better.

balancing. In UTS, the nodes are assigned random priorities, tasks are spawned to process each node as it is discovered. SYNCPQ and PBLKQ with global priority queues and PWSTCD and PWSTXD with deque-based implementations report similar numbers. WSTLPQ, with local priority guarantees, reports priorities close to FRKJPL and THRDPL. PWSTJQ performs best, even outperforming the global priority queues due to a different traversal order. Along with results from Figure 3, this shows that our scheduler performs scheduling close to global priorities but at a low overhead.

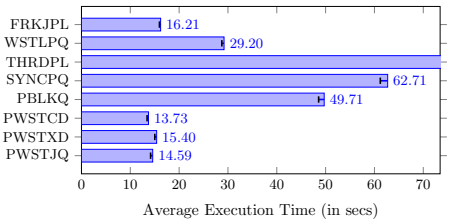

Fig. 5. NQueens: board size of 13, and cut-off after finding first 30K solutions.

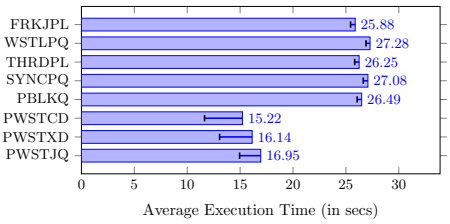

Fig. 6. Knapsack benchmark with 40 items and a sequential cutoff of 34.

Priority to Curtail Computation Benchmarks Priority scheduling can be used to reduce the total amount of work done by an algorithm. In the `NQueens` benchmark (Figure 5), priorities are used based on the number of queens placed on the board populated so far. Our scheduler using all three pools comfortably outperforms all the other variants except FRKJPL. The `Knapsack` benchmark (Figure 6) has been ported from a Cilk implementation [5]. The computation can be reduced by pruning sub-trees that cannot produce a better solution than the best one found so far, this leads to an irregular computation. We can see the benefits of using priorities to guide computations in a depth-first manner as our scheduler variants perform favorably. Note that benefits of using the priority scheduler depends on the *priority-sensitivity* of the benchmark – the amount of additional computation that can be curtailed by using a good schedule.

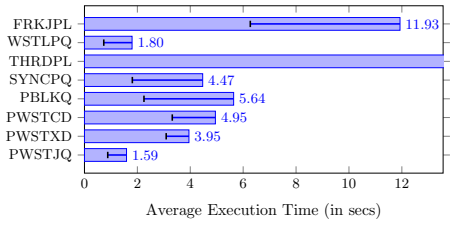

Fig. 7. A-Star Search on a grid of $350 \times 350 \times 350$.

As noted by Lenharth et al. [11], whether to use a queue or a stack depends on the particular algorithm. An efficient solution for A-star benchmark (Figure 7) requires support for priorities, it follows a path of the lowest expected total cost or distance, keeping a sorted priority queue of alternate path segments along the way. It benefits from the use of breadth-first traversal order via queues rather than deques. Hence the decentralized queued variants (WSTLPQ, PWSTJQ) perform better than dequed versions (PWSTCD and PWSTXD). One of the concerns with our approach is poor performance due to overheads from increased frequency of steals compared to standard work-stealing approaches. However, our experimental results in our benchmarks show that this is not the case.

6 Related Work

Lenharth et al. presented a chunk-based priority scheduler for unordered algorithms [11] that provide weak guarantees on the priority order of scheduled tasks. By their own admission, there are many cases where worker threads do not retrieve the highest priority task despite being aware of its existence. Mattheis et al. used a work-stealing scheduler that respects priorities in a soft real-time system [12]. Their approach uses a global queue in addition to the local queues, and a global-first stealing policy.

Wimmer et al. proposed a basic extension to work-stealing that provides good scalability, but can only provide guarantees for local task ordering in worker threads [18,17]. One of the strategies only enforces local prioritization of tasks and a worker only attempts a steal from a random victim when it becomes idle (i.e. only when the local work queue becomes empty). They also propose a ρ-relaxed priority data structure which guarantees that at most the latest k items added by each worker will be ignored, which implies that $W \times k$ items might be globally ignored during scheduling, W being the number of worker threads.

