AGGREGATOR LIBRARY

DESIGN DOCUMENTATION

COMS 4995: LANGUAGE LIBRARY DESIGN C++

RAKESH YARLAGADDA – ry2294
SHARAN SURYANARAYANAN – ss4951
SRIHARI SRIDHAR – ss4964

DESCRIPTION:

Our project idea is to build a Aggregator Library which takes a set of tasks with their dependencies as input, builds a dependency graph for these tasks and executes these tasks as concurrently as possible. In the world of web services, it is commonplace for a web service to communicate with downstream services to process an API request from its clients. Web service developers try to decouple these actions into as many individual tasks as possible. However, some tasks end up as dependencies for other tasks to execute. Typically, web service developers execute these tasks in a sequential order one after the other. However, in this implementation, the throughput and latency of the web service will take a hit because each task is blocked on the previous task even though there is no dependency between them.

On the other hand, another way of implementation is to execute independent tasks in parallel through creation of threads, and use their result to process the dependent tasks. In this implementation even though it improves the web service's latency and throughput, the developers will have an overhead of thread creation and thread management. A better implementation for the above use-case is to decouple the tasks from their execution and hand it over to an executor service which takes care of running tasks while honoring their dependencies on other tasks, which is our project idea. We want to build a task execution library, Aggregator Library, which allows web service developers to exploit parallelism with minimal overhead of handling thread creation and thread management at the same time honoring task dependencies.

EXAMPLE USE-CASE:

Let us consider a sample use-case where we have four tasks A, B, C and D with dependencies as shown below:

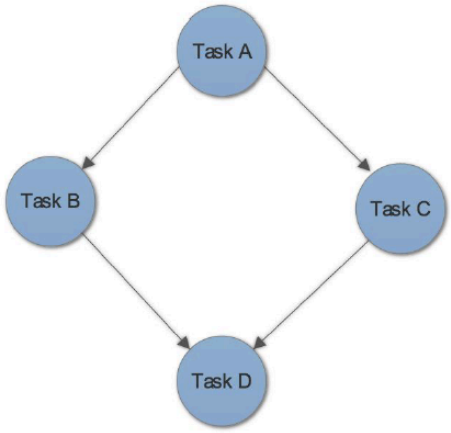

Fig 1 Tasks Dependency Graph

Task A is independent, Tasks B & C are dependent on Task A and Task D is dependent on completion of Tasks B & C.

A straight forward way solving this problem is to perform the tasks sequentially, that is perform task A first, followed by task B, later task C and finally task D as shown below:

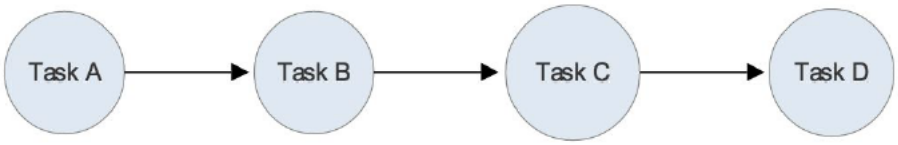

Fig 2 Tasks Execution Schedule

In such a serial execution schedule we cannot make use of the independence between task B and task C. Thus we will lose some time that could have been saved by computing tasks B and tasks C in parallel.

The ideal way of execution would be to implement Task B and C in parallel by creating two parallel threads say, Thread B and Thread C which execute Task B and Task C respectively. Finally the main thread will wait for the child threads to complete and finally execute Task D. Below is their invoking strategy for parallel execution.

