Chapter 15. Parallel Schedulers

Authors: *Soonhoi Ha*

Base classes for parallel schedulers can be found in `$PTOLEMY/src/domains/cg/par-Scheduler`. All parallel schedulers use an APEG as the input. The APEG for parallel schedulers is called ParGraph, which is derived from class ExpandedGraph. Class ParNode, derived from class EGNode, is a node in a ParGraph.

The base scheduler object is ParScheduler. Since it is derived from class SDFScheduler, it inherits many methods and members from the SDFScheduler class. The ParScheduler class has a ParProcessors class that has member methods to implement the main scheduling algorithm. The ParProcessors class has an array of UniProcessor classes. The UniProcessor class, privately derived from class DoubleLinkList, is mapped to a processing element in the target architecture.

Note that all parallel scheduling algorithms are retargettable: they do not assume any specific topology while they take the effect of topology into account to estimate the interprocessor communication overhead.

Refer to class ParScheduler, to see the overall procedure of parallel scheduling. Refer to class UniProcessor, to see the procedure of sub-universe generations.

15.1 ParNode

This class represents a node in the APEG for parallel schedulers, thus contains additional members for parallel scheduling besides what are inherited from class EGNode. It has the same two-argument constructor as class EGNode.

`ParNode(DataFlowStar* Mas, int invoc_no);`

Initializes data members. If the argument star is at the wormhole boundary, we do not parallelize the invocations. Therefore, we create precedence relations between invocations by calling `claimSticky` of EGNode class in the constructor. If this constructor is called, the `type` protected member is set 0.

The ParNode class has another constructor with one argument.

`ParNode(int type);`

The scheduling result is stored in UniProcessor class as a list of ParNodes. This constructor is to model idle time (*type* = 1), or communication time (*type* = -1 for sending time, *type* = -2 for receiving time) as a ParNode. It initializes data members.

15.1.1 ParNode protected members

`int StaticLevel;`

Is set to the longest execution path to a termination node in the APEG. It defines the static level (or priority) of the node in Hu's scheduling algorithm. Initially it is set to 0.

`int procId`
Is the processor index on which this ParNode is scheduled. Initially it is set to 0.

```
int scheduledTime;
int finishTime;
```
Indicate when the node is scheduled and finished, respectively.

`int exTime;`
Is the execution time of the node. If it is a regular node (*`type`* = 0), it is set to the execution time of the original DataFlowStar. Otherwise, it is set to 0.

`int waitNum;`
Indicates the number of ancestors to be scheduled before scheduling this node. during the scheduling procedure. Initially it is set to 0. At a certain point of scheduling procedure, we can schedule a ParNode only when all ancestors are already assigned, or `waitNum` is 0.

```
EGNodeList tempAncs;
EGNodeList tempDescs;
```
These list members are copies of the ancestors and descendants of the node. While EGGateLists, `ancestors` and `descendants`, may not be modified during scheduling procedure, these lists can be modified.

15.1.2 Other ParNode public members

```
void assignSL(int SL);
int getSL();
virtual int getLevel();
```
The first two methods set and get the `StaticLevel` member. The last one returns the priority of the node, which is just `StaticLevel` by default. In the derived classes, this method can be redefined, for example in Dynamic Level Scheduling, to return the dynamic level of the node.

`int getType();`
Returns the type of the node.

```
void setProcId(int i);
int getProcId();
```
These two methods set and get the `procId` member.

```
void setScheduledTime(int i);
int getScheduledTime();
void setFinishTime(int i);
int getFinishTime();
```
These methods are used to set or get the time when the node is scheduled first and finished.

```
void setExTime(int i);
int getExTime();
```
These methods are used to set and get the execution time of the node.

```
void resetWaitNum();
void incWaitNum();
```
Resets the `waitNum` variable to the number of ancestors, and increases it by 1.

```
int fireable();
```
This method decreases `waitNum` by one, and return `TRUE` or `FALSE`, based on whether `waitNum` reaches zero or not. If it reaches 0, the node is declared "fireable".

```
void copyAncDesc(ParGraph* g, int flag);
void removeDescs(ParNode* n);
void removeAncs(ParNode* n);
void connectedTo(ParNode* to);
```
The first method initializes the lists of temporary ancestors and descendants, `tempAncs` and `tempDescs`, from `ancestors` and `descendants` that are inherited members from EGNode class. List `tempAncs` is sorted smallest `StaticLevel` first while list `tempDescs` is sorted largest `StaticLevel` first. The first argument is necessary to call the sorting routine which is defined in the ParGraph class . By virtue of sorting, we can traverse descendant with larger `StaticLevel` first. If the second argument is not 0, we switch the lists: copy `ancestors` to `tempDescs` and `descendants` to `tempAncs`.

The second and the third methods remove the argument node from the temporary descendant list or from the temporary ancestor list. In the latter case, we decrease `waitNum` by one.

The last method above is to make a temporary connection between the node as the source and the argument node as the destination. The temporary descendant list of the current node is added the argument node while the temporary ancestor list of the argument node is added the current node (also increase `waitNum` of the argument node by 1).

```
CGStar* myStar();
```
Returns the original DataFlowStar after casting the type to CGStar, star class type of the CG domain.

```
int amIBig();
```
Returns `TRUE` or `FALSE`, based on whether `myStar` is a wormhole or not. Before the scheduling is performed in the top-level graph, the wormhole executes scheduling the inside galaxy and stores the scheduling results in the Profile object . The ParNode keeps the pointer to the Profile object if it is an invocation of the wormhole. In the general context, the node will be considered "Big" if the master star can be scheduled onto more than one processors. Then, the star is supposed to keep the Profile object to store the schedules on the processors. A wormhole is a special case of those masters.

```
void setOSOPflag(int i);
int isOSOP();
```
After scheduling is performed, we set a flag to indicate whether all invocations of a star are assigned to the same processor or not, using the first method. The second method queries the flag. Note that only the *first* invocation has the valid information.

```
void setCopyStar(DataFlowStar* s, ParNode* prevN);
DataFlowStar* getCopyStar();
ParNode* getNextNode();
ParNode* getFirstNode();
int numAssigned();
```

The above methods are used to create sub-universes . When we create a sub-universe, we make a copy of the master star if some invocations are assigned to the processor. Then, these invocations keep the pointer to the cloned star. Since all invocations may not be assigned to the same processor, we maintain the list of invocations assigned to the given processor. The first and second methods set and get the pointer to the cloned star. The first method also make a chain of the invocations assigned to the same processor. The third method returns the next invocation chained from the current node, while the fourth method returns the starting invocation of the chain. The last method returns the total number of invocations in the chain. It should be called at the starting invocation of the chain.

```
void setOrigin(EGGate* g);
EGGate* getOrigin();
void setPartner(ParNode* n);
ParNode* getPartner();
```

These methods manipulate the connection information of communication nodes. If two adjacent nodes in an APEG are assigned to two different processors, we insert communication nodes between them: Send and Receive nodes. As explained earlier, a communication node is created by one-argument constructor. The first two methods are related to which EGGate the communication node is connected. The last two methods concern the other communication node inserted.

15.1.3 Iterators for ParNode

There are two types of iterators associated with ParNode class: ParAncestorIter, ParDescendantIter. As their names suggest, ParAncestorIter class returns the ParNodes in the temporary ancestor list (`tempAncs`), and ParDescendantIter class returns the ParNodes in the temporary descendant list (`tempDescs`).

