Work Stealing Threadpool

Abhijit Tripathy
Virginia Tech

ABSTRACT

Work stealing [6] is a popular method for scheduling parallel workloads that expresses sub-tasks and their dependent tasks, as a directed acyclic graph. Work stealing threadpools provide an efficient framework for running and scheduling parallel workloads in a decentralized manner. We implement a Work-Stealing based threadpool, called WSPool, along with other classes like FJTask (Fork-Join tasks) and its derivatives, the can run fork-join tasks while using work stealing and work helping to boost CPU utilization. We evaluate our implementation against Java's work-stealing threadpool implementation [5], and observe that it can provide comparable performance for some parallel workloads.

1 INTRODUCTION

Parallel workloads can be expressed as a *directed-acyclic-graph* (DAG), which allows sub-tasks with no dependencies to be executed in parallel. Fork-join parallelism can break a large task into smaller sub-tasks that can `fork()` and run in parallel, whereas `join()` blocks the control until the sub-tasks are finished. Java's ForkJoinPool provides a fork-join threadpool implementation that can be used to implement many parallel applications. A naive example of computing the N^{th} Fibonacci number is given in Figure 1. Although this is a much less efficient algorithm to finding Fibonacci numbers, it demonstrates how a larger task can be broken down into sub-tasks, which can ideally be computed in parallel. It also shows the dependency between the tasks, e.g., $F(N)$ depends on $F(N-1)$ and $F(N-2)$, which is a recursive relation, as $F(N-1)$ would depend on $F(N-2)$ and $F(N-3)$.

One way to approach the implementation of threadpools is to create a centralized controller that will assign sub-tasks to threads. However, in such a case, the controller might become the bottleneck, as the number of tasks scale. Therefore, a decentralized approach towards task scheduling should be preferred, as threads would be independently responsible for which tasks to run. Work stealing [6] is a popular approach towards scheduling parallel tasks, which tries to maximize CPU utilization, by ensuring threads are frequently scanning for work in their own local queue as well as the local queue of other worker's. Several approaches have been proposed for optimizing work-stealing [1–4]. We have implemented a simplified Fork-Join based threadpool based on Java's `ForkJoinPool` implementation, using mostly wait-free concurrent objects, except for a `wait4work` condition variable that uses mutex.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Our implementation utilizes work stealing as well work-helping to help resolve larger tasks into smaller sub-tasks and then join them. The key idea behind this project is to show how CPU utilization can be improved using a combination of work stealing and work helping. Blocked threads during execution of a threadpool leads to poor CPU utilization. Therefore, instead of blocking a thread, when the task it is trying to `join` has been stolen, we let the thread poll the resulting sub-tasks from the end of its local queue and execute it. This allows the thread to help its task's thief by sharing its execution, instead of waiting.

2 DESIGN

We describe the design of our threadpool implementation by first defining the behavior of each worker thread. A worker thread on creation, is assigned an index in the `queues` array, that stores its work queue. It then continues to dequeue tasks from the front of its own work queue, until its empty. If the worker's local work queue becomes empty, it scans through other workers' local queues and tries to steal tasks. If the thread is unable to steal anymore work, it will wait on the `wait4work` condition. Work queues consist of `FJTask` or Fork-Join tasks, which perform some computation, and may return a result. Each task can have one of three statuses, `NOT_YET_STARTED`, `RUNNING`, and `COMPLETE`.

2.1 Queue Management

We use double-ended queues or `deque` to store tasks. Java uses separate shared submission queues available at even indices of its `queues` array. We only use one local work queue for each worker, (`queues.length == threads`), to simplify the submission. The purpose of keeping separate submission queues, is to push tasks from non-Worker threads onto them. In our implementation, if a non-worker thread calls `submit`, `invoke`, or `join`, we randomly select a worker thread and push the task to its worker queue. This refers to the centralized task scheduler part of our implementation, as the pool explicitly decides which thread to assign task to. However, as threads continuously poll other thread's work queues to steal task, the work eventually gets distributed across all threads. This refers to the decentralized queue control where each thread independently scans for tasks to execute. Moreover, only owner threads can call `pollFirst` on the work queues, whereas stealing threads need to call `pollLast`, to avoid contention and interference.

