Work stealing Fork Join Parallel Framework in C++

(Jiarui LI (jiaruil2), Xin JIN (xinj2))

URL

https://cmu15618-project-team-jiaruil2-xinj2.github.io/15618F21-Project/

Summary

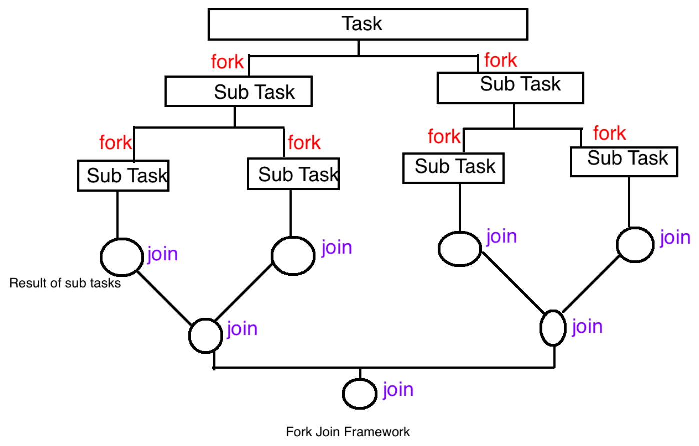
We are going to implement a fork-join parallel framework with work-stealing distributed queues in C++. Also, we would like to analyze its performance, overhead, and bottleneck under different implementation strategies, such as a child or continuation stealing, lock-free queue, etc.

Background

The fork-join model is a simple and efficient approach to executing parallel programs. This parallel pattern allocates threads and dynamically divides tasks recursively until every task reaches a certain granularity. All the threads execute their own tasks parallelly without block except to wait out subtasks. Threads will join at some subsequent points to continue sequential execution.

Problems that suffer a heavy workload of computing and can easily be divided into several subtasks can benefit from this easy-to-use parallel scheme. Similar to divide-and-conquer algorithms, divided tasks are computed on multiple cores. Without too much overhead, programs can be sped up at a significant level.

The challenge



(https://self-learning-java-tutorial.blogspot.com/2015/07/java-fork-join-framework.html)

Implementing a fork-join framework is challenging for the following reasons. For one thing, in order to get the expected speedup, the scheduling cost should be as small as possible. There are a lot of possible implementation strategies with their own tradeoffs. For instance, the number of works a thread should steal when its own work queue becomes idle is a key design problem. In addition, the implementation of the work queue also has lots of possibilities, such as a single centralized queue or per-thread distributed queues with coarse/fine-grained locks or even lock-free synchronization. For another, from the users' perspective, a simple and easy-to-use API is also important. Doing this project, we want to learn the internal mechanism of fork-join parallel paradigms, explore different implementation strategies and analyze their pros and cons.

Workload:

Using the fork-join paradigm, tasks may block to wait for the completion of their subtasks. Therefore, there are dependencies between threads executing parent tasks and sub-tasks. However, since the task abstraction is separated from the execution instances, and we tend to use a dynamic assignment along with a work-stealing strategy, it is less likely to have divergent execution.

Constraints:

Using the fork-join paradigm, because of the nature of dynamic task generation, it is hard to anticipate the workload of each task and impractical to statically map each task to each parallelization instance. Therefore, a distributed work queue with a work-stealing strategy is a better solution in our opinion.

Resources

Although we will start from scratch, we have found some good paper to begin with, which discusses how this parallel design pattern works in detail.

- R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," Proceedings 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 356-368, doi: 10.1109/SFCS.1994.365680.
- Färnstrand, Linus. “Parallelization in Rust with fork-join and friends: Creating the fork-join framework.” (2015).
- Frigo, Matteo & Leiserson, Charles & Randall, Keith. (1999). The Implementation of the Cilk-5 Multithreaded Language. ACM SIGPLAN Notices. 33. 10.1145/277650.277725.
- Lea, Doug. (2000). A Java Fork/Join Framework. ACM 2000 Java Grande Conference. 10.1145/337449.337465.

Goals and deliverables

PLAN TO ACHIEVE

We plan to implement a framework that can be used for the fork-join parallel paradigm. The framework is expected to allocate threads using a thread pool to execute tasks. Also, we want to try two implementation strategies which are child stealing and continuation stealing. On one hand, we will apply this model to a simple problem like merge sort to analyze the performance and speedup under different numbers of processors compared with a simple sequential version of the program. On the other hand, we would like to achieve the results using different strategies, break down the time spent in the execution and discuss the overall performance.

HOPE TO ACHIEVE

If time permits, we would like to build an auxiliary visualization tool to demonstrate the working process and the scheduling graph of the fork-join model.

Besides, we would like to explore the different granularity of parallelism. To be specific, we want to try using processes, threads, and user-level threads as workers.

In addition, we would like to construct a distributed fork-join framework and get rid of the limitation of one single machine.

For the demo, we decided to show the plots of speedup compared to the performance on a single processor to demonstrate the effectiveness of our fork-join model. Running the program to show the output result is also a good way to demonstrate the correctness of our framework. If we finish the visualization tool we hope to achieve, we can use it to generate a gif that can be more intuitive to see the process of the fork-join model.

Further, we hope to learn more about the advantages and disadvantages of this parallel model when dealing with different amounts of workloads and different numbers of processors.

Platform choice

We would like to implement this framework in C++ 11. In addition, we plan to test our framework on GHC machines and PSC machines. Since GHC machines are free and have eight cores, they will be a good place for us to do benchmarking and initial testing on speedup. We also hope to see the performance with a larger number of cores, because we want to know if there is a bottleneck or other factors which limit the speedup. This is one of our focuses in this project, and that's why we choose PSC to do some final testing.

Schedule

- Week 1: Literature review, API design + Thread pool
- Week 2: Dedicated Distributed Queue
- Week 3: work stealing I (intermediate checkpoint)
- Week 4: work stealing II + visualization tool/distributed extension I
- Week 5: measure the performance + visualization tool/distributed extension II
- Week 6: finalize our design, write a report and prepare for the poster session