Chapter 6. Using the Cluster Class for Scheduling

Authors: *José Luis Pino*

6.1 Introduction

The Ptolemy kernel has three main facilities to aid in the implementation of scheduling algorithms: generic clustering mechanisms, graph iterators, and classical graph algorithms. In this chapter, we will cover the use of these facilities and some of the important methods currently available in Ptolemy to implement new scheduling algorithms.

6.2 Basic Classes

User-specifications done in Ptolemy are represented internally as a collection of annotated directed graphs that may contain cycles. Nodes in these directed graphs may themselves contain other directed graphs. An *atomic* node is either a `Star,` which defines code to implement the node operation, or a `WormHole`, which has an internal graph that is hidden from the outside. A `WormHole` is used when there is a change in the semantics between the internal and external graphs, such as a change in the `Domain` or `Scheduler`. For purposes of the outside graph, a `WormHole` is equivalent to a `Star`. A *non-atomic* node, or `Galaxy,` is a node which contains an internal graph which is visible from the outside. This internal graph is stored in a `Galaxy`'s `BlockList`. Finally, a `Scheduler` is a class that determines the firing order of *atomic* nodes in a graph.

`WormHoles`, `Galaxies` and `Stars` are all derived from the class `Block`. A `Block` contains `PortLists`, which store a list of `PortHoles` that provide locations to connect input or output arcs to the `Block`. `Blocks` also contain `StateLists,` which may either be user-specified parameters or run-time states that are used when a graph is executed.

A user specification is compiled into an internal representation known as an *interpreted universe* (`InterpUniverse`). Currently, the user specifications are in the form of ptcl or oct facets. In the future there will probably be also a Tycho specification format. An `InterpUniverse` captures the user hierarchy in the form of a directed graph of `WormHoles`, `Galaxies` and `Stars`. The `InterpUniverse` is derived from `Galaxy` and contains the top-level user-specification in its `BlockList`. Every other level of the user specified hierarchy is represented by either a `Wormhole` or `Galaxy` embedded inside of its *parent* `Galaxy`.

All `Blocks` have a `parent Block` pointer. The parent of a `Block` is the `Galaxy` or `WormHole` in which the `Block` is embedded. The `InterpUniverse`, which is the top-level `Galaxy` user specification, has its `parent` pointer set to `NULL`.

6.3 Galaxies and their relationship to Adjacency Lists

To define graph algorithms, adjacency-lists and adjacency-matrices are commonly used to represent a directed graph [Cor90]. An adjacency-matrix is a square matrix where there is one column, *i*, and one row, *j*, for each node, *i*, the graph. An element *(i, j)* in this matrix is either 1 if there is an arc from *i* to *j*, or 0 if no arc exists. The second representation is an adjacency-list in which each node has a list containing the nodes to which it is connected. Thus an adjacency-list is better suited for sparse graphs, whereas adjacency-matrices are well suited for dense graphs.

`Blocks` with their `PortLists` can be viewed as equivalent to the adjacency-list data structure. A `PortHole`, in most domains, is either an input or an output. It contains a `farSidePort` pointer to the `PortHole` it is connected to (`NULL` if it is not connected). To traverse the adjacency-list, a scheduler writer must make use of two iterators in Ptolemy (See “Iterators” on page 3-10): `GalStarIter` and `SuccessorIter`. By using a `GalStarIter` a scheduler writer can iterate over the nodes in the user-specified graph. Then on each of these nodes we can find the adjacent nodes using the `SuccessorIter`. Although it is not necessary for adjacency-list equivalence, the `PredecessorIter` is provided to iterate over the nodes that are predecessors to a given node.

There is slight overhead in accessing the graph using both `GalStarIter` and `SuccessorIter` over a straight forward implementation of an adjacency-list class. This overhead has a constant cost which is not dependent on the size of the graph. Thus we feel that the robustness achieved by not having two parallel representations of the same graph far outweigh this small overhead.

