Apollo

Scheduling specification

Confidential

# Disclaimer

This specification is not, by any stretch of the imagination, complete. It will need to be revised several times before it is complete. Currently several major parts are either missing or incomplete. This disclaimer will be updated to reflect any change in these sections. Finally a specification is supposed to be a ‘living’ document and therefore never complete.

# Introduction

The goal of the scheduling system is to execute a sequence of data transformations for the purpose of determining the values of one or more variables. A schedule is a directed, possibly cyclic, graph that describes the order in which the data transformations need to be executed in order to obtain the desired end result. A schedule can range from a single transformation with no side effects to a complex set of interacting transformations. gives an example of a simple schedule with a single cycle.

Figure : An example of a simple schedule

The scheduling system consists of several parts.

* **Schedule graph:** The graph defines the different actions that should be taken and how the actions are related. Each graph consists of a set of schedule nodes connected by one or more edges. Each node indicates a type of action that should be executed when an executor reaches the given node. Each node is connected to one or more nodes by an edge, but two nodes can only directly be connected via one edge. An edge can have a condition which indicates if that edge is passable at the moment. Executors will not follow an edge which is not passable.
* **Schedule executor:** The executor walks the *schedule graph* and executes the different actions given by the nodes. Executors are created when the user wants to start executing a schedule and destroyed when the execution is finished. Each executor is linked to one or more **execution token(s)** which are used to control the executor.
* **Schedule manipulators:** The manipulators are able to build and update the *schedule graphs*.
* **Schedule verifiers:** The verifiers ensure that the *schedule graph* contains no undesired constructs. Examples of undesired constructs are for instance:
  + Nodes without any connecting edges.
  + Graphs with no designated start and end.
  + Graphs with irresolvable cyclic dependencies.

## Schedule graph

As explained earlier a schedule describes the relation between different data transformations. The graph achieves this goal by connecting two or more graph nodes, each of which describes a single transformation, via a set of graph edges, each of which indicates if the step from the edge starting point to the edge endpoint can be made. In this way the schedule graph provides the ability to encapsulate a group of, possibly simple, data transformations as a single large data transformation. In some cases it may be necessary to nest data transformations so that from the perspective of the user each transformation in the schedule graph is a simple one, while in reality the schedule contains a, possibly highly, complex step. In order achieve this a schedule can be embedded in another schedule. The first schedule becomes a sub-schedule which is able to provide the parent schedule with the desired information without complicating the parent schedule (for an example see ).

