Getting started with pRPL - Appendix

Qingfeng (Gene) Guan

guanqf {at} gmail.com

Department of Geography

University of California, Santa Barbara

Santa Barbara, CA 93106, USA

April, 2008

Contents

TABLE 2.1 ATTRIBUTES OF PRPL::PRPROCESS *3*

TABLE 2.2 METHODS OF PRPL::PRPROCESS *4*

TABLE 2.3 ATTRIBUTES OF PRPL::NEIGHBORHOOD *9*

TABLE 2.4 METHODS OF PPRL::NEIGHBORHOOD *10*

TABLE 2.5 ATTRIBUTES OF PRPL::CELLSPACE *15*

TABLE 2.6 METHODS OF PRPL::CELLSPACE *16*

TABLE 2.7 ATTRIBUTES OF PRPL::SUBSPACEINFO *22*

TABLE 2.8 METHODS OF PRPL::SUBSPACEINFO *23*

TABLE 2.9 ATTRIBUTES OF PRPL::SUBSPACE *28*

TABLE 2.10 METHODS OF PRPL::SUBSPACE *29*

TABLE 2.11 ATTRIBUTES OF PRPL::LAYER *33*

TABLE 2.12 METHODS OF PRPL::LAYER *34*

TABLE 2.13 ATTRIBUTES OF PRPL::TRANSITION *40*

TABLE 2.14 METHODS OF PRPL::TRANSITION *41*

Table 2.1 Attributes of pRPL::PRProcess

Name	Type	Description
_id	int	The process's ID[1]
_grpID	int	The ID of the group in which the process is included
_nPrcs	int	The number of processes in the group
_masterID	int	The ID of the group's master process
_comm	MPI_Comm	The communicator of the group
_vSubInfos	vector<SubSpaceInfo> [2]	The vector of all the subspaces' information
_pPrcInfoMap	SubSpaceMap* [3]	The processID-indexed map of vectors of subspace information
_pSubIDMap	SubIDMap* [4]	The vector of subspaceID-processID pairs

[1] Here a process is the representation of a hardware processor
[2] For details of SubSpaceInfo, see table 2.8 and table 2.9
[3] For more information about SubSpaceMap (Figure 2.2a), see subSpaceMap.h and subSpaceMap.cpp
[4] For more information about SubIDMap (Figure 2.2b), see subIDMap.h and subIDMap.cpp

Table 2.2 Methods of pRPL::PRProcess

	Name	Description
Constructor & Destructor	**PRProcess**();	Creates an instance of PRProcess, with no communicator associated.
	PRProcess(MPI_Comm *comm*, int *groupID* = -1); IN [5] *comm* The communicator of the group OPT_IN *groupID* The ID of the group the PRProcess belongs to	Creates an instance of PRProcess, with a communicator associated, and belonging to a group. When groupID is -1, the PRProcess belongs to the root group
	~PRProcess();	Destroys a PRProcess object
Process Managing	bool **initialized**() const;	Returns true if MPI has been activated. No MPI processes can be done if it has not been initialized.
	bool **active**() const;	Returns true if a PRProcess object is active, i.e., having a valid communicator and a valid ID.
	bool **init**(int *argc* = 0, char[]* *argv* = 0); OPT_IN *argc* The number of arguments OPT_IN *argv* The values of arguments	Initializes a PRProcess object, i.e., activating MPI if it has not been initialized, determining its ID and the number of processes in the communication context (or group). The arguments, argc and argv, are the arguments for the main() function. Returns true if the initialization worked successfully.
	void **abort**() const;	Aborts the jobs on all the PRProcess objects within a group.
	void **finalize**() const;	Finalizes MPI. After finalized, no MPI functions can be called. Usually used right before the end of the program.

[5] IN – Input; OPT_IN – Optional input; OUT – Output; OPT_OUT – Optional Output

	void **sync**() const;	Synchronizes all the PRProcess objects within a group by blocking them from continuing with new tasks until all the processes reach this point.
	bool **grouping**(int *nGroups*, bool *incldMaster*, PRProcess* *pGrpedPrc* PRProcess* *pGrpMaster* = 0) const; IN *nGroups* The number of child groups IN *incldMaster* Whether include the master process of the parent group in the child groups OUT *pGrpedPrc* The pointer to the PRProcess initialized to represent the hardware processor in the new group OPT_OUT *pGrpMaster* If pGrpMaster is given when this method is called, a special group will be created and it only includes the master processes of all the child groups	Divides a parent group (within which **this* is located) into *nGroups* child groups. Returns true if the division completed successfully.
	const MPI_Comm& **comm**() const;	Returns the communicator of the PRProcess object
	int **id**() const;	Returns the ID of the PRProcess object
	int **groupID**() const;	Returns the ID of the group within which the PRProcess object is located.
	int **nPrcs**() const;	Returns the number of processes in the group
	bool **isMaster**() const;	Returns true if the PRProcess is the master of the group in which it is located
	int **masterID**() const;	Returns the ID of the master process of the group