6.4 Clustering

Clustering is often used in implementing scheduling heuristics. We have provided a generic `Cluster` class in the Ptolemy kernel which scheduler writers can use directly or, if need be, derive specialized clustering classes. The older schedulers such as the BDF scheduler and the SDF loop schedulers have not been upgraded to use the new `Cluster` classes. Thus, the BDF and SDF schedulers should not be used as examples of how to do clustering in Ptolemy, but rather the hierarchical SDF parallel scheduler (`$PTOLEMY/src/domains/cg/hierScheduler`) can be used as a model. The `HierScheduler` in the current version of Ptolemy is a prototype of the hierarchical parallel scheduler detailed in [Pin95]. In addition, we have a specialized loop scheduler [Mur94] which also uses the new cluster facilities.

The class `Cluster` is derived from the `DynamicGalaxy` and as such manages its own memory. The `Cluster` classes use `ClusterPorts` which are derived from `GalPort`. The main difference between the `ClusterPorts` and `GalPorts` is that `ClusterPorts` maintain a `farSidePort` pointer. We need this change in `ClusterPort` in order to easily iterate over the `Clusters` at any level of the clustering hierarchy. A `ClusterPort::farSidePort` pointer will only be `NULL` if the `ClusterPort` is aliased to a `Star PortHole` connected at higher level of the clustering hierarchy.

6.4.1 Initialization — Flattening the User Specified Graph

Clustering is done directly on the internal representation of the user-specified graph. Before we can begin to cluster the internal representation, the irrelevant user hierarchy must be flattened. The flattening is accomplished by creating a temporary `Cluster` instance and then invoking the `Cluster::initializeForClustering` method on the `Galaxy` whose internals we want to cluster. This top-level `Galaxy` will remain intact, but all internal `Galax-`

ies which pass the Cluster::flattenGalaxy test will be flattened and deleted. Thus any Scheduler and Target pointers to the top-level Galaxy will not need to be updated because they do not change. The necessary information from the user-specified hierarchy is preserved automatically with the aid of the Scope class detailed in section 6.5.

After the internals of the top-level Galaxy have been flattened, Clusters are constructed around each individual atomic Block. In that way, the scheduler writer can treat all the Blocks at each level (except the innermost level) as a Cluster. This property is maintained through any sequence of merge/absorb calls. An example initializeForClustering invocation is shown in figure 6-1, frames 1 and 2.

A facility for restoring the internal Ptolemy representation back to the original user-specified hierarchy is detailed in section 6.6.

6.4.2 Absorb and Merge

The basic clustering mechanisms are implemented with the virtual methods: Cluster::merge and Cluster::absorb. Both of these methods can take up to two arguments. The first argument is the Cluster to absorb/merge and the second argument(optional) specifies whether or not to remove the absorbed or merged Cluster from the original parent Galaxy.

The Cluster::merge method takes the contents of the Cluster being merged and moves them into the Cluster pointed to by the this pointer. A merge operation is communicative. A series of merge steps is shown in figure 6-1 frames 3 and 4.

The Cluster::absorb method takes the Cluster being absorbed and moves it into the Cluster pointed to by the this pointer. Unlike merge, absorb is not communicative as shown in figure 6-1 frames 5 and 5'.

The absorbed or merged Cluster is removed from the original parent Galaxy by default when Cluster::merge or Cluster::absorb is called. We provide three ways to update the graph after a clustering operation with differing levels of efficiency. These methods are detailed in the table 6-1. We first list some variable definitions:

- Let N be defined as the number of Clusters in the parent Galaxy
- Let E_t be defined as the number of PortHoles in *this* Cluster
- Let E_c be defined as the number of PortHoles in the Cluster to absorb or merge

Deletion/Update Method	Complexity to update at each clustering step
Using merge/absorb in their default mode of operation. This is the most inefficient way to do clustering.	$O(N + E_t \times E_c)$

TABLE 6-1: Complexity cost of absorb/merge step.