15.2 Class ParGraph

Class ParGraph, derived from class ExpandedGraph, is an APEG graph for parallel schedulers. It has a constructor with no argument.

```
int createMe(Galaxy& g, int selfLoopFlag = 0);
```

Is the main routine to create and initialize the APEG of the argument Galaxy. Using `createMe` method of the ExpandedGraph class, it creates an APEG. After that, it resets the busy flags of the ParNodes, and calls

```
virtual int initializeGraph();
```

This is a protected method. It performs 4 main tasks as follows. (1) Call a protected method `removeArcsWithDelay` to remove the arcs with delays, and to store the source and the destination nodes of each removed arc into the list of NodePairs . The list is a protected member, named `nodePairs` of SequentialList class.

```
void removeArcsWithDelay();
SequentialList nodePairs;
```
(2) For each node, compute the static level (`StaticLevel`) by calling a protected method `SetNodeSL`.

```
int SetNodeSL(ParNode* n);
```
(3) Sum the execution times of all nodes and save the total execution time to a protected member `ExecTotal`.

```
int ExecTotal;
```
(4) Assign the larger static level than any other nodes to the nodes at the wormhole boundary. This let the parallel scheduler schedules the nodes at the wormhole boundary first.

15.2.1 Other ParGraph protected members

```
EGNode* newNode(DataFlowStar*, int);
```
Redefines the virtual method to create a ParNode associated with the given invocation of the argument star.

```
ostream* logstrm;
```
This is a stream object for logging information.

15.2.2 Other ParGraph public members

```
EGNodeList runnableNodes;
void findRunnableNodes();
```
The list of runnable (or fireable) nodes are stored in `runnableNodes`. The above method is to initialize the list with all root ParNodes.

```
int getExecTotal();
Galaxy* myGalaxy();
```
Returns the total execution time of the graph and the original graph.

```
void setLog(ostream* l);
```
Sets the stream object `logstrm`.

```
void replenish(int flag);
```
This method initialize the temporary ancestor list and descendant list of all ParNodes in the graph.

```
void sortedInsert(EGNodeList& l, ParNode* n, int flag);
```
Insert a ParNode, *n,* into the EGNodeList, *l,* in sorted order. It sorts nodes of highest `StaticLevel` first if *flag* = 1, or lowest `StaticLevel` first if *flag* = 0.

```
void restoreHiddenGates();
```
This method restores the hidden EGGates from `removeArcsWithDelay` method to the initial list, either `ancestors` or `descendants` of the parent node.

```
int pairDistance();
```
After scheduling is completed, it is supposed to return the maximum scheduling distance

between node pairs in `nodePairs` list. Currently, however, it just returns -1, indicating the information is not available.

`~ParGraph();`
The destructor initializes the `nodePairs` list.

15.2.3 Class NodePair

Class NodePair saves the source and the destination ParNodes of an arc.

`NodePair(ParNode*` *`src`*, ParNode* *`dest`*);
`ParNode* getStart();`
`ParNode* getDest();`
The constructor requires two arguments of the source and the destination nodes, while the next two methods return the node.

15.3 Class ParScheduler

Class ParScheduler is derived from class SDFScheduler, thus inherits the most parts of `setup` method. They include initialization of galaxy and computation of the repetition counters of all stars in the SDF graph. It redefines the scheduling part of the set-up stage (`computeSchedule`).

`int computeSchedule(Galaxy&` *`g`*);
Is a protected method to schedule the graph with given number of processors. The procedure is

(1) Let the target class do preparation steps if necessary before scheduling begins.

(2) Check whether the number of processors is 1 or not. If it is 1, we use the single processor scheduling (`SDFScheduler` :: computeSchedule). After we set the target pointer of each star, return.

(3) Form the APEG of the argument galaxy, and set the total execution time of the graph to a protected member `totalWork`.

(4) Set the target pointer of each UniProcessor class .

void mapTargets(IntArray* *array* = 0);

If no argument is given, assign the child targets to the UniProcessors sequentially. If the IntArray argument maps the child targets to the UniProcessors. If array[1] = 2, UniProcessor 1 is assigned Target 2.

(5) Before the main scheduling begins, complete the profile information of wormholes. Since we may want to perform more tasks before scheduling, make this protected method virtual. Be default, return `TRUE` to indicate no error occurs.

virtual int preSchedule();

(6) Perform scheduling by calling `mainSchedule`.

`int mainSchedule();`
This public method first checks whether manual assignment is requested or not. If it is, do

manual assignment. Otherwise, call an automatic scheduling routine which will be redefined in each derived class, actual scheduling class. After scheduling is performed, set the `procId` parameter of the stars in the original galaxy if all invocations are enforced to be assigned to the same processor .

`int assignManually();`
Is a protected method to return `TRUE` if manual assignment is requested, or return `FALSE` otherwise.

`virtual int scheduleManually();`
Is a public virtual method. This method first checks whether all stars are assigned to processors (`procId` parameter of a star should be non-negative and smaller than the number of processors). If there is any star unassigned, return `FALSE`. All invocations of a star is set the same `procId` parameter. Based on that assignment, perform the list scheduling . The `procId` of a Fork star is determined by its ancestor. If the ancestor is a wormhole, the `procId` of the Fork should be given explicitly as other stars.

`virtual int scheduleIt();`
Is a public virtual method for automatic scheduling. Refer to the derived schedulers. By default, it does nothing and return `FALSE` to indicate that the actual scheduling is not done in this class.

`int OSOPreq();`
Is a protected method to return `TRUE` or `FALSE`, based on whether all invocations are enforced to be scheduled on the same processor.

Now, all methods necessary for step (5) are explained. Go back to the next step.

(7) As the final step, we schedule the inside of wormholes based on the main scheduling result if automatic scheduling option is taken. In the main scheduling routine, we will determine how many processors will be assigned to a wormhole.

`int finalSchedule();`
If scheduling of wormholes succeeds, return `TRUE`. Otherwise, return `FALSE`.

15.3.1 compileRun method

`void compileRun();`
Is a redefined public method of SDFScheduler class. It first checks the number of processors. If the number is 0, it just calls `SDFScheduler` :: compileRun. This case occurs inside a wormhole. Otherwise,

(1) Set the target pointer of UniProcessors.

(2) Create sub-universes for each processors.

`int createSubGals(Galaxy& g);`
Is a public method. It first checks whether all invocations of stars are scheduled on the same processor, and set the flag if it is the case. After restoring all hidden EGGates of the APEG, create sub-universes.

(3) Prepare each processor (or UniProcessor class) for code generation. It includes sub-universe initialization, and simulation of the schedule on the processor obtained from the parallel scheduling.

(4) Let the target do something necessary, if any, before generating code.

(5) Generate code for all processors.

15.3.2 Other ParScheduler protected members

```
const char* logFile;
pt_ofstream logstrm_real;
ostream* logstrm;
```

These are for logging information. `logFile` indicates where to store the logging information.

```
MultiTarget* mtarget;
```

Is the pointer to the target object, which is MultiTarget type.

```
int numProcs;
```

Is the total number of processors.

```
ParGraph *exGraph
```

Is the pointer to the APEG used as the input graph to the scheduler.

```
ParProcessors* parProcs;
```

This member points the actual scheduler object. It will be set up in the `setUpProcs` method of the derived class.

```
IntArray avail;
```

This array is to monitor the pattern of processor availability during scheduling.

```
int inUniv;
```

This flag is set `TRUE` when it is the scheduler of a universe, not a wormhole. In the latter case, it is set `FALSE`. By default, it is set `TRUE`.

```
int withParallelStar();
```

This method returns `TRUE` or `FALSE`, based on whether the galaxy contains any wormhole or data-parallel star, or not.

```
int overrideSchedule();
```

If the user wants to override the scheduling result after automatic scheduling, he can set the `adjustSchedule` parameter of the target object. This method pokes the value of that parameter. This is one of the future feature, not implemented yet in Ptolemy due to limitation of the graphical interface, pigi.