Sub-cellspaces Management	void **clearSubInfos**()	Clears the information sets of sub-cellspaces
	void **clearPrcInfoMap**();	Clears the map linking processes and sub-cellspaces
	void **clearSubIDMap**();	Clears the map linking sub-cellspaces and processes
	bool **hasPrcInfoMap**() const;	Return true if the map linking processes and sub-cellspaces exists, i.e., the sub-cellspaces have been mapped onto processes.
	bool **hasSubIDMap**() const;	Return true if the map linking sub-cellspaces and processes exists, i.e., the sub-cellspaces have been mapped onto processes
	bool **bcastSubInfos**();	Broadcasts the information sets of sub-cellspaces to all the servant processes from the master process. This method requires the existence of the information sets of sub-cellspaces on the master process, i.e., the master process has established the cellspace divisions. Returns true if the broadcasting completed successfully.
	bool **mapping**();	Maps the sub-cellspaces onto processes. This method requires that bcastSubInfos has been done, i.e., all the processes have the information sets of sub-cellspaces. Since the mapping scheme is identical for all the processes, the process-to-sub-cellspace maps will be the same for all of them. Returns true if the mapping completed successfully.
	int **nTotSubSpcs**() const;	Returns the total number of sub-cellspaces.
	int **nLclSubSpcs**() const;	Returns the number of sub-cellspaces owned by the PRProcess object.
	const vector<SubSpaceInfo*>& **lclSubInfos**() const;	Returns an information set (in a vector form) of sub-cellspaces that are owned by the PRProcess object.

Communication Tools	bool **bcastStr**(string& *str*) const; IN/OUT *str* The string to be broadcast from the master	Broadcasts a string (i.e., str) to the servant processes from the master process. Returns true if the broadcasting completed successfully.
	template <class elemType> void **bcastVal**(elemType& *val*) const; IN/OUT *val* The variable to be broadcast from the master	Method template. Broadcasts a variable (i.e., val) of any type (user-defined) to the servant processes from the master process.
	template <class elemType> void **bcastVct**(vector<elemType>& *vect*) const; IN/OUT *vect* The vector to be broadcast from the master	Method template. Broadcasts a vector (i.e., vect) of any type (user-defined) to the servant processes from the master process.
	template <class elemType> void **gatherVal**(elemType& *val*, vector<elemType>& *vect*) const; IN *val* The variable on each process to be gathered OUT *vect* The vector on the master for storing the values gathered from the processes	Method template. Gathers a variable (i.e., val) of any type (user-defined) from all the processes to the master process, and saves them (including the variable on the master process) in a vector (i.e., vect) on the master process.
	template <class elemType> void **gatherVct**(vector<elemType>& *vSend*, vector<elemType>& *vRecv*) const; IN *vSend* The vector on each process to be gathered OUT *vRecv* The vector on the master for storing the vectors gathered from the processes	Method template. Gathers a vector (i.e., vSend) of any type (user-defined) from all the processes to the master process, and saves them (including the vector on the master process) in a vector (i.e., vRecv) on the master process.

	template <class elemType> void **allGatherVal**(elemType& *val*, vector<elemType>& *vect*) const; IN *val* The variable on each process to be gathered OUT *vect* The vector on each process for storing the values gathered from the processes	Method template. Does the same thing as gatherVal(), but instead of saving the resultant vector only on the master process, it saves a vector (i.e., vect) on every process.
	template <class elemType> void **allGatherVct**(vector<elemType>& *vSend*, vector<elemType>& *vRecv*) const; IN *vSend* The vector on each process to be gathered OUT *vRecv* The vector on each process for storing the vectors gathered from the processes	Method template. Does the same thing as gatherVct(), but instead of saving the resultant vector only on the master process, it saves a vector (i.e., vRecv) on every process.

Table 2.3 Attributes of pRPL::Neighborhood

Name	Type	Description
_vNbrs	vector< WeightedCell<elemType> > [6] [7]	The vector of neighbors, i.e., weighted cells. It's a template member, and capable of accommodating any type of values
_MBR	CoordBR [8]	The minimum bounding rectangle of the neighborhood (Figure 2.6)
_mNbrIDMap	vector< vector<int> >	The neighboring-cell-ID map (Figure 2.7)
_specified	bool	True if the Neighborhood object is already specified to a coordinate in a cellspace

[6] The WeightedCell class stores the coordinate, weight, and value of a cell in a cellspace. For more information about WeightedCell, see basicCell.h and weightedCell.h

[7] elemType may be any type, e.g., int, long, double, even an user-defined type. pRPL::Neighborhood is a class template. When declaring a Neighborhood object, a data type must be given. For example, *Neighborhood<int> myNbr* declares a Neighborhood object, i.e., *myNbr*, of integer values.