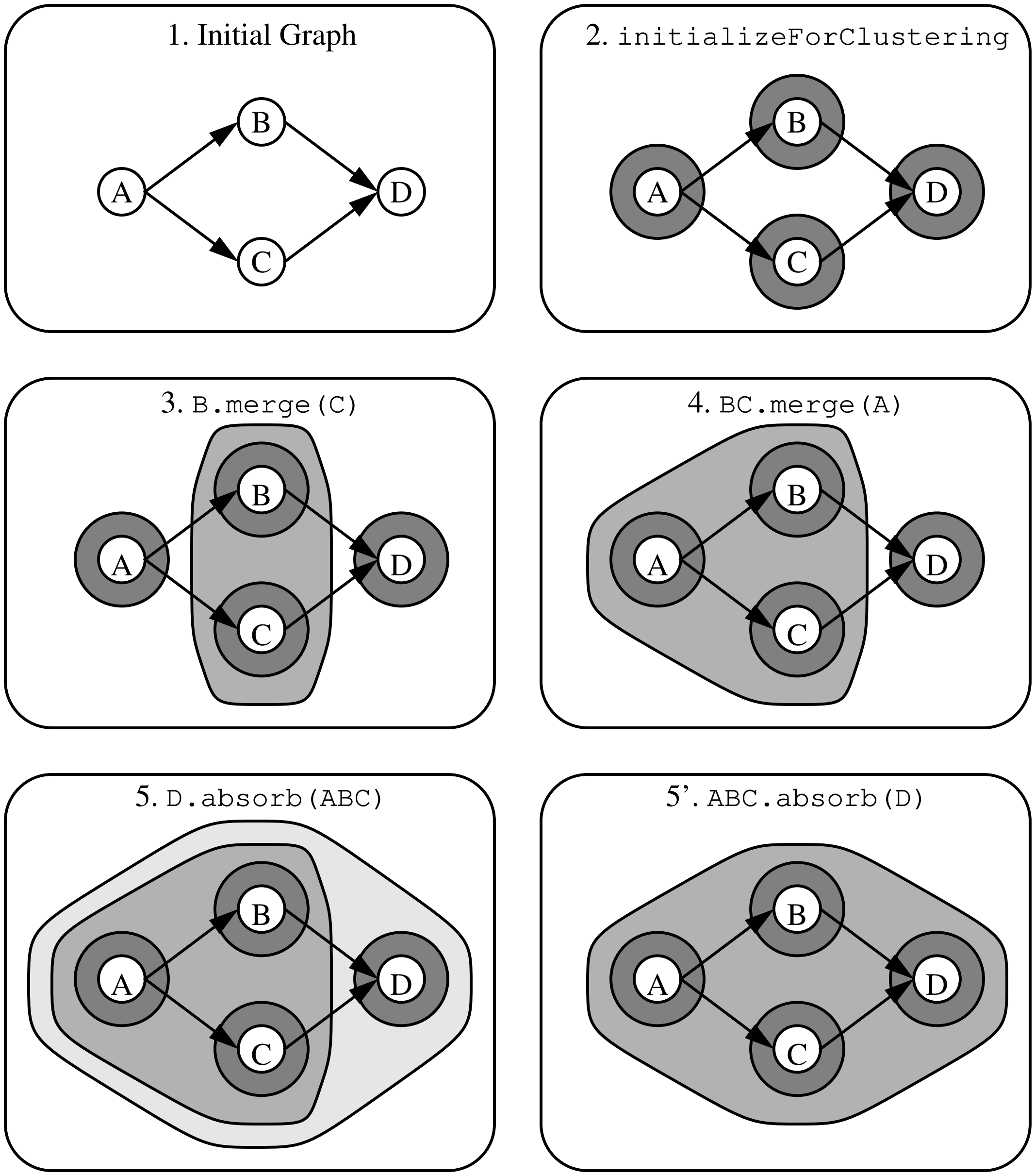

FIGURE 6-1: A five step clustering example. By convention, a `Cluster` in this figure will be named by the listing of its innermost atomic `Block`s. In frame 1, the user-specified graph is shown. `Cluster::initializeForClustering` is called and the resultant graph is shown in frame 2 — this step adds a `Cluster` around all atomic `Block`s. Frames 3-5 show a series of merge/absorb operations. The ordering is important only with absorb operation — as shown by frames 5 and 5'.