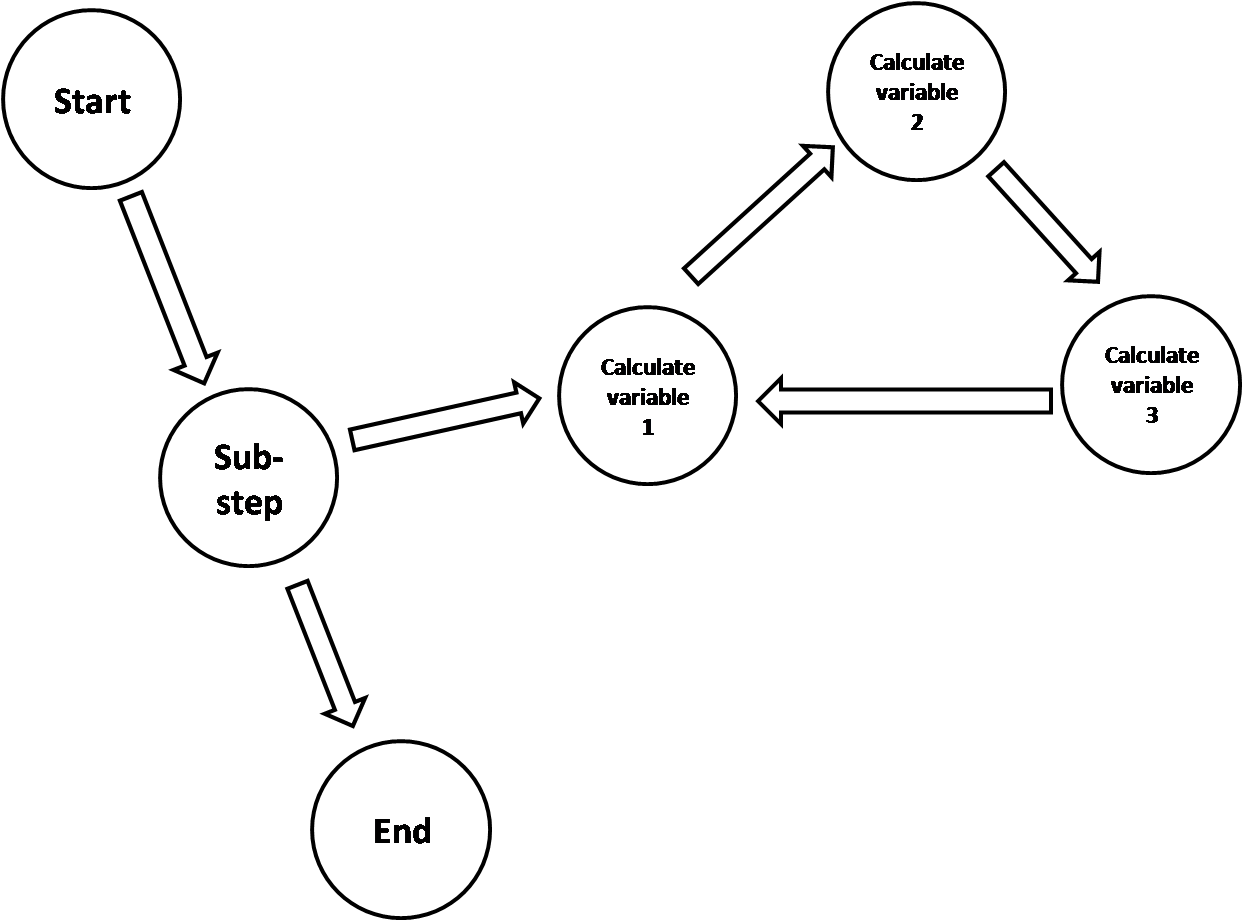


Figure : Simple schedule with sub-schedule

Besides the simplifying nature of using sub-schedules further benefits of sub-schedules are:

* Sub-schedules are considered an atomic unit which provides for easy grouping of related actions.
* Sub-schedules may be executed synchronously or asynchronously meaning that it is possible for the schedule execution to wait for the result directly upon executing, at a later stage, marked by a special node, or not at all, which could be useful for independent data.
* Sub-schedules may also be executed in parallel or even in a child dataset, thus providing simple means of distributing calculations.

In order to be able to handle a wide variety of data transformations only a small number of node types have to be defined. These are:

* **Start node:** Each graph has exactly one of these nodes which defines the only point on the graph where the executer may start with the execution of the schedule. The *start node* is only connected via edges which lead away from the start node.
* **End node:** Each graph has exactly one of these nodes which defines the only point on the graph where the executor will finish the execution of the schedule, providing that there are no errors or intervention from the user during the schedule execution.
* **Action node:** Each graph can have zero or more of these nodes which hold exactly one data transformation action.
* **Sub-schedule node:** Each graph can have zero or more of these nodes which points to a sub-schedule. The node meta-data indicates the conditions for executing the sub-schedule. For instance it may be that the sub-schedule must be executed synchronously in the current dataset, or the sub-schedule execution may be farmed out to one or more child datasets.
* **Synchronize node:** Each graph can have zero or more of these nodes which indicate to the executor that it should wait for one or more sub-schedules to finish their work.
* **Revert history node:** Each graph may contain zero or more of these nodes which when accessed send a request to the history system to revert the timeline to a previously known time marker. These nodes are used to revert to a certain previous state if a calculation becomes numerically unstable.

