Zorro: A Distributed Thread-Pool with Stealing
Final Report

Yukun Jiang
Carnegie Mellon University
Language Technologies Institute
yukunj@andrew.cmu.edu

Leo Guo
Carnegie Mellon University
Language Technologies Institute
jiongtig@andrew.cmu.edu

1 SUMMARY

In this project, we have incrementally designed and implemented **Zorro**, a distributed thread-pool with a work-stealing policy. We showcased that our implementation outperforms quite a few alternative implementations under various kinds of workloads and scaling factors. The experiments are all conducted on the PSC supercomputer cluster.

2 BACKGROUND

Our goal is to test different implementations of the thread-pool framework, so we did not try to focus on one algorithm, application, or system. Instead, we designed various test cases to test various aspects of different frameworks. Specifically, we have tested 5 different implementations:

- 1. Dummy Pool. This is the baseline that runs all test cases using only one thread.
- 2. Global Pool. This is the naive implementation of the thread pool framework, corresponding to the situation in which tasks are submitted to one global queue, and each worker tries to pop from that global queue. Lock on the global queue should be grabbed whenever tasks are submitted to or popped from the queue.
- 3. Local Coarse Pool. This framework provides basic improvement to the Global Pool. In this implementation, each worker has its own queue, and tasks are submitted to and popped from that own queue. The idea is to resolve the contention on the lock of the global queue in the previous implementation. Each pop/push operation on the local queue requires the lock on the local queue.
- 4. Local Fine Pool. This framework further improves on Local Coarse Pool using a fine-grained locking queue. The idea is that the pop operation depends on the head of the queue, and the push operation depends on the tail of the queue. Thus, it is possible to have two separate locks for pop and push.
- 5. Local Fine Pool with Stealing. This is the final version of our framework. Each idle worker could try to steal from another peer's local queue.

We designed 6 test cases to test different aspects of different frameworks:

- 1. Light Test: This test is used to test contention on locks for different frameworks.
- 2. Correctness Test: This test makes sure each submitted task is executed exactly once.
- 3. Computation Test: This test is designed for scenarios in which each task involves a fair amount of computation, each task has the same size, and there is no dependency between tasks. Ideally, we should have linear speed-up in this case.
- 4. Imbalanced Test: This test is used to benchmark the performance of our framework when there is a significant load imbalance across different workers.
- 5. Recursion Test (Divide): This test would demonstrate that our framework can be used for tasks with recursion. Specifically, each task would only generate more tasks, and there is no merging of results.
- 6. Recursion Test (Divide + Merge): This test would demonstrate that our framework can be used for tasks with recursion, where each task tries to merge results from spawned tasks.