[8] The CoordBR class stores the northwest (upper-left) and southeast (bottom-right) corners' coordinates of a rectangle in a cellspace. For more information about CoordBR, see basicTypes.h

Table 2.4 Methods of pPRL::Neighborhood

	Name	Description
Constructors & Destructor	**Neighborhood**();	Creates an instance of Neighborhood with no neighbor cells
	Neighborhood(const vector<CellCoord>&[9] *vNbrCoords*, double *weight* = 1.0); IN *vNbrCoords* The vector of the neighbor coordinates OPT_IN *weight* The weight associated with the neighbors	Creates an instance of Neighborhood based on a vector of neighbor coordinates. All the neighbors will have the same weight (which is set to *1.0* by default). The order of the coordinates also determines the indices of the neighbors (from *0* to *n-1*, where n is the size of the vector).
	Neighborhood(const vector<CellCoord>& *vNbrCoords*, const vector<double>& *vNbrWeights*); IN *vNbrCoords* The vector of the neighbor coordinates IN *vNbrWeights* The vector of the neighbor weights	Creates an instance of Neighborhood based on a vector of neighbor coordinates, and a vector of the corresponding neighbor weights. NOTICE: the sizes of these two vectors must be the same.
	Neighborhood(const Neighborhood<elemType>& *rhs*); IN *rhs* The Neighborhood object to copy from	A copy constructor. Create an instance of Neighborhood that is a copy of the given Neighborhood object, i.e., *rhs*.

[9] The CellCoord class stores the row-column indices of a cell as the location in a cellspace. For more information about CellCoord, see basicType.h

	Template<class elemType2>
	Neighborhood(const Neighborhood<elemType2>& *rhs*);
	
	IN *rhs* The original Neighborhood object to construct a new object based on
		Method template. Create an instance of Neighborhood based on another Neighborhood object, i.e., rhs. The value type of the newly created Neighborhood does not have to be the same as that of the original, but the spatial configurations and weighting schemes will be the same.
		For example, the following code creates an integer-type Neighborhood based on a double-type Neighborhood.
		`Neighborhood<double> nbr1(vCoords, vWeights);`
		`Neighborhood<int> nbr2(nbr1);`

	~Neighborhood();	Destroys an instance of Neighborhood.
Operators	const WeightedCell<elemType>& **operator[]** (int *iNbr*) const; WeightedCell<elemType>& **operator[]** (int *iNbr*); IN *iNbr* The index of the neighbor cell	Returns an individual neighbor (i.e., weighted cell) within the Neighborhood.
	Neighborhood<elemType>& **operator=**(const Neighborhood<elemType>& *rhs*); IN *rhs* The Neighborhood object to copy from	Assignment operator. Copies the attribute values from *rhs* to **this*.
Management tools	bool **empty**() const;	Returns true if the Neighborhood object has no neighbors
	int **size**() const;	Returns the number of neighbors within the neighborhood.
	bool **isEquallyWeighted**(double& *weight*) const; OUT *weight* The weight value	Returns true and *weight* will be the weight value if the neighbors are identically weighted.
	void **clear**();	Deletes all the neighbors in the neighborhood.

<table>
<tr><td></td><td></td><td></td></tr>
<tr><td></td><td>const CoordBR& getMBR() const;</td><td>Returns the minimum bounding rectangle of the Neighborhood object.</td></tr>
<tr><td></td><td>int nRows() const;</td><td>Returns the number of rows of the MBR</td></tr>
<tr><td></td><td>int nCols() const;</td><td>Returns the number of columns of the MBR</td></tr>
<tr><td></td><td>bool hasNbrs(MeshDir dir) const;

IN dir The direction to the central cell</td><td>Returns true if there are neighbors located in the dir direction to the central cell.
MeshDir is an enumeration type for directions (see basicType.h)
<table>
<tr><td>Name</td><td>Value</td></tr>
<tr><td>NORTH_DIR</td><td>0</td></tr>
<tr><td>NORTHEAST_DIR</td><td>1</td></tr>
<tr><td>EAST_DIR</td><td>2</td></tr>
<tr><td>SOUTHEAST_DIR</td><td>3</td></tr>
<tr><td>SOUTH_DIR</td><td>4</td></tr>
<tr><td>SOUTHWEST_DIR</td><td>5</td></tr>
<tr><td>WEST_DIR</td><td>6</td></tr>
<tr><td>NORTHWEST_DIR</td><td>7</td></tr>
</table></td></tr>
<tr><td></td><td>const vector<int>* nbrIDs(MeshDir dir) const;

