**USE case**

**Solution - ADOPT**

**IMPACT**

[*https://msdn.microsoft.com/en-us/magazine/dd569760*](https://msdn.microsoft.com/en-us/magazine/dd569760)

Parallelizing Operations With Dependencies

Design your code to optimize performance

Parallel Async with Dependencies

A little while ago, I had the requirement to write a tool that could execute a series of Async I/O tasks; each with a different set of dependencies, which influenced the order of the operation. This can be addressed simply with sequential execution; however if you want to maximize the performance, sequential operations just wont do - you must build the tasks to run in parallel.

To optimize performance these tasks need to be scheduled based on the dependency and the algorithm must be optimized to run the dependent tasks in serial as necessary and in parallel as much as possible.

Below is a typical example of a data structure such as a *Graph, which* can be used to represent scheduling constraints. A *graph* is an extremely powerful data structure in computer science that gives rise to very powerful algorithms.

A graph data structure consists of two basic elements:

1. **Vertex** - A single node in the graph, often encapsulates some sort of information.
2. **Edge** - Connects one or two vertices. Can contain a value.

A graph is collection of vertices connected by edges.

The implication of a directed graph leads to an expressive programming model. By using directed graph it is easy to enforce the one-way restriction. The definition says that a *directed graph* is a set of vertices and a collection of directed edges. Each directed edge connects an ordered pair of vertices. Here each task is represented as a Node in a graph, and the dependency between two nodes is represented by a direct edge.

4

7

1

8

6

2

5

3

In this representation of a Directed Graph, Node 1 has dependencies on Node 4 and 5, Node 2 depends on Node 5, Node 3 has dependencies on Node 5 and 6 and so on.

See the code below for a simple implementation of a Directed Graph. The implementation is not the most elegant. It follows the imperative paradigm with mutable variable, but for demo purpose you get the idea of implementation.

<pre class="brush: fsharp;">

type DirectGraph(value:int) =  
  
    let mutable e = 0  
  
    let vertices  = Array.init value (fun \_ -> List<int>())  
  
    member this.Vertex = value  
    member this.Edge = e

member this.getVertex(v:int) =  vertices.[v]

    member this.AddEdge(v:int, w:int) =  
           vertices.[v].Add(w)  
           e <- e + 1

This code is a good starting point, but there are some problems. How can we ensure that all the edges required have been registered? Consider if Node 2 with dependencies to Node 7 and 8 is registered, but maybe Node 8 isn’t. Moreover, it could happen that some edges depend on each other, which would lead to a Directed Cycle. In the case of a Directed Cycle, it is critical to run some tasks in parallel; otherwise certain tasks could wait on another forever in a deadlock.

Another problem is registering a set of computations that have an order and precedence constraint. This means that some tasks must complete before some other task is begun. How can the system enforce and verify that all the tasks are completed while still respecting the ordered constraint?

The solution is called *Topological Sort,* which means that I can order all the vertices of the graph in such a way that all its directed edges target from a vertex earlier in the order to a vertex later in order. For example, if a Task A must be completed before Task B, and Task B must be compete before Task C which must complete before Task A; then there is a cycle reference and the system will notify of the mistake by throwing an exception. If a precedence constraint has a direct cycle, then there is not a solution. This kind of checking is called *Directed cycle detection.*

If a Directed Graph has satisfied these rules, it is considered a *Directed Acyclic Graph (DAG),* which is primed to run several tasks, which have dependencies in parallel.

The link here is a great article providing additional information about DAGs: <https://msdn.microsoft.com/en-us/magazine/dd569760>.

Back to the figure above, Task 7 and 8 run in parallel. As soon as Task 8 complete the Task 5 starts and Task 6 will run after Task 7 and 8 are both complete, and so on.

Let’s implement the DAG solution applying the strategy we have learn here to run in tasks in parallel while respecting the order of the dependencies for increasing the performance.

Now, let’s define the data type

    Type TaskInfo =  
        { Context : System.Threading.ExecutionContext    
          Edges : int array  
          Id : int  
          Task : unit -> unit  
          NumRemainingEdges : int option  
          Start : DateTimeOffset option  
          End : DateTimeOffset option }

The Type TaskInfo contains and keeps track of the details of the registered task, the id, function operation and dependency edges. The execution context is captured to be able to access information during the delayed execution such as the current user, any state associated with the logical thread of execution, code-access security information, and so forth. The start and end for the execution time will be published when the event fires.

member this.AddTask(id, task, [<ParamArrayAttribute>] edges : int array) =  
            let data =  
                { Context = ExecutionContext.Capture()  
                  Edges = edges  
                  Id = id  
                  Task = task  
                  NumRemainingEdges = None  
                  Start = None  
                  End = None }  
            dependencyManager.Post(AddTask(id, data))

The purpose of the function AddTask is to register a task including arbitrary dependency edges. This function accepts a unique id, a function task that has to be executed and a set of edges which are representing the ids of other registered tasks which all must all be completed before the current task can executed. If the array is empty, it means there are no dependencies.