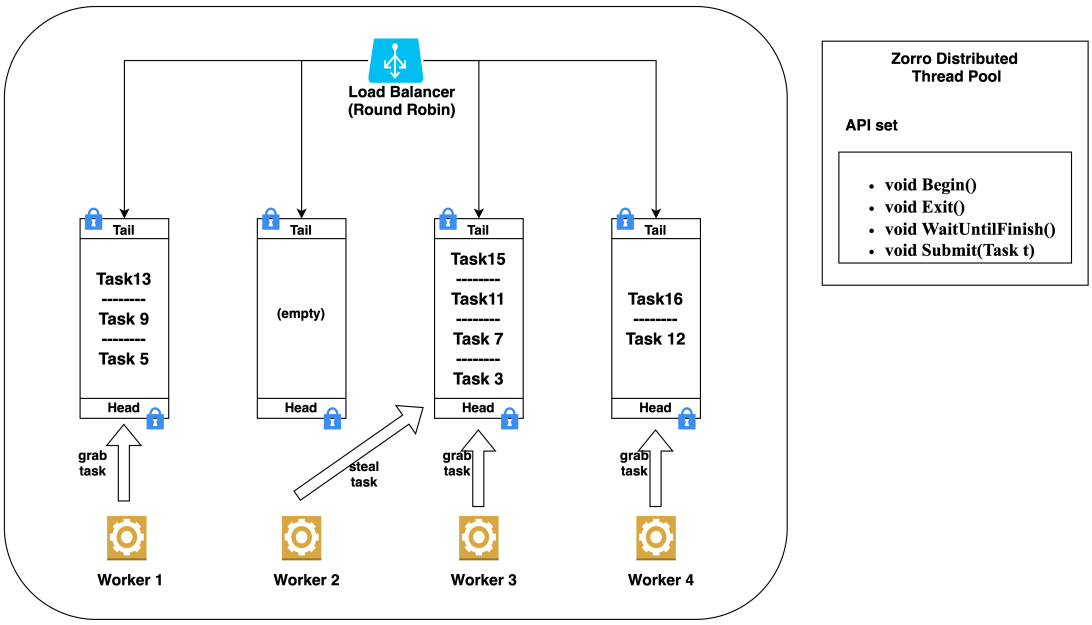

Figure 1: Architecture for Local Fine Pool with Stealing

3 APPROACH

3.1 TECHNOLOGY

Since this is a relatively low-level system project, we use C++17 on Linux mostly, though the code should also be cross-platform compilable since we only rely on the standard template library. We make heavy use of <**thread**> and <**atomic**>.

We benchmark the performance statistics on PSC supercomputer cluster, where we can scale the parallelism in {1, 2, 4, 8, 16, 32, 64, 128}.

3.2 API

We expose a very simple user interface to our **Zorro** library. There are 4 functions publicly available:

1. **void Begin**(): instruct every worker in the threadpool to start working.

2. **void Exit**(): send a signal to every worker to clean up themselves and terminate threads.

3. **void Submit(Task t)**: submit a task to be executed by one of the workers. The **Task** is of type **std::function<void(void)>**, any parameter or return value position could be bounded into such a closure using lambda expression or **std::bind**. The implication of this simplification is that to get a return value, one must either pass in a pointer or reference instead of getting a raw return value.

4. **void WaitUntilFinish**(): blocks until all the submitted tasks, including tasks spawned by the submitted tasks, have been completed.

3.3 ITERATIONS OF FRAMEWORK

3.3.1 Serial

The **DummyPool** is the serial version that we use to calculate speedup later on. It is single-threaded and directly executes the task when the user calls **SubmitTask(Task t)**.

3.3.2 Global Queue

The **GlobalPool** is the first real threadpool implementation. There is a big global work queue maintained with a mutex lock. Every task is submitted to the tail of the queue and every worker continues to poll a task from the head of the queue when they are idle.

Since this is only one single work queue in the pool, **GlobalPool** should have a good workload balancing property. However, since all the workers are polling from one single queue, the data and lock contention might rise rapidly as the number of threads increases.

3.3.3 LocalCoarse Queue

The **LocalCoarsePool** is the first distributed implementation of the threadpool. To solve the contention problem we described above, each worker is now equipped with its own work queue with a mutex lock. Upon a user submits a new Task t, we adopt a Round-Robin approach to insert the new Task into one worker's private work queue.

We speculate that this should reduce the contention problem described above, but is more sus-

ceptible to workload imbalance problems.

3.3.4 LocalFinePool

The **LocalFinePool** is one step further to solve the problem of data and lock contention. So far we have only used **std::queue** as the data structure for the work queue and mutex for synchronization. However, even with a worker's private work queue, it might still experience some contention when one side is trying to insert a new Task into the queue and the other side is trying to poll a Task out of the queue to be executed.

In this implementation, we will design our own fine-grain locking queue. During the implementation, we also consult the book by Williams (2021). There will be two locks protecting the tail and head of the queue respectively. Therefore, this should enable insertion at the tail and pop at the head concurrently. But workload imbalance might still remain a problem to be solved.

3.3.5 LocalFinePoolSteal

The **LocalFinePoolSteal** is the final implementation we plan to implement. Inheriting all the optimization details from **LocalFinePool**, this implementation also enables a work-stealing policy to solve the workload imbalance problem.