IN dir The direction to the central cell</td><td>Returns a vector of the indices of the neighbors located in the dir direction to the central cell.</td></tr>
<tr><td colspan="3"></td></tr>
<tr><td>Specifying tools</td><td>bool specify(int iRow,
int iCol,
const CellSpace<elemType>& cellspace);</td><td>Specifies the Neighborhood object to a coordinate in the cellspace. Returns true if the specifying completed successfully.</td></tr>
</table>

	bool **specify**(const CellCoord& *coord*, const CellSpace<elemType>& *cellspace*); IN *coord* The coordinate to which the Neighborhood is specified IN *cellspace* The cellspace in which the Neighborhood is specified	Specifies the Neighborhood object to a coordinate in the cellspace. Returns true if the specifying completed successfully.
	bool **specify**(const BasicCell<elemType>&[10] *cell*, const CellSpace<elemType>& *cellspace*); IN cell The cell to which the Neighborhood is specified IN cellspace The cellspace in which the Neighborhood is specified	Specifies the Neighborhood object to a cell in the cellspace. Returns true if the specifying completed successfully.
	bool **totalVal**(elemType& *value*, bool *includingCtr* = true) const; OUT *value* The summary of the neighbors' weighted values OPT_IN *includingCtr* Whether include the central cell's value	Calculates the summary of the neighbors' weighted values. Returns true if the calculation completed successfully. NOTICE, the Neighborhood object must be specified in a cellspace, and the values of the cells covered by the Neighborhood have been copied to the corresponding neighbor cells. $$value = \sum_{i=0}^{n-1} (weight_i \times value_i)$$

[10] For more information about BasicCell, see basicCell.h

	template<class Predicate>
	int **count**(Predicate *pred*,
	bool *includingCtr* = true) const;
	
	IN *pred* The evaluation operator applied on the neighbors
	OPT_IN *includingCtr* Whether include the central cell's value
		Applied the *pred* evaluation on all the neighbors in the neighborhood, and returns the counts of the cases when *pred* returns true. NOTE that the Neighborhood object must be specified in a cellspace, and the values of the cells covered by the Neighborhood have been copied to the corresponding neighbor cells.
		For example:
		`Neighborhood<int> nbr(vCoords);`
		`nbr.specify(coord, cellspace);`
		`int nLessThanTen = nbr.count(bind2nd(less<int>(), 10), true); /* count the neighbors whose values are less than 10 */`

Table 2.5 Attributes of pRPL::CellSpace

Name	Type	Description
_dims	SpaceDims [11]	The spatial dimensions of the cellspace, i.e., the numbers of rows and columns of cells
_matrix	elemType* [12]	The array of cell values. It is a template member, and capable of accommodating any type of data
_mUpdtCells	map<elemType, vector<int> >	The updated-cell map (Figure 2.10)

[11] The SpaceDims class stores the numbers of rows and columns of a cellspace. For more information about SpaceDims, see basicTypes.h

[12] pRPL::CellSpace is a class template. The data type must be given when declaring a CellSpace object. For example, *CellSpace<double> myCellSpc* declares a CellSpace object, i.e., *myCellSpc*, of double values, which means the cell values stored in *_matrix* are double precession floating point numbers.

Table 2.6 Methods of pRPL::CellSpace

	Name	Description
Constructors & Destructor	**CellSpace**();	Creates an empty instance of CellSpace
	CellSpace(const SpaceDims& *dims*); IN *dims* The dimensions of the cellspace	Creates an instance of CellSpace of the dimensions of *dims*
	CellSpace(const SpaceDims& *dims*, const elemType& *initVal*); IN *dims* The dimensions of the cellspace IN *initVal* The initial value of the cellspace	Creates an instance of CellSpace of the dimensions of *dims*, and initializes the cell values as *initVal*
	CellSpace(int *nRows*, int *nCols*); IN *nRows* The number of rows of the cellspace IN *nCols* The number of columns of the cellspace	Creates an instance of CellSpace of *nRows* rows and *nCols* columns
	CellSpace(int *nRows*, int *nCols*, const elemType& *initVal*); IN *nRows* The number of rows of the cellspace IN *nCols* The number of columns of the cellspace IN *initVal* The initial value of the cellspace	Creates an instance of CellSpace of *nRows* rows and *nCols* columns, and initializes the cell values as *initVal*
	CellSpace(const CellSpace<elemType>& *rhs*); IN *rhs* The CellSpace object to copy from	A copy constructor. Create an instance of CellSpace that is a copy of the given CellSpace object, i.e., *rhs*.