15.3.3 Other ParScheduler public members

`ParScheduler(MultiTarget*` *t*, const char* *log* = 0);
`virtual ~ParScheduler();`

The constructor has two arguments: the target pointer and the name of log file name. The

virtual destructor does nothing.

`virtual void setUpProcs(int` *num*);
The number of processors is given as an argument to this method. It will initialize the `avail` array. In the derived class, this method will create a ParProcessors class (set `par-Procs` member).

```
ParProcessors* myProcs();
UniProcessor* getProc(int n);
```
These methods will return the pointer to the ParProcessors object associated with this scheduler and the UniProcessor object indexed by the argument. The range of the index is 0 to `numProcs-1.`

`void ofWorm();`
Resets `inUniv` flag to `FALSE`.

`int getTotalWork();`
Returns the total execution time of the graph.

`void setProfile(Profile*` *profile*);
Copy the scheduling results to the argument Profile . If the scheduling is inside a wormhole, the scheduling results should be passed to the outside of the wormhole by a Profile object.

15.4 class ParProcessors

Class ParProcessors is the base class for all actual scheduler object. Refer to derived classes to see how scheduling is performed. This class just provide the set of common members and methods. Among them, there is a list scheduling routine.

`int listSchedule(ParGraph*` *graph*);
This method performs the list scheduling with the input argument APEG. It should be called after all nodes are assigned to the processors. It is the last routine to be called for all parallel schedulers. It adds communication nodes to the APEG (`findCommNodes`) and schedule them with the regular ParNodes. It returns the makespan of the schedule.

`void findCommNodes(ParGraph*` *graph*);
This method puts a pair of communication ParNodes, a send node and a receive node, on the arc between two nodes assigned to the different processors. Note that we use `tem-pAncs` and `tempDescs` list of ParNode class to insert these nodes instead of modifying the APEG. We store the newly created communication ParNodes in `SCommNodes.` The procedure consists of two stages. In the first stage, all regular arcs in the APEG are considered. The `StaticLevel` of the send node is assigned to that of the source node plus one to ensure that the send node is scheduled right after the source node. The static level of the receive node is assigned to the same value as the destination node. In the second stage, all hidden arcs are considered. In this case, the `StaticLevel` of communication nodes are assigned to 1, the minimum value since they may be scheduled at the end of the schedule. The number of interprocessor requirements are saved in a protected member, `com-`

`mCount`.

`int getMakespan();`
Returns the longest scheduled time among all UniProcessors.

15.4.1 Other ParProcessors protected members

```
int numProcs;
MultiTarget* mtarget;
EGNodeList SCommNodes;
```
These members specify the number of processors, the pointer to the multiprocessor target class, and the list of communication nodes added during `listSchedule`.

`IntArray pIndex;`
Is used to access the processors in the order of available time.

`void scheduleParNode(ParNode*` *`node`*`);`
This method schedules a parallel node (a wormhole or a data-parallel star) inside the `listSchedule` method. Note that the processors are already assigned for the node.

`virtual ParNode* createCommNode(int` *`i`*`);`
Is a virtual method to create a ParNode with type given as an argument. It is virtual since the derived scheduler may want to create a node of derived class of ParNode.

`void removeCommNodes();`
Clears the `SCommNodes` list.

`void sortWithAvailTime(int` *`guard`*`);`
Sort the processors with their available times unless no node is assigned to the processor. All idle processors are appended after the processors that are available at *`guard`* time and before the processor busy at *`guard`* time. Store the results to `pIndex` array.

`int OSOPreq();`
Returns `TRUE` or `FALSE`, based on whether all invocations of a star are enforced to be scheduled on the same processor or not.

15.4.2 Other ParProcessors public members

```
ParProcessors(int, MultiTarget*);
virtual ~ParProcessors();
```
The constructor has two arguments: the number of processors and the target pointer. It creates `pIndex` array and initialize other data structures. The destructor clears `SCommNodes`.

```
void mapTargets(IntArray* array);
void prepareCodeGen();
void createSubGals();
void generateCode();
```
The above methods perform the actual action defined in the ParScheduler class. For description, refer to class ParScheduler. The last method deliver the generate code from

each processor to the target class.

`int size();`
returns the number of processors.

`virtual UniProcessor* getProc(int` *num*);
This method returns the UniProcessor with a given index. It is virtual since the derived class wants to return it own specific class derived from UniProcessor class.

`void initialize();`
Initializes `pIndex, SCommNodes,` and processors.

`StringList display(NamedObj*` *gal*);
`StringList displaySubUnivs();`
These methods return the StringList contains the scheduling result and the sub-universe description.

`ParNode* matchCommNodes(DataFlowStar*, EGGate*, PortHole);`
This method is used in sub-universe generation. The first argument is a communication star, either a send star or a receive star, that the system automatically inserts for interprocessor communication. The second argument is the EGGate that the interprocessor communication (IPC) occurs. If the second argument is `NULL`, the third argument indicates the porthole that the IPC occurs. In case all invocations of any star are assigned to the same processor, the sub-universe creation procedure is greatly simplified: we do not need to look at the APEG, rather look at the original SDF graph to create the sub-universe. It is the case when the second argument becomes `NULL`. This method sets the pointer of the communication star to the corresponding ParNode that are inserted during `listSchedule` method.

15.5 UniProcessor

Class UniProcessor simulates a single processing element, or a processor. It is derived from class DoubleLinkList to hold the list of ParNodes from parallel scheduling. Class NodeSchedule is derived from class DoubleLink to register a ParNode into the DoubleLinkList.

A UniProcessor keeps two target pointers: one for multiprocessor target (`mtarget),` and the other for the processor (`targetPtr).` They are both protected members.
`MultiTarget* mtarget;`
`CGTarget* targetPtr;`
The pointer to the processor can be obtained by a public method:

`CGTarget* target();`
The pointers to the multiprocessor target and to the ParProcessors class that this UniProcessor belongs to, are set by the following method:

`void setTarget(MultiTarget*` *t*, ParProcessors* *p*);

15.5.1 Class NodeSchedule

A NodeSchedule is an object to link a ParNode to a linked list. It indicates whether the node represents an idle time slot or not. It also contains the duration of the node. There is no protected member in this class.

```
void resetMembers();
void setMembers(ParNode* n, int dur);
```

These methods set the information for the associated node: the pointer to the node, duration, and a flag to tell whether it is an idle node or not. In the first method, the idle flag is set FALSE. The constructor also resets all internal information of the class.

```
ParNode getNode();
int getDuration();
int isIdleTime();
```

The above methods return the pointer to the node associated with this class, its duration, and the flag to say TRUE if it represents an idle time slot.

```
NodeSchedule* nextLink();
NodeSchedule* previousLink();
```

These methods return the next and the previous link in the linked list.

15.5.2 Members for scheduling

Since a list scheduling (with fixed assignment) will be performed as the last stage of all scheduling algorithms in Ptolemy , basic methods for list scheduling are defined in the UniProcessor class. In list scheduling, we need the available time of the processor.

```
int availTime;
```

Is a protected member to indicate the time when the processor available. There are public methods to access this member:

```
void setAvailTime(int t);
int getAvailTime();
NodeSchedule* curSchedule;
```

This protected member points to the NodeSchedule appended last to the linked list. There are two public methods to access a NodeSchedule:

```
NodeSchedule* getCurSchedule();
NodeSchedule* getNodeSchedule(ParNode* n);
```

The first method just returns `curSchedule` member while the second one returns the NodeSchedule associated with the argument ParNode.