Especially, when a worker becomes idle and there is no more Task in its own private work queue to poll from, this worker will follow a uni-direction to steal batch tasks from its neighbor. We speculate that this will greatly alleviate the workload imbalance problem at a small cost of lock contention with neighbors.

If given more time, we might try to implement more complex stealing policy as described by Leiserson (1999).

3.4 IMPLEMENTATION OF TEST CASES

3.4.1 LIGHT TEST

As mentioned above in the Background section, this test is used to test contention on locks for different frameworks. The submitter would submit n tasks to the thread pool when the test starts, where each task is an empty task that does nothing. Thus, the time to execute this test case is the cost for workers to submit/grab tasks from the thread pool. Thus, the higher the contention is, the higher the execution time would be.

3.4.2 CORRECTNESS TEST

This test is the sanity check for our implementations. Its sole function is to make sure that each submitted task is executed exactly once. At the beginning of this test case, the main thread would generate a zeros array A with length n, and submit n task to the thread pool. Each task takes in a unique index i, and its job is to add one to $A[i]$. After execution, we want to make sure that each location of array A contains 1, which means that each submitted is executed exactly one time.

3.4.3 COMPUTATION TEST

This test is designed for scenarios in which the tasks are embarrassingly parallelizable: each task involves a fair amount of computation, each task has the same size, and there is no dependency between tasks. At the beginning of this test case, the main thread submits n tasks to the thread pool, where each task's job is to let the thread executing it sleep for 50 milliseconds, to emulate a fair amount of computation. Ideally, our framework should achieve linear speedup under these test cases.

3.4.4 IMBALANCED TEST

This test is explicitly designed to test the case in which there is a workload imbalance across workers. Assume there are T threads in total. At the beginning of this test case, the main thread submits n tasks to the thread pool. For the i-th submitted task, if i **mod** $T == 0$, the job would be heavy, which is to sleep the executing thread for x milliseconds, where x follows a uniform distribution with a lower bound of 90 and an upper bound of 110. Otherwise, the job would be light, which is to sleep the executing thread for y milliseconds, where y follows a uniform distribution with a lower bound of 5 and an upper bound of 15.

If there is a round-robin scheduler (which is the case for Local Coarse Pool, Local Fine Pool, and Local Fine Pool with Stealing), one of the T threads would only get heavy jobs and the remaining threads would only get light jobs. By doing so, we explicitly created a workload imbalance.

We expect that the Global Pool should perform reasonably well in this test case since each worker would poll the global queue for a task as soon as it finishes the current task. We also believe that the Local Fine Pool with Stealing framework should also perform well since it can use stealing to resolve workload imbalance. Local Coarse Pool and Local Fine Pool should suffer the most in this case.

3.4.5 RECURSION TEST (DIVIDE)

This test would demonstrate that our framework can be used for recursion **without** merging results from spawned tasks. We chose QuickSort to test the capability of our framework to handle recursion.

One thing to notice is that in order to support recursion, the original algorithm needs to be modified such that it needs to call **Submit** to spawn new tasks.

We modified an implementation of QuickSort online to test our framework, and the link to that implementation is https://www.geeksforgeeks.org/cpp-program-for-quicksort/ by GeeksforGeeks (2022).

3.4.6 RECURSION TEST (DIVIDE + MERGE)

This test would demonstrate that our framework can be used for recursion **with** merging results from spawned tasks. We chose MergeSort to test the capability of our framework to handle recursion tasks from which parent tasks would try to merge results from spawned children tasks.

Similar to QuickSort, the MergeSort algorithm needs to be modified such that it needs to call **Submit** to spawn new tasks. Our implementation is to spawn the children tasks first, and then spawn the merging process as well to make sure the parent task would not be blocked on waiting for spawned tasks.

In addition, we need to make sure the merging task is only executed after all children tasks finish. To achieve this, each child task needs to take in a flag that is set to 0 initially, and the child task would set the flag to 1 after the tasks spawned by the child task and the child task itself finish. The merging task needs to have all flags from all children. It would execute the merging process if and only if all flags are 1. Otherwise, the merging task would submit itself back to the thread pool.