Deletion/Update Method	Complexity to update at each clustering step
GalTopBlockIter::remove We can use this method if the Cluster to absorb/merge was found using a GalTopBlockIter (or derived iterator class) on the parent Galaxy. The scheduler writer needs to do two things: • remove the absorbed/merged cluster using from the parent Galaxy using the iterator's remove method. • delete the removed Cluster using the C++ operator delete. This is the most efficient way of updating the graph after a clustering operation — but it is not always possible because we may be traversing the graph in some other way such as using a SuccessorIter.	$O(E_t \times E_c)$
cleanupAfterCluster (defined in Cluster.{h,cc}) If we cannot use the previous method, we can leave the Cluster in the parent Galaxy list (it will be marked invalid automatically). The Cluster iterator classes automatically skip these invalid Clusters. Periodically (but not at each clustering step), the cleanupAfterCluster function should be invoked to remove and delete the invalid Clusters. This function will cost $O(N + E_t \times E_c)$ to execute, but since it is not done at each clustering step — the result on the overall complexity will be additive versus being multiplicative. For an example of how this is done, refer to: $PTOLEMY/src/domains/cg/hierScheduler/HierScheduler.cc.	$O(E_t \times E_c)$

TABLE 6-1: Complexity cost of absorb/merge step.

6.4.3 Cluster Iterator Classes

The Cluster iterator classes assume that all Blocks in the Galaxy being iterated on are Clusters. This property is TRUE assuming that the Galaxy (or one of its parent Galaxies) has been properly initialized (section 6.4.1) and merge/absorb have been the only functions that have modified the topology of the graph since the initialization. These iterators ignore pointers to invalid Clusters which have been left in the Galaxy using merge/absorb with the removeFlag set to FALSE (last two cases in table 6-1). The cluster iterators are listed in table 6-2.

Iterator	Description
ClusterIter	Iterate over all valid Clusters in the given Galaxy.
SuccessorClusterIter	Iterate over all successor (adjacent) Clusters for a given Cluster.

TABLE 6-2: Cluster Iterators

Iterator	Description
PredecessorClusterIter	Iterate over all predecessor Clusters for a given Cluster.

TABLE 6-2: Cluster Iterators

6.5 Block state and name scoping hierarchy

Recall, that when we initialize a Galaxy for clustering, we flatten the original user-specified hierarchy. Before this action, we extract the important information in the hierarchy using the Scope class. In this section we detail this class. The details in this section, however, are not necessary to understand clustering in Ptolemy.

Blocks inherit states from their parent. The Scope class makes it possible for a Target or Scheduler to change the Block hierarchy by saving the inherited states in the user-specified hierarchy. The scoping hierarchy was first released in Ptolemy 0.6, and is only created when the static method Scope::createScope(Galaxy&) is invoked. Currently, the only code that uses the scoping hierarchy is the Cluster class.

The Scope class manages its memory. Once a Scope is created, it will not be deleted until all Blocks within the given Scope are deleted. The Scope class is privately derived from Galaxy. To turn on scoping a programmer simply calls the static method:

```
static Scope* Scope::createScope(Galaxy&)
```

This method constructs a parallel tree corresponding to each Galaxy and copies the StateList and name() for each level.

6.6 Resetting an InterpUniverse back to actionList

Ptolemy 0.6 and later includes the ability to reset an InterpUniverse back to the original user-specification. Resetting is occasionally necessary to undo certain operations done on a universe by a Scheduler or Target. An example is in parallel scheduling, where the original stars in the InterpUniverse are moved to the subGalaxies for the child Targets (see $PTOLEMY/src/domains/cg/parScheduler/ParProcessors.cc). To signal that a the InterpUniverse needs to be rebuilt upon the next run, the scheduler writer should invoke Target::requestReset().

6.7 References

[Cor90] Cormen, Leiserson and Rivest, *Introduction to Algorithms*, New York: MIT Press, 1990.

[Mur94] Murthy, Bhattacharyya, and Lee, *Combined code and data minimization for synchronous dataflow programs*, Memorandum UCB/ERL M94/93, University of California at Berkeley, December,1994. (http://ptolemy.eecs.berkeley.edu/papers/jointCodeDataMinimize)

[Pin95] Pino, Bhattacharyya, and Lee, *A Hierarchical Multiprocessor Scheduling Framework for Synchronous Dataflow Graphs* Memorandum UCB/ERL M95/36, University of California at Berkeley, May, 1995. (http://ptolemy.eecs.berkeley.edu/papers/hierStaticSched)