When a ParNode is runnable earlier than the available time of the processor, we want to check whether there is an idle slot before `availTime` to fit the ParNode in the middle of the schedule:

```
int filledInIdleSlot(ParNode*, int start, int limit = 0);
```

The first two arguments are the pointer to the ParNode to be scheduled and the earliest time when the node can be scheduled. Without the third argument given explicitly, this

method returns the earliest time that the processor is available to schedule the node. If the third argument is given, the available time of the processor should be less than *limit.* If this method could not find an idle slot to schedule the node, it returns -1. Otherwise, it returns the possible scheduling time of the node.

`int schedInMiddle(ParNode* n, int when, int);`
Schedule the node, *n,* at *when* inside an idle-time slot of the processor. The third argument indicates the duration of the node. This method returns the completion time of the schedule if scheduling is succeeded. If it fails to find an idle-time slot at *when* to accommodate the node, it returns -1.

If a node is to be appended at the end of the schedule in a processor,
`void appendNode(ParNode* n, int val);`
Does that blindly. To schedule a non-idle node, we have to use

`int schedAtEnd(ParNode* n, int start, int leng);`
In case *start* is larger than the processor available time, this method put an idle time slot in the processor and calls `appendNode.` And, it sets the schedule information of the node, and increases `availTime` of the processor.

`void addNode(ParNode* node, int start);`
This method is given a ParNode and its runnable time, and schedule the node either inside an idle time slot if possible, or at the end of the schedule list. The methods described above are used in this method.

`void scheduleCommNode(ParNode* n, int start);`
When we schedule the given communication node, *n,* available at *start,* we also have to check the communication resource whether the resources are available or not. For that purpose, we detect the time slot in this processor to schedule the node, and check whether the same time slot is available in the communication resources: we use `scheduleComm` of the multiprocessor target to check it. If we find a time slot available for this processor and the communication resources, we schedule the communication node.

`int getStartTime();`
Returns the earliest time when the processor is busy.

All methods described in this sub-section are public.

15.5.3 Sub-Universe creation

After scheduling is performed, a processor is given a set of assigned ParNodes. Using the scheduling result, we will generate code for the target processor. To generate code for the assigned nodes, we need to allocate resources for the nodes and examine the connectivity of the nodes in the original graph. These steps are common to the generic routine for single processor code generation in which a processor target is given a galaxy. Therefore, we want to create a sub-galaxy that consists of stars of which any invocation is assigned to the processor. Note that a sub-galaxy is NOT a subgraph of the original SDF graph. Besides the stars in the original program graph, we include other stars such as communication stars and spread/collect stars. This

subsection will explain some details of sub-universe creation.

`void createSubGal();`

Is the main public method for sub-universe creation. It first creates a galaxy data structure, `subGal`, a private member. Then, it clones stars of at least one of whose invocations is assigned to the processor. Make a linked list for all assigned invocations (nodes) of each star in order of increasing invocation number, and set the pointer of cloned star. As for a wormhole, we create a CGWormStar instead of cloning the wormhole. A CGWormStar class will replace a wormhole in the sub-universe. If an original star is not supported by the processor target (for example, with heterogeneous scheduling), we create a star with the same name as the original star in the target domain.

In the next step, we connect the cloned stars by referring to the original galaxy. If a star is at the wormhole boundary in the original graph, we connect the cloned star to the same event horizon; by doing so the wormhole in the original graph is connected to the sub-universe. If the star at the wormhole boundary is scheduled on more than one processors (or not all invocations are assigned to the same processor), the wormhole in the original graph will be connected to the last created sub-universe.

If an arc connects two stars who have some invocations assigned to the same processor, we examine whether all of the two stars' invocations are assigned to the same processor. If they are, we just connect the cloned stars in the sub-universe. If they aren't, we have a cloned star of either one star whose invocations are assigned to the current sub-universe. In this case, we create a send star (`createSend` method of the multiprocessor target) or a receive star (`createReceive` of the target), based on whether the cloned star is a source or destination of the connection . We create a communication star and set the communication star pointer of the communication nodes in the APEG , by `matchCommNodes` method of ParProcessors class. If the partner communication star was already created in another sub-universe, we pair the send and receive stars by `pairSendReceive` method of the multiprocessor target .

The last case is when an arc connects two stars whose invocations are distributed over more than one processor. If no invocation of the destination star is assigned to this processor, we call the `makeBoundary` method.

`void makeBoundary(ParNode* src, PortHole* orgP);`

The first argument points to the earliest invocation of the star assigned to this processor, and the second is the pointer to the output porthole in the original SDF graph as the source of the connection. We examine all assigned invocations to check whether the destination invocations are assigned to the same processor. If they are, we create one send star and connect it to the cloned star. Otherwise, we create a Spread star to distribute the output samples to multiple processors. We connect the cloned star to the Spread star, and the Spread star to multiple send stars.

Otherwise, we call the `makeConnection` method.

`void makeConnection(ParNode* dest, ParNode* src, PortHole* ref, ParNode* firstS);`

The first argument is the pointer to the first assigned invocation of the destination star

while the second one is the source node connected to the first argument. The third argument is the pointer to the destination porthole of the connection in the original graph. The last argument is the pointer to the first invocation of the source star. Note that the last argument node may not be assigned to the current processor. This method examines all assigned invocations of the destination star to identify the sources of the samples to be consumed by the cloned star in this processor. If the number of sources are more than one, we create a Collect star and connect the Collect star to the cloned destination star in the sub-universe. For each source in the other processors, we create a receive star and connect it to the Collect star. Similarly, we examine all assigned invocations of the source star to identify the destinations of the samples to be produced by the source star in this processor. If the number of destinations is more than one, we create a Spread star and connect it to the source star. For each destination in the other processors, we create a send star and connect it to the Spread star. As a result, it may occur that to connect two stars in the sub-universe, we need to splice a Spread and a Collect star on that connection.

Spread and Collect stars

A Spread or a Collect star is created by `createSpread` or `createCollect` method of the multiprocessor target. The following illustrates when we need to create a Spread or a Collect star in a sub-universe.

Suppose we have star A connected to star B in original graph. Star A produces two samples and star B consumes one. Then, one invocation of star A is connected to two invocations of star B. If one invocation of star A and only one of invocation of star B are assigned to the current processor. Then, we need to connect the cloned stars of A and B in the sub-universe. We can not connect stars A and B in the sub-universe directly since among two samples generated by star A, one sample should be transferred to another processor through a send star. In this case, we connect a Spread star to star A, and one send star and star B to the Spread star in the sub-universe. Then, star A produces two samples to the Spread star while the Spread star *spread* the incoming two samples to the send star and star B one sample each. The number of output portholes and the sample rate of each porthole are determined during the sub-universe creation. If there is no initial delay on the connection and neither star A nor B needs to access the past samples, the Spread star does not imply additional copying of data in the generated code.

Similarly, we need to connect a Collect star to the destination star if samples to that star come from more than one sources.

15.5.4 Members for code generation

```
void prepareCodeGen();
```

This method performs the following tasks before generating code.

(1) Initialize the sub-universe, which also initialize the cloned stars.

(2) Convert a schedule (or a linked list of ParNodes) obtained from the parallel scheduler to a SDFSchedule class format (list of stars). The code generation routine of the processor target assumes the SDFSchedule class as the schedule output.