We modified an implementation of MergeSort online to test our framework, and the link to that implementation is https://www.geeksforgeeks.org/in-place-merge-sort/ by GeeksforGeeks (2023).

4 RESULTS

For the experimental results, we mainly focus on execution time and relative speedup to the serial single-threaded execution. We will calculate the speedup by work size-constrained experiment with a large enough workload.

For the experiment setup, we will benchmark the five tests above with all the five types of threadpool implementations, with parallelism ranging from 2, 4, 8, $\cdots$, 128.

The experiments are all conducted on PSC supercomputer cluster regular-memory station session. The task parameters are as follows: LIGHT=100000, NORMAL=3000, IMBALANCED=1000, CORRECTNESS=100000, RECURSION=10000000, RECURSION_MERGE=200000.

One point to mention is that we don't specifically optimize for recursion tasks, neither implementation-wise nor algorithm-wise. The reason why we are providing such a benchmark here is to prove that, our threadpool is able to handle the type of tasks that are "self-spawning".

All the recursive tasks submitted to the threadpool need to be in the form of "Passing the Continuation". Because if the recursive tasks just blindly spawn new tasks and wait for their completion in the original task, it will quickly exhaust all the thread workers in the threadpool, stuck in a deadlock. To solve this, the recursive task needs to spawn its children tasks and also spawn a "checking-for-completion" task into the threadpool. When its children tasks are all finished, they will write a flag into a memory space to be checked by the checking task, and then the rest continuation tasks may be picked up and resumed. This is essentially how the system **Cilk** does but on an application level not the compiler-expansion level.

From figure[2] (an empty task body), we could infer that the execution time mostly comes from lock contention. We can see the contention intensity in order is Global Pool > Local Coarse-locking Pool > Local Fine-locking Stealing Pool > Local Fine-locking. This aligns well with our initial assumption. By switching from one big global worker queue to a per-worker private work queue, a lot of contentions could be avoided. And the work-stealing policy does not come for free. It adds a slight contention when workers are trying to steal from neighbors and grab a neighbor's lock.

From figure[3] (computationally-intensive task), unsurprisingly we get perfect linear scaling, meeting up our expectations. The computation task is in itself heavy enough to outweigh the little overhead of task submission and synchronization. Plus, they are mostly the same workload in each task, leaving little space for workload imbalance. Hence, our

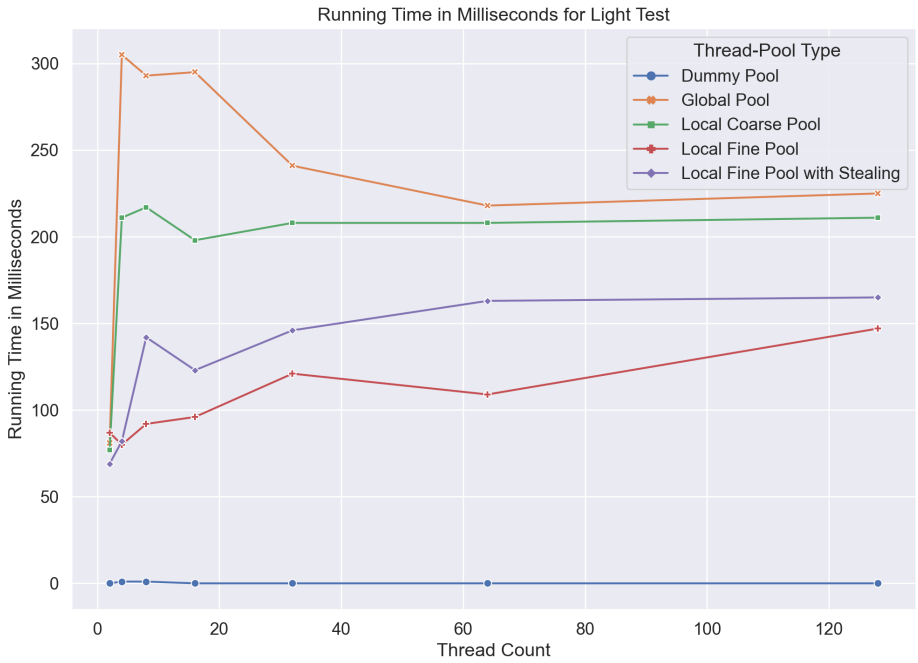

Figure 2: Running Time in Milliseconds for Light Test, Which Tests for Contention.