[<CLIEventAttribute>]  
member this.OnTaskCompleted = onTaskCompleted.Publish

The event OnTaskCompleted triggered each time a task is completed providing details such as execution time.

member this.ExecuteTasks() = dagAgent.Post ExecuteTasks

The function ExecuteTasks starts the process executing the tasks.

The core of the solution is implemented using a MailboxProcessor (aka Agent) which provides several benefits. Because the natural thread-safety of this is primitive, I can use .NET mutable collection to simplify the implementation of DAG. Immutability is an important component for writing correct and lock-free concurrent applications. Another important component to reach a thread safe result is isolation. The MailboxProcessor provides both concepts out of the box. In this case, I am taking advantage of isolation.

The MailboxProcessor dagAgent is keeping the registered tasks in a current state “tasks” which is a map (tasks : Dictionary<int, TaskInfo>) between the id of each task and its details. Moreover, the Agent also keeps the state of the edge dependencies for each task id (edges : Dictionary<int, int list>). When the Agent receives the notification to start the execution, part of the process involves verifying that all the edge dependencies are registered by calling the function and verifying that there are no cycles with the function.

(verifyThatAllOperationsHaveBeenRegistered(tasks))

        let verifyThatAllOperationsHaveBeenRegistered (tasks:Dictionary<int, TaskInfo>) =  
            let tasksNotRegistered =             
                tasks.Values                   
                |> (Seq.collect (fun f -> f.Edges) >> set)  
                |> Seq.filter(tasks.ContainsKey >> not)  
            if tasksNotRegistered |> Seq.length > 0 then  
                let edgesMissing = tasksNotRegistered |> Seq.map (string) |> Seq.toArray   
                raise (InvalidOperationException(sprintf "Missing operation: %s" (String.Join(", ", edgesMissing))))

verifyThereAreNoCycles(tasks)

        let verifyTopologicalSort(tasks:Dictionary<int, TaskInfo>) =  
            // Build up the dependencies graph  
            let tasksToFrom = new Dictionary<int, MList<int>>(tasks.Values.Count, HashIdentity.Structural)  
            let tasksFromTo = new Dictionary<int, MList<int>>(tasks.Values.Count, HashIdentity.Structural)  
  
            for op in tasks.Values do  
                // Note that op.Id depends on each of op.Edges  
                tasksToFrom.Add(op.Id, new MList<int>(op.Edges))  
                // Note that each of op.Dependencies is relied on by op.Id  
                for deptId in op.Edges do  
                    let success, \_ = tasksFromTo.TryGetValue(deptId)   
                    if not <| success then tasksFromTo.Add(deptId, new MList<int>())  
                    tasksFromTo.[deptId].Add(op.Id)  
            // Create the sorted list  
            let partialOrderingIds = new MList<int>(tasksToFrom.Count)  
            let iterationIds = new MList<int>(tasksToFrom.Count)  
                          
            let rec buildOverallPartialOrderingIds() =    
                match tasksToFrom.Count with  
                | 0 -> Some(partialOrderingIds)  
                | \_ ->  iterationIds.Clear()  
                        for item in tasksToFrom do  
                            if item.Value.Count = 0 then  
                                iterationIds.Add(item.Key)  
                                let success, depIds = tasksFromTo.TryGetValue(item.Key)  
                                if success = true then  
                                    // Remove all outbound edges  
                                    for depId in depIds do                                    
                                        tasksToFrom.[depId].Remove(item.Key) |> ignore  
                        // If nothing was found to remove, there's no valid sort.  
                        if iterationIds.Count = 0 then None  
                        else  
                            // Remove the found items from the dictionary and   
                            // add them to the overall ordering  
                            for id in iterationIds do  
                                tasksToFrom.Remove(id) |> ignore  
                            partialOrderingIds.AddRange(iterationIds)  
                            buildOverallPartialOrderingIds()              
            buildOverallPartialOrderingIds()

If any of the validations are not satisfied, the process is interrupted and an error is thrown. For demo purposes only the message is printed in the console, a better solution should be to provide a public handler.

 inbox.Error |> Observable.add(fun ex -> printfn "Error : %s" ex.Message )

If the validation passed successfully, the process starts the execution, checking each task for dependencies thus enforcing the order and prioritization of execution. In this last case the edge task is re-queued into the dagAgent using the “QueueTask” message. Upon completion of a task, we simply remove the task from the graph. This frees up all its dependencies to be executed.