2.2 Work Stealing and Helping

In our implementation, when tasks are split into smaller sub-tasks, they eventually need to be joined as a way of synchronization in the application. For example, in a parallel merge-sort application, all parallel sub-tasks need to be joined before merging the two halves of the sub-array. We provide `join()` and `get()` methods the provide this feature.

```java
public class Fibonacci extends RTask<Integer> {
    private final int n;
    public Fibonacci(int n) {
        this.n = n;
    }
    protected Integer compute() {
        if(n == 0 || n == 1) {
            return n;
        }
        Fibonacci large = new Fibonacci( n: n - 1);
        Fibonacci small = new Fibonacci( n: n - 2);
        large.fork();
        return small.compute() + large.join();
    }
}
```

Figure 1: Code Listing for Fibonacci Parallel Task

If a non-worker thread calls these functions, they will be blocked until the task is computed and a result is available. If a worker-thread calls these functions, we have a few choices. If the task's status is `NOT_YET_STARTED` then we atomically change the status to `RUNNING` (using `casStatus()`), and execute it. If `casStatus()` fails, it would imply that some other thread might have won the race, and was able to dequeue or steal this task from the worker. We then check if the status if `COMPLETE`, in which case we return the result. If the status is `RUNNING`, we try to help join the sub-tasks created by the stolen tasks. We do this by scanning the end of our own queue, to look for tasks and execute them. We look at the end of the queue, because often in parallel applications, the smaller sub-tasks are enqueued after the larger sub-tasks. Therefore, we try to execute smaller sub-tasks that will help join the larger sub-tasks. Once we are out of tasks in our own local queue, we block (`park`) the current thread until the task's status becomes `COMPLETE`.

3 IMPLEMENTATION

Our implementation uses concepts provided by Java's `ForJoinPool` and `ForkJoinTask` classes. We have implemented `WSPool` (Work Stealing Pool) and `FJTask` (Fork-Join Task) classes, along with some utility classes to help users test our implementation (`RTask<T>` (RecursiveTask), `RAction` (RecursiveAction), etc.)

3.1 Work Queues

We use Java's `ConcurrentLinkedDeque` to implement the work queues. Java itself uses an array-based self-growing deque for its work queue implementation in `ForkJoinPool`. We use a Linked-list based deque, as most of our deque operations just require pop and `poll`. The only time a worker thread iterates a deque is when it is trying to remove a task during `join` and doesn't want it to be stolen by another worker.

When a worker thread is executing tasks from its own queue, it uses `pollFirst()`, whereas when it is trying to steal a task or help join a task, it uses `pollLast()`. During stealing, we use `pollLast` to avoid interfering with the queue's owner performing `pollFirst`. When trying to help join a stolen task, we use `pollLast`, as for most parallel workloads, the smallest tasks are often at the end of work queue (similar to function-call stack).

3.2 Signalling Work

We use Java's `Condition` class to block threads that are waiting for work. `Condition` is derived from `Lock`, and is similar in semantics to `pthread_cond_t` in *pthreads*. Use of `Condition` is the only source of locks or mutexes in our implementation. Therefore, when a worker thread has no task to execute or steal, it waits on the `wait4work` condition variable. Meanwhile, whenever the main thread or any worker thread executes `submit`, `invoke`, or `fork`, we signal the waiting threads using `condition.signalAll`.

3.3 FJTask.join()

When a thread calls `join` or `get` on a task, it must compute the task and return it's result. However, if the caller thread is the `main` thread, or it is a worker thread but its task is stolen and is being already run by another thread, then the caller thread will need to block until the task has been computed. We use an inner `Node` class to maintain a linked list of waiters for each `FJTask`. `FJTask.join()` and `FJTask.get()` internally call `awaitDone`, that tries to compute the task itself or helps join the sub-tasks created by this task. However, if neither is possible, the thread en-queues itself to list of waiters, and then parks itself, using `LockSupport.park()`.

3.4 Atomic Operations

We try to avoid using locks in our implementation, to reduce sequential sections in our code. We do not use mutual exclusion in any part of the code apart from the `wait4work` condition variable. The work-queues internally use `compareAndSet` operations to obtain a wait-free deque. We also use Java 9's `VarHandle` to atomically update a task's `status` field (`casStatus()`). Similar operations are used to atomically en-queue a thread to `FJTask`'s waiting list (`casHead`).

4 EVALUATION

We evaluated our implementation with four micro-benchmarks, `MergeSort`, `PrimeCounter`, `Fibonacci`, and `ArrayTransform` (scales a large array by a scalar quantity). We were unable to run our implementation on *rlogin*, as our code uses features from Java's latest version, that is not available in the *rlogin* cluster. We run our code on `openjdk 18.0.2.1`, on a machine with Apple M1 Pro CPU with 8 cores. We compare our results to Java's own work-stealing threadpool implementation. We observe the performance of our implementation gets worse as we increase the number of threads for Fibonacci and ArrayTransform, as they are diverging tasks, as compared to MergeSort and PrimeCounter, which are converging tasks. We expect that the degradation in performance is due to more threads failing to steal tasks, thus causing a skewed distribution of work.