(3) Simulate the schedule to compute the maximum number of samples to be collected

at runtime if we follow the schedule. This information is used to determine the buffer size to be allocated to the arcs.

`void simRunSchedule();`
Performs the step (3). It is a protected member.

`StringList& generateCode();`
This method generate code for the processor target by calling `targetPtr->generateCode()`.

`int genCodeTo(Target* t);`
This method is used to insert the code of the sub-universe to the argument target class. It performs the almost same steps as `prepareCodeGen` and then calls `insertGalaxyCode` of the processor target class instead of `generateCode` method.

15.5.5 Other UniProcessor protected members

There are a set of methods to manage NodeSchedule objects to minimize the runtime memory usage as well as execution time.

```
void putFree(NodeSchedule* n);
NodeSchedule* getFree();
void clearFree();
```
If a NodeSchedule is removed from the list, it is put into a pool of NodeSchedule objects. When we need a NodeSchedule object, a NodeSchedule in the pool is extracted and initialized. We deallocate all NodeSchedules in the pool by the third method.

`void removeLink(NodeSchedule* x);`
Removes the argument NodeSchedule from the scheduled list.

`int sumIdle;`
Indicates the sum of the idle time slots after scheduling is completed. The value is valid only after `display` method is called.

15.5.6 Other UniProcessor public members

There are a constructor with no argument to initialize all data members and a destructor to delete `subGal` and to delete all NodeSchedule objects associated with this processor.

```
Galaxy* myGalaxy();
int myId();
DoubleLinkList :: size;
int getSumIdle();
```
The above methods return the pointer to the sub-universe, the index of the current processor, the number of scheduled nodes, and the sum of idle time after scheduling is completed (or `sumIdle`).

`void initialize();`
This method puts all NodeSchedules in the list to the pool of free NodeSchedules and initialize protected members.

`void copy(UniProcessor* org);`
Copies the schedule from the argument UniProcessor to the current processor.

```
StringList displaySubUniv();
StringList display(int makespan);
int writeGantt(ostream& os, const char* universe, int numProcs, int
span);
```
The first method display the sub-universe. The second method displays the schedule in the textual form while the third one forms a string to be used by Gantt-chart display program.

15.5.7 Iterator for UniProcessor

Class ProcessorIter is the iterator for UniProcessor class. A NodeSchedule object is returned by `next` and ++ operator.

`ParNode* nextNode();`
Returns the ParNode in the list.

15.6 Dynamic Level Scheduler

Dynamic Level Scheduling is one of the list scheduling algorithms where the priority of a node is not fixed during the scheduling procedure. The scheduling algorithm is implemented in `$PTOLEMY/src/domains/cg/dlScheduler`. All classes in that directory are derived from the base parallel scheduling classes described above in this chapter. For example, DLNode class is derived from class ParNode, and redefines `getLevel` method to compute the *dynamic* level of the node.

`int getLevel();`
This method returns the sum of the static node and the worst case communication cost between its ancestors and this DLNode.

Class DLNode has the same constructors as class ParNode.

The dynamic level scheduler maintains a list of runnable nodes sorted by the `getLevel` value of the DLNodes. It fetches a node of highest priority and choose the best processor that can schedule the node earliest while taking interprocessor communication into account.

15.7 Class DLGraph

Class DLGraph, derived from class ParGraph, is the input APEG graph to the dynamic level scheduler. It consists of DLNode objects created by redefining the following method:

`EGNode* newNode(DataFlowStar* s, int i);`
This method creates a node in the APEG graph. Here, it creates a DLNode.

DLGraph has a protected member maintaining the number of unscheduled nodes.

```
int unschedNodes;
```

We may check whether the scheduler is deadlocked or not by examining this variable

when the scheduler halts. This can be manipulated by the public methods

```
void decreaseNodes();
int numUnSchedNodes();
```

The first method decrements *unschedNodes* and the second method returns it.

The DLGraph class redefines `resetGraph` method.

```
void resetGraph();
```

This makes the initial list of runnable nodes and sets the variable described above. This method internally calls the following protected method:

```
virtual void resetNodes();
```

This method resets the busy flag and the `waitNum` member of DLNodes.

There are three other public members.

```
DLNode* fetchNode();
```

Fetches a DLNode from the head of the list of runnable nodes.

```
StringList display();
```

Displays the APEG and the list of source nodes.

15.8 class DLScheduler

Class DLScheduler is derived from class ParScheduler. It has a constructor with three arguments.

`DLScheduler(MultiTarget*` *t*, `const char*` *log*, `int` *i*);

The arguments are the pointer to the multiprocessor target, the name of the logging file, and a flag to indicate whether the communication overhead can be ignored or not. If the processor target has special hardware for communication separate from the CPU, then most of the communication can be simultaneous with processor computation. In this case, we do not reserve the communication time slot in the processor schedule but in the access schedule of the communication resources only. This mode of operation is selecte if *i* is set `TRUE`. This is not implemented since we haven't dealt with that kind of architecture yet.

There is a protected member to point to a ParProcessor class:

DLParProcs* parSched;

This pointer is set in the following redefined method:

`void setUpProcs(int` *num*);

This method first performs `ParScheduler::setUpProcs,` and then create a DLParProcs object. While this class defines the overall procedure of the dynamic level scheduling algorithm, the DLParProcs class provides the details of the algorithm .

```
~DLScheduler();
```

Deallocate the DLParProcs object.

The main procedure of the dynamic level schedule is defined in

```
int scheduleIt();
```

This method does the following:

(1) Initializes the DLParProcs and resets the DLGraph and the communication resources of the multiprocessor target.

(2) Fetch a node from the list of runnable nodes until there are no more nodes in the list.

(2-1) If the node is not a parallel node, call `scheduleSmall` method of the DLParProcs class to schedule the node.

(2-2) If the node is the first invocation of a parallel-star node, we give up. NOTE: We do not support parallel stars since we haven't had to deal with them yet.

(3) When there are no more runnable nodes, we check whether the graph is deadlocked. In case of successful completion, we perform an additional list scheduling (`listSchedule` of the ParProcessors class), based on the processor assignment determined by the above procedure.

```
StringList displaySchedule();
```

Displays the final schedule results.

15.9 Class DLParProcs

Class DLParProcs, derived from the ParProcessors class, defines a main object to perform the dynamic level scheduling algorithm. It has a constructor with two arguments:

`DLParProcs(int` *`pNum`*`, MultiTarget*` *`t`*`);`

The arguments are the number of processors and the pointer to the multiprocessor target. This method creates *`pNum`* UniProcessors for processing elements. These UniProcessors are deallocated in the destructor `~DLParProcs()`.

Based on the type of node described in `scheduleIt` method of the DLScheduler class, we call one of the following methods to schedule the node. `scheduleSmall, scheduleBig, copyBigSchedule.`

`virtual void scheduleSmall(DLNode*` *`n`*`);`

This method is a public method to schedule an atomic node that will be scheduled on one processor. This is virtual since the HuParProcs class redefines this method. The scheduling procedure is as follows:

(1) Obtain the list of processors that can schedule this node. Refer to `candidateProcs` method of the CGMultiTarget class . Here, we examine the resource restriction of the processor, as well as the types of stars that a processor supports, in case of a heterogeneous target. If all invocations of a star should be scheduled to the same processor and another invocation of the star is already scheduled, we put only that processor only into the list of candidate processors that can schedule this node.

(2) Among all candidate processors, we select the processor that can schedule this node the earliest. In this stage, we consider all communication overhead if ancestors would be

assigned to the different processors.