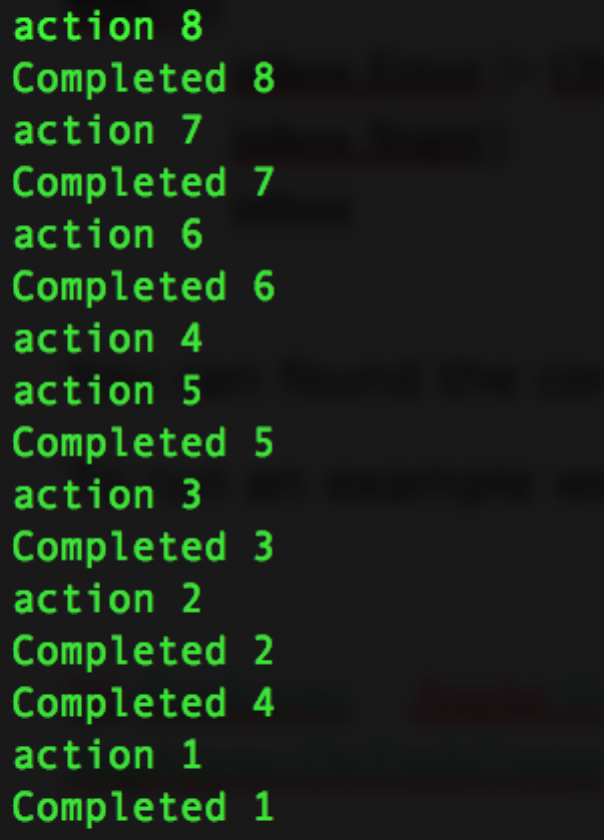
        let dagAgent =  
            let inbox = new MailboxProcessor<TaskMessage>(fun inbox ->  
                let rec loop (tasks : Dictionary<int, TaskInfo>)   
                             (edges : Dictionary<int, int list>) = async {  
                        let! msg = inbox.Receive()  
                        match msg with  
                        | ExecuteTasks ->  
                            // Verify that all operations are registered  
                            verifyThatAllOperationsHaveBeenRegistered(tasks)  
                            // Verify no cycles  
                            verifyThereAreNoCycles(tasks)  
  
                            let dependenciesFromTo = new Dictionary<int, int list>()  
                            let operations' = new Dictionary<int, TaskInfo>()  
  
                            // Fill dependency data structures  
                            for KeyValue(key, value) in tasks do  
                                let operation' =  
                                    { value with NumRemainingEdges = Some(value.Edges.Length) }  
                                for from in operation'.Edges do  
                                    let exists, lstDependencies = dependenciesFromTo.TryGetValue(from)  
                                    if not <| exists then   
                                        dependenciesFromTo.Add(from, [ operation'.Id ])  
                                    else  
                                        dependenciesFromTo.[from] <- (operation'.Id :: lstDependencies)  
                                operations'.Add(key, operation')  
                             
                              
                            operations' |> Seq.filter (fun kv ->  
                                                   match kv.Value.NumRemainingEdges with                                                   
                                                   | Some(n) when n = 0 -> true  
                                                   | \_ -> false)  
                                        |> Seq.iter (fun op -> inbox.Post(QueueTask(op.Value)))  
                            return! loop operations' dependenciesFromTo  
                       
                        | QueueTask(op) ->  
                                Async.Start <| async {   
                                    // Time and run the operation's delegate  
                                    let start' = DateTimeOffset.Now  
                                    match op.Context with  
                                    | null -> op.Task()  
                                    | ctx ->  
                                        ExecutionContext.Run(ctx.CreateCopy(),  
                                                                (fun op -> let opCtx = (op :?> TaskInfo)  
                                                                           (opCtx.Task())), op)  
                                    let end' = DateTimeOffset.Now  
                                    // Raise the operation completed event  
                                    onTaskCompleted.Trigger  { op with Start = Some(start')  
                                                                       End = Some(end') }  
                                      
                                    // Queue all the operations that depend on the completation   
                                    // of this one, and potentially launch newly available  
                                    let exists, lstDependencies = edges.TryGetValue(op.Id)  
                                    if exists && lstDependencies.Length > 0 then  
                                        let dependentOperation' = getDependentOperation lstDependencies tasks []  
                                        edges.Remove(op.Id) |> ignore  
                                        dependentOperation'  
                                            |> Seq.iter (fun nestedOp -> inbox.Post(QueueTask(nestedOp))) }  
                                return! loop tasks edges  
                        
                        | AddTask(id, op) -> tasks.Add(id, op)  
                                             return! loop tasks edges  
                    }  
                loop (new Dictionary<int, TaskInfo>(HashIdentity.Structural)) (new Dictionary<int, int list>(HashIdentity.Structural)))  
            inbox.Error |> Observable.add(fun ex -> printfn "Error : %s" ex.Message )  
            inbox.Start()  
            inbox

You can find the complete code implementation here.

To run an example we can replicate the edge dependencies in the figure above.

let dagAsync = Async.DAG.ParallelTasksDAG()  
dagAsync.OnTaskCompleted |> Observable.add(fun op ->  System.Console.ForegroundColor <- ConsoleColor.Red  
                                                      printfn "Completed %d" op.Id)  
  
dagAsync.AddTask(1, acc1, 4,5)  
dagAsync.AddTask(2, acc2, 5)  
dagAsync.AddTask(3, acc3, 6, 5)  
dagAsync.AddTask(4, acc4, 6)  
dagAsync.AddTask(5, acc5, 7, 8)  
dagAsync.AddTask(6, acc6, 7)  
dagAsync.AddTask(7, acc7)  
dagAsync.AddTask(8, acc8)  
dagAsync.ExecuteTasks()

The ouput should be



I hope you find this to be a useful example of how you can leverage parallelism to optimize your applications.