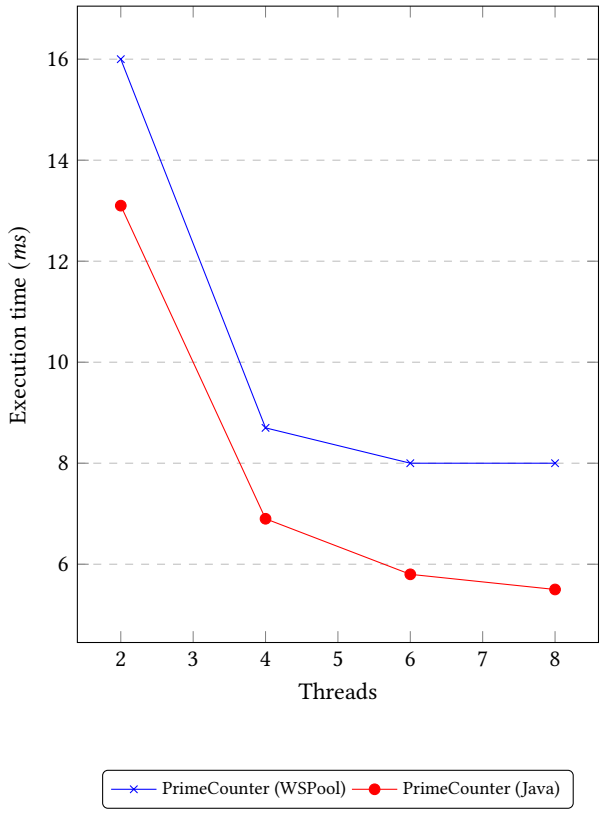

Figure 2: *Execution time* vs. thread count for PrimeCounter between 10 and 1M

5 CONCLUSION

We observe that our implementation performs poorly compared to Java's ForkJoinPool implementation. There are numerous design choices that lead to a superior performance from Java's implementation. Java's ForkJoinPool, does not keep all threads running all the time, and creates new thread only when required. Moreover, it uses a single 32-bit field to store information about threads waiting for work. It also uses a much larger number of work queues, where workers can switch queues with little overhead. This project has been an excellent opportunity to learn how work-stealing threadpools can be implemented in a language with no pointers. We plan on improving the work-stealing algorithm of our implementation by using zero locks, and providing the threads more opportunities to steal by increasing the number of the worker queues.

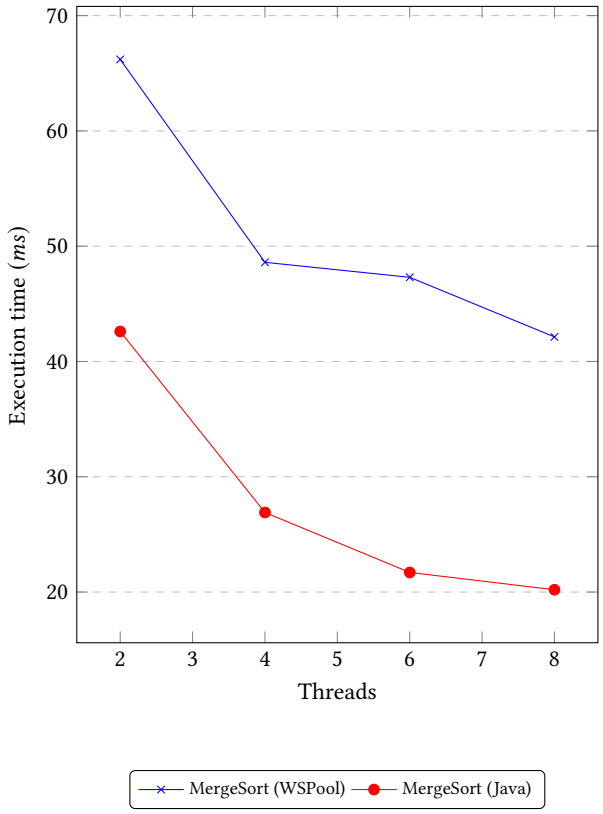

Figure 3: *Execution time* vs. thread count for MergeSort with 1M elements

REFERENCES

[1] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable Work Stealing *(SC '09)*. Association for Computing Machinery, New York, NY, USA, Article 53, 11 pages. https://doi.org/10.1145/1654059.1654113

[2] Vivek Kumar, Karthik Murthy, Vivek Sarkar, and Yili Zheng. 2016. Optimized Distributed Work-Stealing. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms (IA^3'16)*.*IEEEPress*, 74˜77.