(3) Assign the node to that processor:

`void assignNode(DLNode* n, int destP, int time);`

Is a protected method to assign an atomic node (*n)* to the processor of index *destP* at *time.* This method also schedules the communication requirements in the communication resources.

(4) Indicate that the node was fired:

`virtual void fireNode(DLNode* node);`

It is a virtual and protected method. It fires the argument node and insert its descendants into the list of runnable nodes if they become runnable after this node is fired.

(5) Finally, we decrease the number of unscheduled nodes and the total remaining work of the DLGraph class.

The DLNode class has the pointer to the Profile class that determines the inside schedule of the node. We insert idle time slots to the processors to match the pattern of processor availability with the starting pattern of the profile and append the node at the end of the processors. We record the assignment of the profile to the processors in `assignedId` array of the Profile class . We also save the scheduling information in the DLNode: when the node is scheduled and completed, and on which processor it is scheduled. To determine the latter, we select the processor that the inside scheduler assumes the first processor it is assigned.

After appending the profile at the end of the available processors, we fire the node and update the variables of the DLGraph.

`void initialize(DLGraph* graph);`

This method calls `ParProcessors` :: initialize and resets the `candidate` member to the index of the first UniProcessor, and `myGraph` member to *graph* argument.

`DLGraph* myGraph;`

A protected members to store the pointer to the input APEG.

`int costAssignedTo(DLNode* node, int destP, int start);`

This method is a protected method to compute the earliest time when the processor of index *destP* could schedule the node *node* that is runnable at time *start.*

15.10 Hu Level Scheduler

Hu's level scheduling algorithm is a simple list scheduling algorithm, in which a node is assigned a fixed priority. No communication overhead is considered in the scheduling procedure. The code lies in `$PTOLEMY/src/domains/cg/HuScheduler`. All classes, except the HuScheduler class in this directory, are derived from the classes for the dynamic level schedulers.

15.10.1 Class HuNode

Class HuNode represents a node in the APEG for Hu's level scheduling algorithm. It is derived from the DLNode class so that it has the same constructors. The level (or priority) of a node does not depend on the communication overhead.

`int getLevel();`
Just returns `StaticLevel` of the node.

A HuNode has two private variables to indicate the available time of the node (or the time the node becomes runnable) and the index of the processor on which the node wants to be assigned. The latter is usually set to the index of the processor that its immediate ancestor is assigned. There are five public methods to manipulate these private variables.

`int availTime();`
`void setAvailTime(int` *t*`);`
`void setAvailTime();`
`void setPreferredProc(int` *i*`);`
The first three methods get and set the available time of the node. If no argument is given in `setAvailTime,` the available time is set to the earliest time when all ancestors are completed. The last two methods get and set the index of the processor on which the node is preferred to be scheduled.

15.10.2 Class HuGraph

Class HuGraph is the input APEG for Hu's level scheduler. It redefines three virtual methods of its parent classes.

`EGNode* newNode(DataFlowStar*` *s*, int *invoc*`);`
Creates a HuNode as a node in the APEG.

`void resetNodes();`
This method resets the variables of the HuNodes: visit flag, `waitNum,` the available time, and the index of the preferred processor.

`void sortedInsert(EGNodeList&` *nlist*, `ParNode*` *n*, `int` *flag*`);`
In the ParGraph class, this method sorts the nodes in order of decreasing `StaticLevel` of nodes. Now, we redefine it to sort the nodes in order of increasing available time first, and decreasing the static level next.

15.10.3 Class HuScheduler

Class HuScheduler, derived from the ParScheduler class, is parallel to the DLScheduler class in its definition.

`HuScheduler(MultiTarget*` *t*, `const char*` *log*`);`
The constructor has two arguments: one for the multiprocessor target and the other for the log file name.

The HuScheduler class has a pointer to the HuParProcs object that will provide the details of the Hu's level scheduling algorithm.

`HuParProcs* parSched;`
This is the protected member to point to the HuParProcs object. That object is created in the following method:

`void setUpProcs(int` *num*`);`
This method first calls `ParScheduler::setUpProcs` and next creates a HuParProcs

object. The HuParProcs is deallocated in the destructor.

`int scheduleIt();`
The scheduling procedure is exactly same as that of the Dynamic Level Scheduler except that the actual scheduling routines are provided by a HuParProcs object rather than a DLParProcs object. Refer to the `scheduleIt` method of class DLScheduler . Also note that the runnable nodes in this scheduling algorithm are sorted by their available time first.

`StringList displaySchedule();`
Displays the scheduling result textually.

15.10.4 Class HuParProcs

Class HuParProcs is derived from class DLParProcs so that it has the same constructor. While many scheduling methods defined in the DLParProcs class are inherited, some virtual methods are redefined to realize different scheduling decisions. For example, it does not consider the communication overhead to determine the processor that can schedule a node earliest. And it does not schedule communication resources. Another big difference is that Hu's level scheduling algorithm has a notion of global time clock. No node can be scheduled ahead of the global time. At each scheduling step, the global time is the same as the available time of the node at the head of the list of runnable nodes.

`void fireNode(DLNode* n);`
This redefined protected method sets the available time and index of the preferred processor of the descendants, if they are runnable after node *n* is completed. This is done before putting them into the list of runnable nodes (`sortedInsert` method of the HuGraph class).

`void scheduleSmall(DLNode* n);`
When this method is called, the node *n* is one of the earliest runnable nodes. We examine a processor that could schedule the node at the same time as the available time of the node. If the node is at the wormhole boundary, we examine the first processor only. If the node should be assigned to the same processor on which any earlier invocations were already assigned, we examine that processor whether it can schedule the node at that time or not. If no processor is found, we increase the available time of the node to the earliest time when any processor can schedule it, and put the node back into the list of runnable nodes. If we find a processor to schedule the node at the available time of the node, we assign and fire the node, then update the variables of the HuGraph. Recall that no communication overhead is considered.

15.11 Declustering Scheduler

Declustering scheduler is the most elaborate scheduler developed by Sih . This algorithm only applies to the homogeneous multiprocessor targets and it does not support wormholes nor parallel stars. Since it takes into account the global information of the graph, it may overcome the weaknesses of list schedulers which consider only local information at each scheduling step. It turns out that this scheduling algorithm is very costly since it involves recursive executions of the list scheduler with various assignment, and choosing the best scheduling result. The sched-

uling routine was originally written in LISP for the Gabriel system and was translated into C++. Since the algorithm itself is very complicated, the reader of the code is highly encouraged to read Sih's paper on the scheduling algorithm.

Class DeclustScheduler is derived from class ParScheduler. It has a constructor with two arguments as the ParScheduler class. The subclass of ParProcessors used in DeclustScheduler is DCParProcs. The DeclustScheduler maintains two kinds of DCParProcs instances, one to save the best scheduling result so far, and the other is for retrying list scheduling whose results will be compared with the best result so far. These two DCParProcs are created in

`void setUpProcs(int num);`

They are deleted in the destructor.

`StringList displaySchedule();`

Displays the best scheduling result obtained so far.

The overall procedure of the declustering algorithm is:

(1) Make elementary clusters of nodes. To make elementary clusters, we examine the output arcs of all branch nodes (a branch node is a node with more than one output arc) and the input arcs of all merge nodes (a merge node is a node with more than one input arc). Those arcs are candidates to be cut to make clusters. An arc is cut if the introduced communication overhead can be compensated by exploiting parallelism.