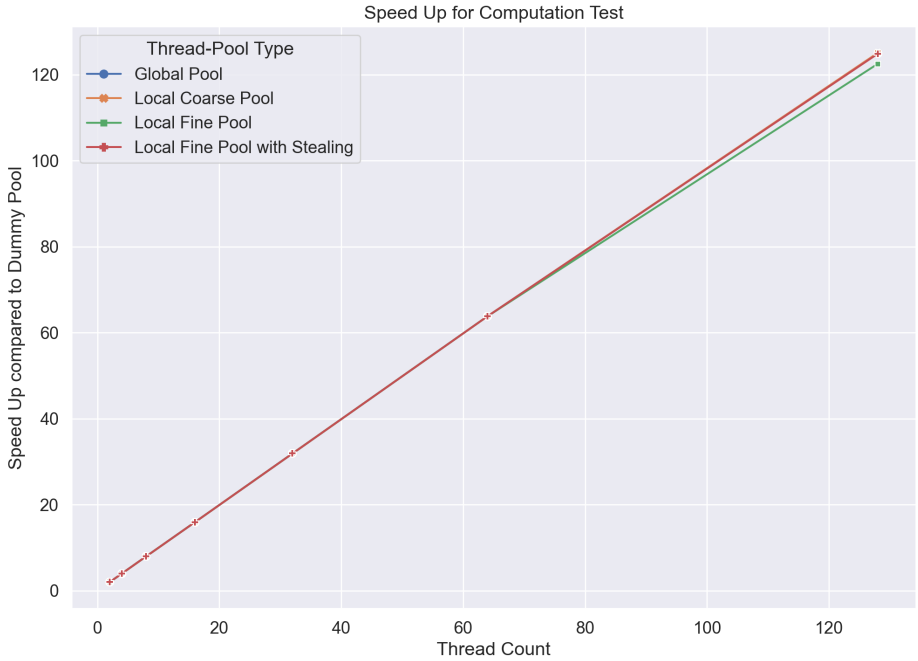

Figure 3: Speed Up of various frameworks compared to Dummy Pool for Computation Test, Which Contains Embarrassingly Parallelizable Tasks.