	~CellSpace();	Destroys a CellSpace object
Initializing tools	bool initMem(const SpaceDims& *dims*) IN *dims* The dimensions of the cellspace	Allocates a memory space for storing the cell values of the dimensions *dims.* Returns true if the memory is successfully allocated.
	bool initVals(const elemType& *initVal*); IN *initVal* The initial value of the cellspace	Initializes the cell values as *initVal*. NOTE: this method can only be called after the memory is already allocated, i.e., initMem is called. Returns true if the initialization finishes successfully.
	void clear();	Releases the memory space of the CellSpace object.
Operators	const elemType* **operator[]**(int *iRow*) const; elemType* **operator[]**(int *iRow*); IN *iRow* The row index in the cellspace	Returns a pointer pointing to the first element of a row indexed by *iRow*. This operator is especially useful when locating a cell using a row-column coordinate pair. For example: *myCellSpc[10][15]* returns the value of the cell located at row 10 and column 15. NOTE: the indices are 0-based.
	CellSpace<elemType>& **operator=**(const CellSpace<elemType> &*rhs*); IN *rhs* The CellSpace object to copy from	Assignment operator. Copies the attribute values from *rhs* to **this*, including the cell values.

Dimension Information	template <class elemType2>
	bool **equalDim**(const CellSpace<elemType2>& *rhs*) const;
	
	IN *rhs* The CellSpace object to compare dimensions with
		Returns true if the dimensions of *this CellSpace is the same as that of rhs. NOTE: the data types of these two CellSpace objects may be different. For example:
		`CellSpace<int> myCellSpc1;`
		`CellSpace<double> myCellSpc2;`
		`bool haveSameDims = myCellSpc1.equalDim(myCellSpc2);`

	bool **empty**() const;	Returns true if the CellSpace has not been allocated with memory space for storing cell values.

	int **nRows**() const;	Returns the number of rows of the cellspace
	int **nCols**() const;	Returns the number of columns of the cellspace
	int **size**() const;	Returns the number of cells in the cellspace, i.e., the number of rows multiplies the number of columns
	const SpaceDims& **dims**() const;	Returns the spatial dimensions of the cellspace
	bool **validCoord**(const CellCoord& *coord*, bool *warning* = true) const; IN *coord* The row-column coordinate pair to be examined OPT_IN *warning* Whether prompt a warning message if the coordinate is invalid	Examines whether a coordinate is valid, i.e., it is within the cellspace's dimensions. If not, and the *warning* option is on, a warning message will be prompted
	bool **validCoord**(int *iRow*, int *iCol*, bool *warning* = true) const; IN *iRow* The row index of the coordinate to be examined IN *iCol* The column index of the coordinate to be examined OPT_IN *warning* Whether prompt a warning message if the coordinate is invalid	Examines whether a coordinate is valid, i.e., it is within the cellspace's dimensions. If not, and the *warning* option is on, a warning message will be prompted
	bool **validIdx**(int *idx*, bool *warning* = true) const; IN *idx* The array index to be examined OPT_IN *warning* Whether prompt a warning message if the coordinate is invalid	Examines whether an array index is valid, i.e., it is within the range of the value matrix. If not, and the *warning* option is on, a warning message will be prompted
	int **coord2idx**(const CellCoord& *coord*) const;	Returns the array index corresponding to the input row-column coordinate pair. (Function 2.3)

	IN *coord* The row-column coordinate	
	int **coord2idx**(int *iRow*, int *iCol*) const;	Returns the array index corresponding to the input row-column coordinate pair. (Function 2.3)
	const CellCoord **idx2coord**(int *idx*) const; IN idx The array index	Returns the row-column coordinate corresponding to the input array index. (Function 2.1 and 2.2)
Cell Attribute Information	bool **values**(vector<elemType>& *vVals*) const; OUT *vVals* The vector of cell values	Queries all distinct cell values store in the CellSpace object. Returns true if it completes successfully
	bool **find**(vector<int>& *vFoundIdxs*, const elemType& *val*) const; bool **find**(vector<int>& *vFoundIdxs*, const vector<elemType>& *vVals*) const; bool **find**(vector<int>& *vFoundIdxs*, const vector<elemType>& *vVals*, const CoordBR& *rectangle*) const; OUT *vFoundIdxs* The vector of the indices of the cells whose values match *val/vVals* IN *Val* The attribute value to find IN *vVals* The vector of attribute values to find IN *rectangle* The search range	Queries the cells with (a) certain attribute value(s), and the query may be constrained within a certain area (defined by *rectangle*) in the cellspace.
	template<class Predicate> int **count**(Predicate *pred*) const;	Applied the *pred* evaluation on all the cell values in the cellspace, and returns the counts of the cases when *pred* returns true. The query may be constrained within a certain area (defined by *rectangle*) in the cellspace.