(2) We make a hierarchy of clusters starting from elementary clusters up to one cluster which includes all nodes in the APEG. Clusters with the smallest work-load will be placed at the bottom level of the hierarchy.

The above two steps are performed in the following protected method:

`int preSchedule();`

Before making clusters, it first checks whether the APEG has wormholes or parallel stars. If it finds any, it returns `FALSE`.

(3) We decompose the cluster hierarchy. We examine the hierarchy from the top. We assign two processors to each branch (son cluster) of the top node at the next level. Then, we execute list scheduling and save the scheduling result. We choose a son cluster with a larger work-load. Then, we introduce another processor to schedule two branches of the son cluster. Execute a different list scheduling to compare the previously best scheduling result, then save the better result. Repeat this procedure until all processors are consumed. It is likely to stop traversing the cluster hierarchy since all processors are consumed. At this stage, we compare the loads of processors and try to balance the loads within a certain ratio by shifting some elementary clusters from the most heavily loaded processor onto a lightly loaded processor.

(4) In some cases, we can not achieve our load-balancing goals by shifting clusters. We try to breakdown some elementary clusters in heavily loaded processors to lightly loaded clusters. This is cluster "breakdown".

(5) In each step of (3) and (4), we execute list scheduling to compare the previously best scheduling result. This is the reason why the declustering algorithm is computationally expensive. Finally, we get the best scheduling result. Based on that scheduling result, we make a final version of the APEG including all communication nodes (in the `finalizeGalaxy`

method of DCParProcs class). Note that we do not call the list scheduling algorithm in the ParProcessors class after we find out the best scheduling result since we already executed that routine for that result.

Steps (3), (4), and (5) are performed in the following public method:

```
int scheduleIt();
```

Many details of the scheduling procedure are hidden with private methods. The remaining section will describe the classes used for Declustering scheduling one by one.

15.11.1 Class DCNode

Class DCNode, derived from class ParNode, is an APEG node for the declustering scheduler. It has the same constructors with the ParNode class. This class does not have any protected members.

```
int amIMerge();
int amIBranch();
```

These methods return `TRUE` if this node is a merge node or a branch node.

```
DCCluster* cluster;
DCCluster* elemDCCluster;
```

These pointers point to the highest-level cluster and the current elementary cluster owning this node.

```
void saveInfo();
int getBestStart();
int getBestFinish();
```

The first method saves the scheduling information of this node with the best scheduling result, which includes the processor assignment, the scheduled time, and the completion time. The next two methods return the scheduled time and the completion time of the current node.

`int getSamples(DCNode*` ***`destN`***`);`

Returns the number of samples transferred from the current node to the destination node ***`destN`***`.` If no sample is passed, it returns 0 with an error message.

`DCNode* adjacentNode(DCNodeList&` ***`nlist`***`,` int ***`direction`***`);`

DCNodeList is derived from class EGNodeList just to perform type casting. This method returns an adjacent node of the current node in the given node list. If ***`direction`*** is 1, look at the ancestors, if -1, look at the descendants.

```
StringList print();
```

Prints the master star name and the invocation number of the node.

There are three iterators defined for DCNode class: `DCNodeListIter, DCAncestorIter,` and `DCDescendantIter.` As names suggest, they return the DCNode in the list, in the ancestors of a node, and in the descendants of a node.

15.11.2 Classes DCArc and DCArcList

Class DCArc represents an candidate arc in the APEG to be cut in making elementary clusters. It has a constructor with five arguments.

`DCArc(DCNode* src, DCNode* sink, int first, int second,int third);`

The first two arguments indicate the source and destination nodes of the arc. The remaining three arguments define a triplet of information used to help find the arcs to be cut. We call these arcs "cut-arcs". We consider a pair of a branch node and a merge node and two paths between them to determine if a pair of cut-arcs to parallelize these two paths is beneficial. The *first* argument is the sum of execution times of nodes preceding this arc, starting from the branch node. The *second* argument is the communication overhead for this arc. The *third* argument is the sum of execution times of nodes following from this arc to the merge node.

The five arguments given to the constructor can be retrieved by the following methods:

```
DCNode* getSrc();
DCNode* getSink();
int getF();
int getS();
int getT();
```

They can be printed by

```
StringList print();
```

The sink and the source nodes can be reversed, and can be copied from an argument DCArc by the following methods:

```
void reverse();
int operator==(DCArc& arc);
```

There are other public methods as follow:

```
DCArcList* parentList();
```

A DCArc will be inserted to a list of DCArcs, call DCArcList. This method returns the pointer to the list structure.

```
int betweenSameStarInvoc();
```

Returns TRUE or FALSE, based on whether this arc is between invocations of the same star.

Class DCArcList is derived from class SequentialList to make a list of DCArcs. It has a constructor with no argument and a copy constructor. The destructor deletes all DCArcs in the list.

```
void insert(DCArc* arc);
void append(DCArc* arc);
```

These methods to put *arc* at the front and the back of the list, respectively.

```
DCArc* head();
```

Returns the DCArc at the front of the list.

```
int remove(DCArc* arc);
```

Removes *arc* from the list.

`int member(DCArc* arc);`
Returns `TRUE` if the given DCArc is a member of the list.

`int mySize();`
Returns the number of DCArcs in the list.

`StringList print();`
It prints a list of DCArcs in the list.

There is a iterator for DCArcList, called DCArcIter, which returns a DCArc.

15.11.3 Class DCGraph

Class DCGraph, derived from class ParGraph, is an input APEG for DeclustScheduler. It has no explicit constructor.

`EGNode* newNode(DataFlowStar*, int);`
Creates a DCNode as an APEG node for DCGraph.

```
DCNodeList BranchNodes;
DCNodeList MergeNodes;
```
These lists store the branch nodes and merge nodes.

`int initializeGraph();`
This protected method initializes the DCGraph. It sets up the lists of branch nodes and merge nodes (`BranchNodes, MergeNodes),` and the list of initially runnable nodes. We sort these lists by the static levels of the nodes: the branch nodes are sorted by smallest static level first while the merge nodes are sorted by largest static level first. In this method, we also initialize the DCNodes, which includes the detection of the merge nodes that are reachable from the node and the branch nodes reachable to the node.

The remaining methods are all public.
`const char* genDCClustName(int type);`
Generate a name for the cluster. If *type* = 0, we prefix with "ElemDCClust" to represent an elementary cluster. Otherwise, we prefix with "MacroDCClust".

`StringList display();`
Displays the APEG with the lists of initially runnable nodes, the branch nodes, and the merge nodes.

`DCNode* intersectNode(DCNode* d1, DCNode* d2, int direction);`
This method returns a merge node with the smallest static level, reachable from both *d1* and *d2* if *direction* = 1. If *direction* = 0, it returns a branch node with the smallest static level, that can reach both *d1* and *d2* nodes.

`DCArcList* traceArcPath(DCNode* branch, DCNode* src, DCNode* dest, int direction);`

This method makes a list of candidate cut-arcs between *branch* and *dest* nodes, and returns the pointer to the list. The second argument, *src,* is an immediate descendant of the *branch* node on the path to the *dest* node. If *direction* = 1, we reverse all arcs and find cut-arcs from *dest* to *branch* nodes.

`void addCutArc(DCArc*` *arc*);
This method adds a DCArc to a list of cut-arcs in DCGraph.

`void formElemDCClusters(DCClusterList&` *EClusts*);
In this method, we remove all cut-arcs in the APEG and make each connected component an elementary cluster. The argument *EClusts* is the list of those elementary clusters. We connect these clusters at the end.