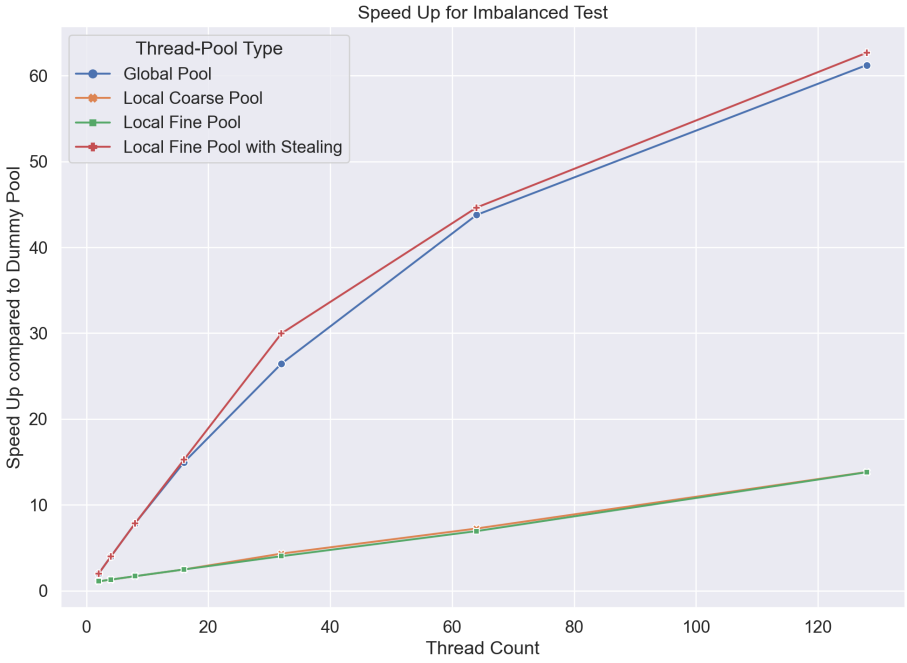

Figure 4: Speed Up of various frameworks compared to Dummy Pool for Imbalanced Test, Which Tests for How Well Different Frameworks Handle Workload Imbalance.

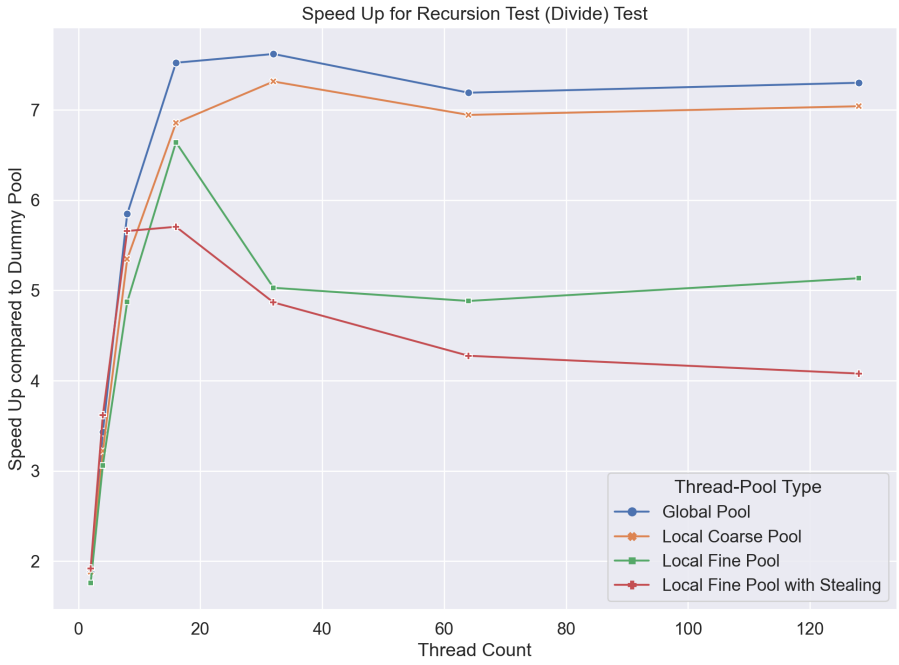

Figure 5: Speed Up of various frameworks compared to Dummy Pool for Recursion Test (Divide), Which Executes QuickSort on Workers.