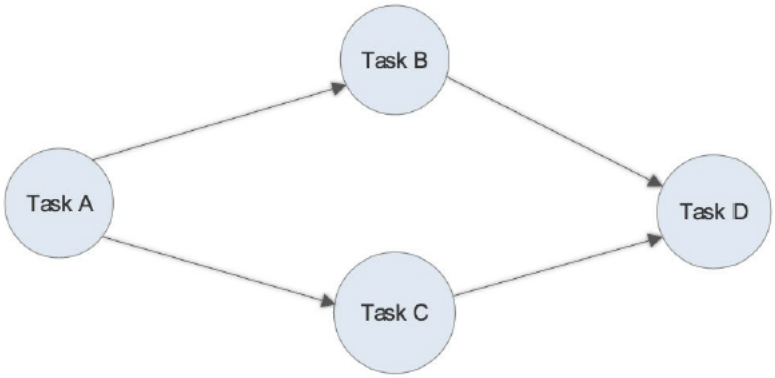

Fig 3. Parallel Execution Schedule

Example code for parallel execution of tasks:

```
void taskA(int* a){
        *a+=10;
}
void taskB(int* a,int *b){
        *b=*a+20;
}
void taskC(int *a,int *c){
        *c=*a+10;
}
void taskD(int *b, int *c){
        cout<<"The value of b+c is "<<(*b+*c)<<endl;
}

int main(){
        int a, b, c, d;
        a=0;
        thread A{taskA,&a};
        A.join();
        thread B{taskB, &a, &b};
        thread C{taskC, &a, &c};
        B.join();
        C.join();
        thread D{taskD, &b, &c};
        D.join();
        return 0;
}
```

This may look simple but when there are a lot of tasks and a lot of dependencies between tasks then it becomes complex as you will have to manually manage all the tasks which are being executed in parallel. Also an user might forget to join or wait for all dependencies to be completed before he calls the next task to execute. The developers have to deal with thread creation, management, and dependency injection of the output results to dependent tasks. This is the problem, that our aggregator library tries to address and mitigates it, by taking away the burden of thread creation, management and dependency injection from developers. Below is the rewritten code using our aggregator library.

```
int main() {

    shared_ptr<Aggregator> aggregator = AggregatorFactory::newFixedThreadPoolAggregator();
    aggregator->addNode(shared_ptr<Runnable>(new A()));
    aggregator->addNode(shared_ptr<Runnable>(new B()));
    aggregator->addNode(shared_ptr<Runnable>(new C()));
    aggregator->addNode(shared_ptr<Runnable>(new D()));
    aggregator->execute();
    return 0;
}
```

Sample Class Defenition:

```
class B : public Runnable {
private:
    string label{"B"};
    unordered_set<string> dependencies{"A"};
public:
    string getLabel() const {
        return label;
    }

    unordered_set<string> getDependencies() const {
        return dependencies;
    }

    void run(unordered_map<string, shared_ptr<Runnable>> dependencies) {
        cout << "Running node" << label << " with dependencies ";
        for(auto it = dependencies.begin(); it != dependencies.end(); it++) {
            shared_ptr<Runnable> node = it->second;
            cout << node->getLabel() << " ";
        }
        cout << endl;
    }
};
```

Output:

```
Running nodeA
Running nodeC with dependencies A
Running nodeB with dependencies A
Running nodeD with dependencies B C
Finished executing
```

AGGREGATOR LIBRARY ARCHITECTURE:

Below is the architecture diagram describing individual components of our aggregator library.