`void computeScore();`
In scheduling stage (3) of the DeclustScheduler, we may want to shift clusters from heavily loaded processors to lightly loaded processors. To prepare this step, we compute the score of top-level clusters in that scheduling phase. The score of a cluster is the number of samples passed to other processors minus the number of samples passed inside the same processor along the cut-arcs within that cluster. The score indicates the cost of shifting a cluster due to communication.

`void commProcs(DCCluster*` *clust*, int* *procs*);
This method finds processors that *clust* communicates with. We set the component of the second argument array to 1 if that processor communicates with the cluster.

`void copyInfo();`
Used for saving the scheduling information if the most recent scheduling result is better than the previous ones.

15.11.4 Class DCCluster

Class DCCluster represents a cluster of nodes in the declustering algorithm. There is no protected member in this class. It consists of two DCClusters, called component clusters, to make a hierarchy of clusters. An elementary cluster has `NULL` component clusters. It is constructed by a one-argument constructor.

`DCCluster(DCNodeList*` *node-list*);
Makes the cluster contain all nodes from the list.

To make a macro cluster, we use the following constructor:
`DCCluster(DCCluster*` *clust1*, `DCCluster*` *clust2*);
The argument clusters become the component clusters of this higher level cluster. The cluster-arcs are established from the cluster-arcs of two component clusters by calling the following method:

`void fixArcs(DCCluster*` *clust1*, `DCCluster*` *clust2*);
In this method, arcs put inside this cluster are removed from the arcs of two argument clusters.

In both constructors, we compute the sum of execution times of all nodes in the cluster, which can be obtained by

```
int getExecTime();
DCCluster* getComp1();
DCCluster* getComp2();
```

The last two methods above return two component clusters.

```
void setName(const char* name);
const char* readName();
```

The above methods set and get the name of the cluster.

```
void addArc(DCCluster* adj, int numSample);
```

This method adds a cluster-arc that is adjacent to the first argument cluster with sample rate *numSample.*

```
void setDCCluster(DCCluster* clust);
```

Sets the `cluster` pointer of the nodes in this cluster to the argument cluster.

```
void assignP(int procNum);
int getProc();
```

The first method assigns all nodes in the cluster to a processor. The second returns the processor that this cluster is assigned to.

```
void switchWith(DCCluster* clust);
```

Switches the processor assignment of this cluster with the argument cluster.

```
DCCluster* pullWhich();
```

Returns the cluster with the smaller execution time between two component clusters, and pull it out.

```
DCCluster* findCombiner();
```

This method returns the best cluster, in terms of cluster-arc communication cost, to be combined. We break ties by returning the cluster with smallest execution time.

```
void broken();
int getIntact();
```

These methods indicate whether the cluster or its subclusters were broken into its components in the scheduling stage (3) of DeclustScheduler. The first method indicates that it happens. The second method queries whether it happens or not.

```
int getScore();
int setScore(int score);
void resetMember();
```

These methods get and set the score of the cluster. Refer to the `computeScore` method of the DCGraph class to see what the score of a cluster is. The last method resets the score to 0.

```
StringList print();
```

Prints the name of this cluster and the names of component clusters.

`~DCCluster();`
The destructor deletes the nodes in the cluster and cluster-arcs if it is an elementary cluster.

15.11.5 Class DCClusterList

Class DCClusterList, derived from class DoubleLinkList, keeps a list of clusters. It has no protected members. It has a default constructor and a copy constructor.

`void insert(DCCluster* clust);`
`void append(DCCluster* clust);`
`void insertSorted(DCCluster* clust);`
These methods put *clust* at the head and the back of the list. The last method inserts the cluster in order of increasing execution time.

```
DCClusterLink* firstLink();
DCCluster* firstDCClust();
DCCluster* popHead();
```
The above methods return the DCClusterLink and DCCluster at the head of the list. The last method removes and returns the cluster from the list. Class DCClusterLink is derived from class DoubleLink as a container of DCCluster in the DCClusterList. It has a public method to access the cluster called

```
DCCluster* getDCClustp();
DCClusterLink* createLink(DCCluster* clust);
void removeDCClusters();
```
Create a DCClusterLink and removes all clusters in the list, respectively.

```
void resetList();
void resetScore();
void setDCClusters();
```
The first two methods reset the scores of all clusters to 0. The first method also declares that each cluster is not broken. The third method resets the cluster pointer of the nodes of the clusters in the list.

`int member(DCCluster* clust);`
Returns `TRUE` or `FALSE`, based on whether the argument cluster is in the list or not.

`void findDCClusts(DCNodeList& nlist);`
Add to the list clusters that own the nodes of the argument list. If the number of clusters is 1, we break the cluster into two component clusters and put them into the list.

`int listSize();`
Returns the number of clusters in the list.

`StringList print();`
Prints the list of clusters.

There is an iterator associated with the DCClusterList called DCClusterListIter that returns a DCCluster. It can return a DCClusterLink by the `nextLink` method.

15.11.6 Class DCClustArc and class DCClustArcList

Class DCClustArc represents a cluster-arc. It has a constructor with two arguments:

`DCClustArc(DCCluster* neighbor, int nsamples);`
The first argument is the pointer to the neighboring cluster while the second argument sets the sample rate of the connection.

```
DCCluster* getNeighbor();
void changeNeighbor(DCCluster* clust);
```
These methods return the neighbor cluster and change to it.

```
void changeSamples(int newsamps);
void addSamples(int delta);
int getSamples();
```
The above methods modify, increment, and return the sample rate of the current arc.

```
StringList print();
```
Prints the name of the neighbor cluster and the sample rate.

Class DCClustArcList is derived from class SequentialList to make a list of cluster-arcs. It has four public methods.

`DCClustArc* contain(DCCluster* clust);`
Returns the DCClustArc that is adjacent to the argument cluster. If no cluster-arc is found in the list, return 0.

`void changeArc(DCCluster* oldC, DCCluster* newC);`
This method changes the pointer of neighbor cluster, *oldC,* in all cluster-arcs in the list to *newC.*

```
void removeArcs();
```
Deletes all cluster-arcs in the list.

```
StringList print();
```
Prints the list of DCClustArcs.

There is an iterator associated with the DCClustArcList, called DCClustArcListIter, which returns a DCClustArc.

15.11.7 Class DCParProcs

Class DCParProcs is derived from class ParProcessors. It has the same constructor and destructor with the ParProcessors class.

There is one protected method:

`ParNode* createCommNode(int i);`
Creates a DCNode to represent a communication code. The argument indicates the type of the node.

The other methods are all public, and support the main scheduling procedure described

in the DeclustScheduler class .

`int commAmount();`

Returns the communication overhead of the current schedule.

`void saveBestResult(DCGraph* graph);`

This method saves the current scheduling information of the nodes as the best scheduling result.

`void finalizeGalaxy(DCGraph* graph);`

After all scheduling is completed, we make a final version of the APEG including all communication loads based on the best scheduling result obtained.

`void categorizeLoads(int procs);`

This method categorizes each processor as either heavily or lightly loaded. It sets an integer array, *nprocs,* 1 for heavy and -1 for light processors. The initial threshold is 50 processors are heavily loaded if all processors are loaded beyond a 75 maximum load. We regard at most one idle processor as lightly loaded.

`int findSLP(DCNodeList* nlist);`

This method finds the progression of nodes (regular or communication) in the schedule which prevents the makespan from being any shorter. We call this set of nodes and *schedule limiting progression: SLP* (refer to Sih's paper). The SLP can span several processors and can't contain idle times. If there are several SLPs it will return just one of them.