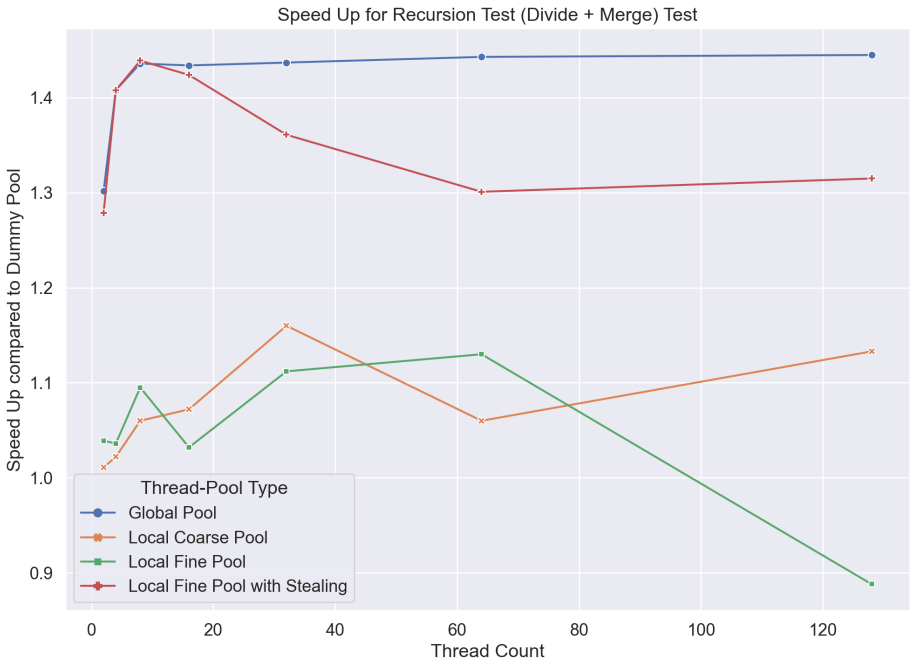

Figure 6: Speed Up of various frameworks compared to Dummy Pool for Recursion Test (Divide + Merge), Which Executes MergeSort on Workers.

implementation achieves the same speedup as the number of threads.

From figure[4] (load imbalanced task), we see a dramatic difference between various versions of implementation. This test suite is specifically designed so that one of the many worker threads will get way more tasks than its peers. Without any work balancing control, Local coarse-locking and Local fine-grained locking's speedup is severely impacted by the slowest worker. This could be inferred from Amdahl's law. And Global pool and Local work-stealing version perform well on this task. Notice that since there is only a single global work queue in the Global version, it essentially has no workload imbalance problem. From the statistics, we can see our work-stealing policy approximates close enough to the Global version and even surpasses it in speedup because of lower lock contention.

From figure[5] (Recursion Test (Divide)), we can see that the speedup is reasonable for QuickSort. The Speedup plateau after the thread counts > 16, and we believe the reason is on implementation of the QuickSort algorithm, rather than the inefficiency of our framework. This project focuses on improving the thread-pool framework instead of implementing and tuning QuickSort, so we did not spend too much time tuning the result. One thing to notice is that the performance of Local Fine Pool and Local Fine Pool with Stealing drops after the thread counts > 16. We hypothesized this occurs due to the fact that these two frameworks used our implementation of a fine-grained locking queue. Our fine-grained locking used spinning to try to pop work from the queue. On the other hand, the other two frameworks used condition variables, workers would be notified when there is a job available in the queue. Thus, our fine-grained locking queue would try to get/release the lock to pop much more frequently, which adds additional overhead. We used spinning instead of a conditional variable to implement the fine-grained locking queue since that can provide the best speedup to Light Test. Due to the time limit, we do not have the time to further our hypothesis.

From figure[6] (Recursion Test (Divide + Merge)), we can see that the speedup is not significant at all. The problem is with the nature of MergeSort. The first problem is that if we want to sort an array with length n, the heaviest task is the last task, which merges two sorted subarrays with length $n/2$. The closer we are to the end of

the execution, the fewer the tasks, and the heavier the tasks. Such a pattern is inherently bad for parallel execution. Additionally, there is a dependency between tasks: the algorithm cannot merge the two subarrays until those two subarrays are sorted. Such dependency is sequential in nature, which further constrains the speedup. Even with these two disadvantages, we can see that Global Pool and Local Fine Pool with Stealing still outperform Local Coarse Pool and Local Fine Pool, which suggests they can resolve the issue of workload imbalance better.

5 WORKLOAD DISTRIBUTION

Yukun Jiang and Leo Guo roughly evenly split the workload of this project, i.e. 50% v.s 50%. Yukun mainly takes care of the project build setup, interface design, global locking, and local coarse-locking version implementations. Leo mainly takes care of the test suite design, local fine-locking, and stealing version implementations. And they collaborate together on debugging and writing reports.

References

GeeksforGeeks. 2022. C++ program for quicksort. *online*.

GeeksforGeeks. 2023. In-place merge sort. *online*.

Robert D. Blumofe Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *MIT Lab of Computer Science*.

Anthony Williams. 2021. C++ concurrency in action. *Manning*, 248-53-7955.