Besides storing information about the specific action each node may also store additional metadata. Examples of this meta data are:

* The ability to parallelize the actions on the node.
* The ability to push the actions on the node to one or more child datasets.
* What parameters the current node depends on.
* What data is produced by the node.

Finally each data transformation node (i.e. the action node and the sub-schedule node) may provide a set of parameters which determine how many times the data transformation must be executed.

## Schedule executor

The executor is responsible for traversing the *schedule graph* and executing the actions attached to each node that is visited. In order to traverse the graph the executor starts at the *start node* and then follows the directional edges until either the *end node* is reached or until an error condition occurs. The rules for traversing the graph are:

* When a node is reached the action on that node is executed in a synchronous context.
* If the node splits of the calculation to another thread or another dataset then the executor will wait for that calculation to finish unless there is a matching synchronization node present.
* Once the action on a node is finished the executor will iterate past the available exit edges (i.e. the edges that lead away from the current node) in priority order. The first edge that allows passage will be taken.
* For each node that the executor passes a passing count is registered. This passing count can be used as iteration counter and it is also used to detect infinite loops. This iteration counter will only be reset once the executor stops executing.
* If the executor encounters an error condition from which no recovery is possible then the executor will jump straight to the end node while noting the cause for the error, the node on which the error occurred (which may or may not be the node that actually caused the error in the first place) and the error information.

While an executor is running the entire dataset that contains the schedule is locked against changes from the outside. The only external commands that are allowed are commands that just request data but have no side-effects.

The executor stops the processing of the schedule in the following situations:

* The executor has reached the *end node*. The execution of the graph is thus finished and no further nodes will be processed.
* The executor encounters an error condition. In this case the executor will immediately move to the *end node* while storing all the information about the error.
* The user requests that the executor stops the execution of the graph. The execution stop may either be:
  + An immediate stop. In this case the executor will terminate all running processes and return control back to the user as fast as possible. This type of stop request may lead to corrupted state in the dataset.
  + Stop the execution as soon as the current node is finished working. The state of the dataset is unknown. It may be possible that some or all state is corrupted if stopping on the current node leaves the dataset in an undesirable state.
  + Stop the executor as soon as the current block is finished. The state of the dataset should not be corrupted.

Once the graph execution process is stopped it cannot be resumed from the node where the stop occurred. Restarting the execution always starts the execution from the *start node*.

Finally it is possible for the user to request that the executor pauses the execution of the graph. This is the only case where the execution of the schedule may be resumed exactly where it was halted. Note that pausing the schedule execution does not unlock the dataset so no changes are allowed to be made.

## Schedule creation

From the users perspective the goal of executing a schedule is to determine the values of a set of variables. Most likely the user doesn’t really care how these values are found as long as they are correct. The Apollo user interface allows users to select a set of variables for which the values need to be determined. The schedule creator will then create a schedule which is able to obtain the desired information. The creation of a schedule in this way is done by selecting the desired schedule actions from the library and then combining these actions in to a schedule that yields the desired variable values.

The schedule library contains all the different schedule actions that are known to the system. Only the actions in the library can be used for the creation of any kind of schedule. The entries in the schedule library may be templated in order to allow adaption of the schedule for specific situations. If a schedule action is a template then the schedule creator will ensure that the entry is turned into a concrete schedule element before insertion into the schedule.

The schedule actions in the library are provided both by the system and by the components that have been or will be loaded. The following types of schedule actions are found in the library.

* **Single schedule actions:** These are concrete actions which are mostly provided by the components.
* **Sub-schedule actions:** These can either be template or concrete sub-schedules which are only provided by the components. If the sub-schedule is templated then the schedule creator will provide a concrete version upon insertion into the final schedule.
* **Revert history actions:** These are templates provided by the system. The template allows adaption of the final schedule action by the schedule creator or by the component that requests the use of one of the *revert history actions*.
* **Synchronization actions:** These are templates provided by the system. The template provides the ability to stop the schedule execution until the values for a given set of variables are available. These values may come from a parallel calculation or from an external process.