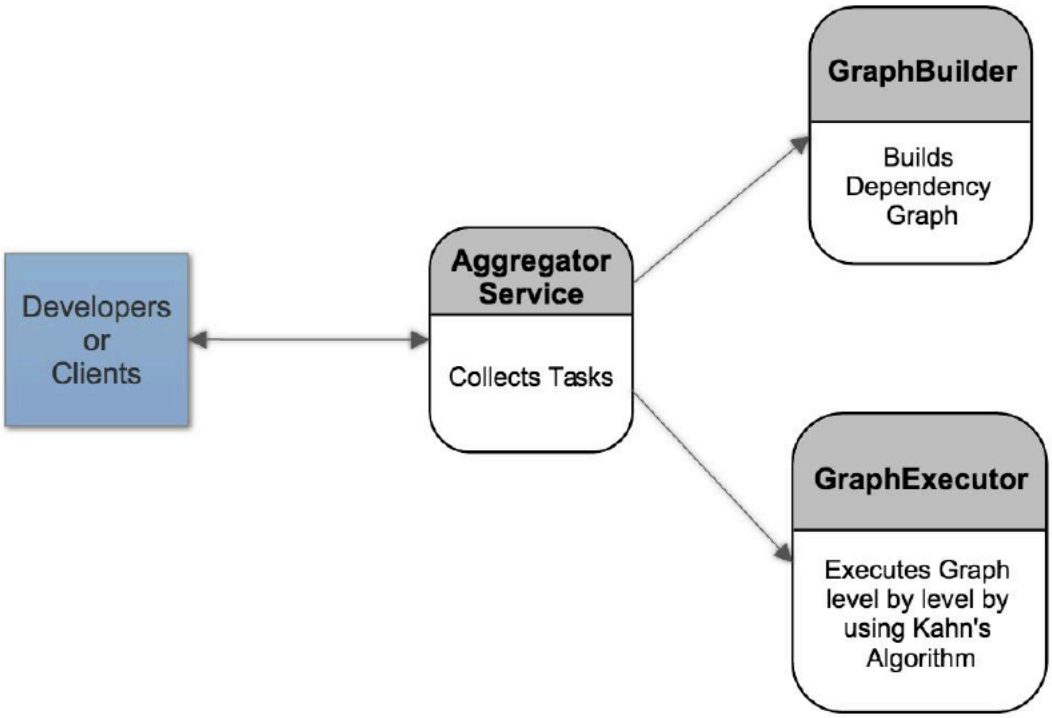

Fig 4: Architecture Diagram

From the above diagram we can see that, our library mainly has three components which termed as AggregatorService , GraphBuilder , GraphExecutor . Our first component AggregatorService interacts with clients to collect their tasks, and passes them first to GraphBuilder. The GraphBuilder's main purpose is to perform dependency injection for tasks and build a graph using the dependencies. Then AggregatorService checks if there exists a cycle in the graph and if so it throws an exception and exists. If the graph is acyclic, then it injects the first level of tasks into GraphExecutor. Now comes the role of the GraphExecutor which manages the thread pool that it has created to execute the tasks in parallel. GraphExecutor implements the Khan's Algorithm of Graph level by level traversal to execute the tasks. It starts executing the tasks as soon as their dependencies finish executing.

DESIGN:

Class Definition for Each Node/Task:

The user needs to define a class which implements Runnable for every node/task he wishes to add. This class must override three methods- setDependencies, run and getLabel.
getLabel is to set name to the node/task. The user can name the node with any string, but we would recommend using the class name. The user specifies the task to be done by the node inside the run function.

Dependency Specification:
Inside each class defined which implements Runnable the user needs to override the set dependencies function to return an unordered_set of strings which are the labels of all the nodes on which this node is dependent on. These set of dependencies are used to construct the dependency graph.

Thread Pool-
We have used a Factory design pattern for the thread pools. But we have currently implemented only one version, a singleton thread pool. We have design a singleton thread pool because we want to have only one thread pool at any given point of time. We have currently fixed number of threads in the thread pool to be equal to the hardware concurrency, but we plan to implement functions which allow the user to set the number of threads in the next version. The thread pool picks up tasks from the thread-safe task queue and runs them.

Thread Safe Queue-
We have made use of a Factory Design Pattern even for thread safe queues. We have currently implemented only one type of thread safe queue which make use of a single mutex and a queue. The thread safe queue provides push, wait_and_pop, empty, size functions. We plan to implement various other versions in the next version.

Graph Builder and Execution-
We make use of Kahn's algorithm to build the graph and execute it. The Graph Builder calculates the in-degree and out-degree for each task. In-degree is the set of nodes the current task is dependent on, and Out-degree is the set of nodes which are dependent on the current task. After calculating in-degrees and out-degrees, Graph executor will select all the Tasks which have in-degree as 0 (which are independent tasks). These tasks will be pushed to the Graph Executor which communicates with the Thread Pool to execute the tasks. The Graph Executor will return completed tasks to Graph Builder. For each completed task, Graph Builder reduces the in-degree of completed task's dependents. Finally, for each dependent it checks if the it's in-degree has been reduced to 0. If so, it will push it to the Graph Executor else skips.

Use of shared_ptr:
We are using shared_ptr's for transferring access to tasks across Graph Builder, Graph Executor and Thread Pool. Since, the task is referenced by these components we have chosen to use shared_ptr. However, since these pointers are internal to the library and do not end up being dangling pointers, we still preferred to use shared_ptr to avoid the risk.

REFERENCES:

1. Kahn's algorithm:
   https://en.wikipedia.org/wiki/Topological_sorting#Kahn.27s_algorithm
2. Java Implementation of the Aggregator Library:
   We implemented a similar aggregator library using Java for the course COMS 6156: Topics in Software Engineering.