[3] D. Brian Larkins, John Snyder, and James Dinan. 2019. Accelerated Work Stealing *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 75, 10 pages. https://doi.org/10.1145/3337821.3337878

[4] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. 2009. Idempotent Work Stealing. *SIGPLAN Not.* 44, 4 (feb 2009), 45–54. https://doi.org/10.1145/1594835.1504186

[5] Oracle. 2020. Standard Edition 8 API Specification Java™Platform. Class Executors - newWorkStealingPool. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html. (2020).

[6] Jixiang Yang and Qingbi He. 2018. Scheduling Parallel Computations by Work Stealing: A Survey. *International Journal of Parallel Programming* 46 (04 2018). https://doi.org/10.1007/s10766-016-0484-8

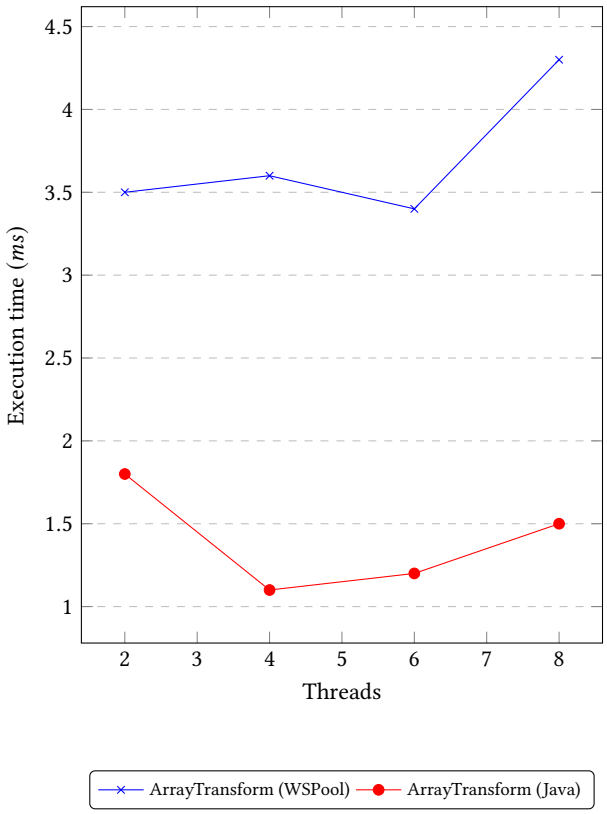

Figure 4: *Execution time* vs. thread count for ArrayTransform with $1M$ elements

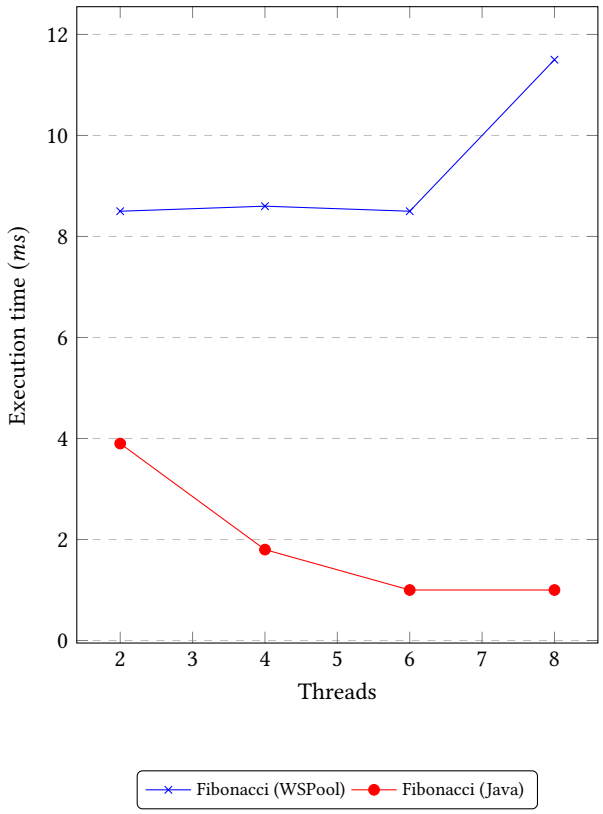

Figure 5: *Execution time* vs. thread count for 25^{th} Fibonacci number