	template<class Predicate> int **count**(Predicate *pred*, const CoordBR& *rectangle*) const; IN *pred* The evaluation operator applied on the cells IN *rectangle* The search range	For Example: `int nLessThanTen = myCellSpc.count(bind2nd(less<int>(), 10)); /* count the cells whose values are less than 10 */`
Updating Tools	bool **updateCells**(const vector< BasicCell<elemType> >& *vCells*); bool **updateCells**(const vector<pair<int, elemType> >& *vCells*); IN *vCells* The vector of cells to be updated IN *vCells* The vector of index-value pairs to be updated	Updates a set of cells in the cellspace. Returns true if the updating completes successfully. NOTE: the cell values will be updated immediately.
	bool **update**(Transition<elemType>* *pTransition*, Neighborhood<elemType>* *pNbrhood*, const CoordBR *const *pWorkBR* = 0); IN *pTransition* The transition process used to update the cellspace IN *pNbrhood* The neighborhood configuration used to update the cellspace OPT_IN *pWorkBR* The work range. If it is not a NULL pointer, i.e., not equal to zero, it defines a rectangle bounding an area where the transition will be applied on	Evaluates and updates the cellspace using a certain transition process and a neighborhood configuration (within a certain area if pWorkBR is not a NULL pointer). NOTE: if the *needFinalize* option of the Transition object is ON, the updated cells will be stored in the updated-cell map (Figure 2.10), i.e., *_mUpdtCells*, and the cell values will not be changed. Otherwise the values of the updated cells will be changed immediately. Returns true if the updating completes successfully.

	bool **update**(vector<int>& *vIdxs2Eval*, Transition<elemType>* *pTransition*, Neighborhood<elemType>* *pNbrhood*, const CoordBR *const *pWorkBR* = 0); IN *vIdxs2Eval* The vector of array indices of the cells to be evaluated IN *pTransition* The transition process used to update the cellspace IN *pNbrhood* The neighborhood configuration used to update the cellspace OPT_IN *pWorkBR* The work range. If it is not a NULL pointer, i.e., not equal to zero, it defines a rectangle bounding an area where the transition will be applied on	Evaluates and updates a set of cells (i.e., *vIdxs2Eval*) in the cellspace using a certain transition process and a neighborhood configuration (within a certain area if pWorkBR is not a NULL pointer). NOTE: if the *needFinalize* option of the Transition object is ON, the updated cells will be stored in the updated-cell map (Figure 2.10), i.e., *_mUpdtCells*, and the cell values will not be changed. Otherwise the values of the updated cells will be changed immediately. Returns true if the updating completes successfully.
	bool **updateFinalize**(Transition<elemType>* *pTransition* = 0, Neighborhood<elemType>* *pNbrhood* = 0); OPT_IN *pTransition* The transition process used to finalize the updating OPT_IN *pNbrhood* The neighborhood configuration used to finalize the updating	Finalizes an updating process, i.e., post-processing the updated cells stored in *_mUpdtCells*. If the transition is not given, i.e., *pTransition* is zero, then this method simply loads values to the updated cells without any additional processing. Returns true if the finalizing completes successfully. NOTE: this method should be called after the **update** method is called, i.e., after the updated-cell map is populated.
	void **cleanUpdtMap**();	Cleans the updated-cell map

Table 2.7 Attributes of pRPL::SubSpaceInfo

Name	Type	Description
_id	int	The ID of the sub-cellspace
_domDcmpType	DomDcmpType [13]	The decomposition type
_glbDims	SpaceDims	The dimensions of the global (whole) cellspace
_MBR	CoordBR	The minimum bounding rectangle of the sub-cellspace in the global dimensions. NOTE: it is in global coordinate system (Figure 2.12)
_dims	SpaceDims	The dimensions of sub-cellspace
_workBR	CoordBR	The work range of the sub-cellspace, i.e., the Edge cells and Interior Cells. NOTE: it is in local coordinate system (Figure 2.12)
_mNbrSpcID	vector< vector<int> >	The neighboring sub-cellspace map (Figure 2.13)

[13] DomDcmpType is an enumeration type (see basicTypes.h)

Name	Value	Description
NON_DCMP	0	No decomposition
ROWWISE_DCMP	1	Row-wise decomposition
COLWISE_DCMP	2	Column-wise decomposition
BLOCK_DCMP	3	Block-wise decomposition