In Intel Thread Building Blocks (TBB), priority levels can be assigned to individual tasks or task groups [14]. In TBB, worker threads always attempt to execute tasks with highest priorities, while master threads execute any task they have started even if higher priority ones become available. Like TBB, we use non-preemptive scheduling; unlike TBB, our priority scheduler is decentralized and relies on work-stealing strategies for load balancing.

7 Summary

We have developed a priority-based lock-free work-stealing algorithm to work with multiple pool implementations to support priority scheduling of tasks. Our

scheduler performs competitively with existing priority and non-priority schedulers in the JDK. We are exploring the idea of using a model similar to Tchiboukdjian et al. [15] to provide a theoretical analysis of our approach. Another area of future work is to integrate priorities for eureka-style computations [8].

Acknowledgments: We are very grateful to the anonymous reviewers, Suguman Bansal, Prasanth Chatarasi, Vivek Kumar, Sri Raj Paul, and Hamim Zafar for their suggestions to clarify the contents of the paper.

References

1. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread Scheduling for Multiprogrammed Multiprocessors. In: SPAA '98. pp. 119–129 (1998)
2. Blumofe, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. J. ACM 46(5), 720–748 (Sep 1999)
3. Charles, P., Grothoff, C., et al.: X10: An Object-Oriented Approach to Non-uniform Cluster Computing 40, 519–538 (Oct 2005)
4. ConcurrentLinkedQueue (Java Platform SE 8) (May 2014), http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html
5. Frigo, M.: knapsack.cilk. http://courses.cs.tau.ac.il/368-4064/cilk-5.3.1/examples/knapsack.cilk, http://courses.cs.tau.ac.il/368-4064/cilk-5.3.1/examples/knapsack.cilk
6. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. SIGPLAN Not. 33(5), 212–223 (May 1998)
7. Guo, Y., Barik, R., Raman, R., Sarkar, V.: Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In: IPDPS '09. pp. 1–12. IEEE Computer Society (2009)
8. Imam, S., Sarkar, V.: The Eureka Programming Model for Speculative Task Parallelism. In: ECOOP '15 (2015)
9. Kumar, V., Frampton, D., et al.: Work-stealing Without the Baggage. SIGPLAN Not. 47(10), 297–314 (Oct 2012)
10. Lea, D.: A Java Fork/Join Framework. In: Java Grande. pp. 36–43 (2000)
11. Lenharth, A., Nguyen, D., Pingali, K.: Priority Queues Are Not Good Concurrent Priority Schedulers. UT Austin, Department of CS, Tech. Rep. TR-11-39 (2011)
12. Mattheis, S., Schuele, T., Raabe, A., Henties, T., Gleim, U.: Work Stealing Strategies for Parallel Stream Processing in Soft Real-Time Systems. In: ARCS 2012, Lecture Notes in Computer Science, vol. 7179, pp. 172–183. Springer Berlin Heidelberg (2012)
13. Silberschatz, A., et al.: Operating System Concepts. Wiley Publishing, 8th edn. (2008)
14. Task and task group priorities in TBB. https://software.intel.com/en-us/blogs/2011/04/01/task-and-task-group-priorities-in-tbb (Apr 2011)
15. Tchiboukdjian, M., Gast, N., Trystram, D.: Decentralized list scheduling. Annals of Operations Research 207(1), 237–259 (2013)
16. Vyukov, D.: Task Scheduling Strategies. http://www.1024cores.net/home/scalable-architecture/task-scheduling-strategies (Dec 2010)
17. Wimmer, M., Cederman, D., Träff, J.L., Tsigas, P.: Work-stealing with Configurable Scheduling Strategies. In: PPoPP '13. pp. 315–316 (2013)
18. Wimmer, M., Versaci, F., Träff, J.L., Cederman, D., Tsigas, P.: Data Structures for Task-based Priority Scheduling. In: PPoPP '14. pp. 379–380 (2014)