In order to select and order the schedule actions it is necessary to know what variables are resolved by a given schedule action and what dependencies are required for that specific schedule action. For this purpose every schedule action defines the following meta-data:

* **Resolved variables:** The collection of variables for which a value has been determined after the current schedule action has completed successfully.
* **Dependencies:** The collection of dependencies which are required for the current schedule action to be able to execute successfully.
* **Ability to parallelize the node:** Some schedule actions can be run in parallel with other schedule actions. The schedule action needs to indicate this.
* **Ability to split action to child dataset:** Some schedule actions can be split to child datasets.
* **Parameters:** The collection of parameters and their range. When the schedule action is executed it will be executed exactly once for each combination of parameters.

Based on the available meta-data the selection of the schedule actions is done via the following method:

1. Determine which variables are available directly from the problem as specified by the user.
2. Determine which variables need to be resolved
3. For each variable determine if there is a schedule action that can directly resolve this variable. If there are multiple schedule actions that resolve for that given variable, collect all and sort them one or more of:
   1. Estimated execution velocity. This velocity may be a complete guess as long as it is able to roughly rank schedule actions.
   2. Number of other variables that are solved by each schedule action.
   3. The relative accuracy of the different schedule nodes.
4. For each schedule actions determine the dependencies and resolve those with the same algorithm.
5. Verify that all dependencies and variables are resolved.
   1. If not then error out.
   2. If so then order the selected schedule actions based on their dependencies.

This general algorithm should be capable of generating a schedule for most cases. A few sub-cases require special attention. These are:

* **Circular dependencies:** ?????
  + How do we order the variable schedules, especially around a cycle. Usually there is a mathematical preference?
* **Initialization dependencies:** Some schedule actions may need to be preceded by another schedule action, even though that preceding action does not directly provide the values for any of the dependencies. Examples are:
  + A flow solver may solve for a set of variables but it needs those variables to be initialized in some way. That initialization may be done by another schedule action.
  + A boundary condition gives us certain amount of information but we need other information, then what?
* **Asynchronous actions:** These should at some point be followed by a synchronization action. This synchronization action should be placed directly before the first schedule action which depends on the outcome of the asynchronous schedule action.
* **History revert actions:** These can only be inserted if:
  + There is only one stream of actions running. This means that there shouldn’t be any local parallel computations. Running child datasets are allowed, although the revert may make these irrelevant.

Besides creating a schedule based on the resolution of simulation variables there is another way of creating a schedule. This other method combines a series of schedule actions in a predetermined order. This method of creating schedules leads to so called *fixed schedules*, schedules of which the order of the actions cannot be changed. Fixed schedules will mostly be used to define a group of actions for a component or a group of components. For instance a flow solver may consist of multiple components each of which may define one or more actions. All these actions need to be combined in the right order for the flow solver to function.

In order to allow sub-components to insert their own actions in a fixed schedule the main component, which defines the schedule, can provide extension points or insert points. At these points specific sub-components can insert their own schedule actions of any kind.

## Schedule reordering

This one is tricky we may actually have to find some literature on this. However the implementation of this can wait till after we do all the rest. It may make the system more efficient but it shouldn’t influence the correct running of the system.

## Schedule verification

Before executing a schedule it is necessary to verify that the schedule can actually be executed successfully and that once started the execution will have a possibility of finishing if there is no user intervention or error conditions. In order to achieve this one or more verification algorithms are consulted prior to running the schedule. These algorithms check for the following items:

* A schedule must have start and end node.
* It must be possible to reach the end node from all nodes. This means that each node, except for the end node must have at least one edge leading towards another node and that the entry condition for that edge must not permanently be blocking the entry to the edge.
* Sub-schedules cannot link back to the original schedule
* All schedules must be completely defined before the schedule is started
* Each node is only connected to another node by exactly one edge
* History nodes can only occur ????
* All parameter expansions are actually possible and not out of bounds, e.g. calculate something for every value of a double.