Table 2.8 Methods of pRPL::SubSpaceInfo

	Name	Description
Constructors and Destructor	**SubSpaceInfo**();	Creates an empty instance of SubSpaceInfo
	SubSpaceInfo(int *id*, DomDcmpType *domDcmpType*, const SpaceDims& *glbDims*, const CoordBR& *MBR*, const CoordBR& *workBR*, const vector< vector<int> >& *mNbrSpcIDs*); IN *id* The ID of the sub-cellspace IN *domDcmpType* The decomposition type IN *glbDims* The dimensions of the global (whole) cellspace IN *MBR* The minimum bounding rectangle of the sub-cellspace in the global dimensions. IN *workBR* The work range of the sub-cellspace, i.e., the Edge cells and Interior Cells. IN *mNbrSpcIDs* The neighboring sub-cellspace map	Creates an instance of SubSpaceInfo, and initializes the attributes
	SubSpaceInfo(const SubSpaceInfo& *rhs*); IN *rhs* The SubSpaceInfo object to copy from	A copy constructor. Creates an instance which is a copy of *rhs*
	~SubSpaceInfo()	Destroys a SubSpaceInfo object
operator	SubSpaceInfo& **operator=**(const SubSpaceInfo& *rhs*);	Assignment operator. Copies the attribute values from *rhs* to **this*

	IN *rhs* The SubSpaceInfo object to copy from	
	bool **operator==**(const SubSpaceInfo& *rhs*) const; IN *rhs* The SubSpaceInfo object to compare with	Returns true if the two SubSpaceInfo objects are identical
	bool **operator!=**(const SubSpaceInfo& *rhs*) const; IN *rhs* The SubSpaceInfo object to compare with	Returns true if the two SubSpaceInfo objects are different
Attribute Information	int **id**() const;	Returns the ID of the sub-cellspace
	DomDcmpType **domDcmpType**() const;	Returns the domain decomposition type
	const SpaceDims& **glbDims**() const;	Returns the dimensions of the global (whole) cellspace
	const SpaceDims& **dims**() const;	Returns the dimensions of the sub-cellspace
	const CoordBR& **MBR**() const;	Returns the minimum bounding rectangle of the sub-cellspace in the context of the global cellspace. NOTE: in global coordinates
	const CoordBR& **workBR**() const;	Returns the work range of the sub-cellspace. NOTE: in local coordinates
	double **sizeRatio**() const;	Returns the ratio of the sub-cellspace's size over the global cellspace's size
Coordinate Translators	const CellCoord **glbIdx2glbCoord**(int *glbIdx*) const;	Returns the global row-column coordinate corresponding to the given global array index
	int **glbCoord2glbIdx**(const CellCoord& *glbCoord*) const; int **glbCoord2glbIdx**(int *iRowGlb*, int *iColGlb*) const; IN *glbCoord* The global row-column coordiate	Returns the global array index corresponding to the given global row-column coordinate

	IN *iRowGlb* The global row index IN *iColGlb* The global column index	
	const CellCoord **idx2coord**(int *idx*) const; IN *idx* The local array index	Returns the local row-column index corresponding to the given local array index
	int **coord2idx**(const CellCoord& *coord*) const; int **coord2idx**(int *iRow*, int *iCol*) const; IN *coord* The local row-column coordiate IN *iRow* The local row index IN *iCol* The local column index	Rerurns the local array index corresponding to the given local row-column coordinate
	const CellCoord **lclCoord2glbCoord**(const CellCoord& *lclCoord*) const; const CellCoord **lclCoord2glbCoord**(int *iRowLcl*, int *iColLcl*) const; IN *lclCoord* The local row-column coordiate IN *iRowLcl* The local row index IN *iColLcl* The local column index	Returns the global row-column coordinate corresponding to the given local row-column coordinate (Equation 2.4 and 2.5)
	const CellCoord **glbCoord2lclCoord**(const CellCoord& *glbCoord*) const; const CellCoord **glbCoord2lclCoord**(int *iRowGlb*, int *iColGlb*) const; IN *glbCoord* The global row-column coordiate IN *iRowGlb* The global row index	Returns the local row-column coordinate corresponding to the given global row-column coordinate (Equation 2.7 and 2.8)

<table>
<tr><td></td><td>IN iColGlb The global column index</td><td></td></tr>
<tr><td></td><td>int lclCoord2glbIdx(const CellCoord& lclCoord) const;
int lclCoord2glbIdx(int iRowLcl,
int iColLcl) const;

IN lclCoord The local row-column coordiate
IN iRowLcl The local row index
IN iColLcl The local column index</td><td>Returns the global array index corresponding to the given local row-column coordinate (Equation 2.6)</td></tr>
<tr><td></td><td>const CellCoord glbIdx2lclCoord(int glbIdx) const;

IN glbIdx The global array index</td><td>Returns the local row-column coordinate corresponding to the given global array index</td></tr>
<tr><td></td><td>int glbIdx2lclIdx(int glbIdx) const;

IN glbIdx The global array index</td><td>Return the local array index corresponding to the given global array index (Equation 2.11)</td></tr>
<tr><td></td><td>int lclIdx2glbIdx(int lclIdx) const;

