Revised Plan

Here is our adjusted detailed schedule for the coming weeks

1. week 4:
    a. lock-free queue optimization using dequeues as work queues (Xin)
    b. Implementation of continuation stealing policy (Jiarui)
    c. visualization tool I and distributed fork-join model (Xin)
    d. analysis and reduction of tail latency (Jiarui)
2. week 5:
    a. visualization tool II and distributed fork-join model (Xin)
    b. analysis and reduction of tail latency (Jiarui)
    c. use case analysis: (measure performance of different applications using our fork-join model)
        i. Sudoku (Xin)
        ii. 8 queen puzzle (Jiarui)
3. week 6:
    a. Final report and poster (Xin, Jiarui)
    b. Preparation for the poster session (Jiarui, Xin)

Completed Work

We designed the API for our fork-join parallel framework in the first week. To be specific, from the users' perspective, they need to inherit and implement a **callable Task class**, which is what they want to execute. Within the class's function call operator, users can spawn a subtask by creating a new Task object and using **fork()** method provided by the base class. Later, users can use **join()** to wait for the completion of that task. To start the execution, users need to create a thread pool instance and a root task. Then users can use the **startProcess()** method of the thread pool to execute the root task.

Then in the next few weeks, we completed a basic framework with a naive child-stealing strategy. The framework allocates threads using a thread pool to execute tasks. The thread pool also has a set of work queues for each thread. However, the root task and those tasks that stem from it are assigned to the main thread first. So initially all other threads are busy waiting and trying to steal tasks from a random work queue. If stealing fails, it will sleep for a while and try again. If stealing succeeds, it will push the stolen task into its own work queue and start working on it. Tasks can use the fork function to create subtasks and push them on a work queue. When all the subtasks in the main threads are finished, the major task is finished at that time. Furthermore, we use fine-grained locks to guarantee that every operation on the work queue is mutually exclusive.

Must-haves and Nice-to-haves

We are halfway towards the goal we would like to achieve. However, we may not be able to implement a distributed version of our fork-join model because it is quite challenging and time-consuming, and we are not sure how to implement message passing among workers in different machines, from scratch or using RPC or using MPI. Besides, analysis on the 8 queen puzzle and analysis of tail latency are desirable options.

Below is the list of minimum goals that we would like to achieve.

1. A fork-join parallel framework with different stealing strategies, i.e. child-stealing and continuation stealing.
2. Plots of performance with different applications such as the Fibonacci sequence and sudoku.
3. Visualization tool.

For the poster session, we plan to show the plots of speedup compared to the performance on a single processor to demonstrate the efficacy of our fork-join model. We can apply this framework to different applications. For example, solving sudoku using backtracking under our fork-join model will be a good choice besides the Fibonacci sequence. Under different workload constraints, the effectiveness of our model can differ which is an interesting topic to do analysis. We would like to explore the tail latency effect in our fork-join model to help us break down the time distribution in the whole computing process. Also, running the program to show the output result is a straightforward way to show the correctness of the fork-join model.

Without further optimization, namely exponential backoff when steal fails, lock free queue, etc, our current fork join framework can achieve following performance.

Fib(n)	serialization (s)	fork join (s) - no seq	fork join (s) - seq
25	0.002	0.071	0.003
30	0.011	0.789	0.013
35	0.077	7.375	0.143
40	0.749	too long	1.788

Issues

On one hand, we had a weird bug in our current implementation. We used the Fibonacci sequence to test our performance of the fork-join model. In our original design, we called **fork()** for all subtasks, then we **join()** all the subtasks. However, when the number got large, it threw

out a bus error which might be a stack overflow in our opinion. For now, we **fork()** and **join()** subtask one by one. However, the performance was bad. We would like to know how to solve it in order to achieve high performance and correctness under a heavy workload.

On the other hand, in order to build a distributed framework for the fork-join model, we have to come up with a method that allows us to do communication across different workers in different machines. We are not sure if using RPC or MPI is practical.