IN lclIdx The local array index</td><td>Returns the global array index corresponding to the given local array index (Equation 2.10)</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>Neighbor Information</td><td>int nNbrDirs() const;</td><td>Returns the number of neighboring directions which is directly related to the decomposition type.
<table>
<tr><td>Decomposition</td><td>Number of neighboring directions</td></tr>
<tr><td>NON_DCMP</td><td>0</td></tr>
<tr><td>ROWWISE_DCMP</td><td>2: upper(0) and lower(1)</td></tr>
<tr><td>COLWISE_DCMP</td><td>2: left(0) and right(1)</td></tr>
<tr><td>BLOCK_DCMP</td><td>8: MeshDir</td></tr>
</table></td></tr>
<tr><td></td><td>int nEdges() const;</td><td>Returns the number of edges which is directly related to the decomposition type
<table>
<tr><td>Decomposition</td><td>Number of edges</td></tr>
<tr><td>NON_DCMP</td><td>0</td></tr>
</table></td></tr>
</table>

<table>
<tr><td></td><td></td><td><table><tr><td>ROWWISE_DCMP</td><td>2</td></tr><tr><td>COLWISE_DCMP</td><td>2</td></tr><tr><td>BLOCK_DCMP</td><td>4</td></tr></table></td></tr>
<tr><td></td><td>int nTotNbrs() const;</td><td>Returns the number of neighboring sub-cellspaces</td></tr>
<tr><td></td><td>bool hasNbrs(int iDir) const;

IN iDir The neighboring direction</td><td>Returns true if in the iDir direction there are neighboring sub-cellspaces. NOTE: the valid range of iDir varies with the decomposition types<table><tr><td>Decomposition</td><td>Valid range of iDir</td></tr><tr><td>NON_DCMP</td><td>0</td></tr><tr><td>ROWWISE_DCMP</td><td>0 or 1</td></tr><tr><td>COLWISE_DCMP</td><td>0 or 1</td></tr><tr><td>BLOCK_DCMP</td><td>[0, 7]</td></tr></table></td></tr>
<tr><td></td><td>int nNbrs(int iDir) const;

IN iDir The neighboring direction</td><td>Returns the number of neighboring sub-cellspaces in the iDir direction</td></tr>
<tr><td></td><td>const vector<int>& nbrSubSpcIDs(int iDir) const;

IN iDir The neighboring direction</td><td>Returns the vector of the IDs of the neighboring sub-cellspaces in the iDir direction</td></tr>
<tr><td></td><td>int nbrDir(int nbrID) const;

IN nbrID The ID of the queried neighboring sub-cellspace</td><td>Finds the neighboring sub-cellspace which has the given ID and returns the direction in which it is to *this sub-cellspace</td></tr>
<tr><td></td><td></td><td></td></tr>
</table>

Table 2.9 Attributes of pRPL::SubSpace [14]

Name	Type	Description
_pInfo	const SubSpaceInfo*	The spatial division information about the sub-cellspace
_mSendBRs	vector< vector<CoordBR> >	The map of send ranges (Figure 2.14)
_vEdgeBRs	vector<CoordBR>	The vector of the ranges of edge cells (Figure 2.14)
_interiorBR	CoordBR	The range of the interior cells (Figure 2.14)
_mStream2Send	vector< vector< vector<char> > >	The map of value-headed index streams to send to the neighboring sub-cellspaces
_vStreamRecved	Vector<char>	The value-headed index stream received from the neighboring sub-cellspaces

[14] The SubSpace class is derived from the CellSpace class and inherits all the attributes and methods of the CellSpace class. Also it is a class template, and users can declare a SubSpace object of any data type, e.g., int, double, and even an user-defined type

Table 2.10 Methods of pRPL::SubSpace

	Name	Description
Constructor & Destructor	**SubSpace**();	Creates an empty instance of SubSpace
	SubSpace(const CellSpace<elemType>& *cellSpace*, const SubSpaceInfo& *subSpaceInfo*); IN *cellSpace* The global (whole) cellspace from which the sub-cellspace will be extracted IN *subSpaceInfo* The sub-cellspace's spatial division information	Creates an instance of SubSpace based on the spatial division information given in *subSpaceInfo*, and copies the corresponding cell values from the global (whole) cellspace.
	SubSpace(const SubSpaceInfo& *subSpaceInfo*); IN *subSpaceInfo* The sub-cellspace's spatial division information	Creates an instance of SubSpace based on the spatial division information given in subSpaceInfo, and allocates memory space for storing the cell values.
	SubSpace(const SubSpace<elemType>& *rhs*); IN rhs The SubSpace object to copy from	Copy constructor. Creates a SubSpace object which is a copy of *rhs*, including the cell values stored in *rhs*.
	~SubSpace();	Destroys the SubSpace object
Operator	SubSpace<elemType>& **operator=**(const SubSpace<elemType>& *rhs*); IN rhs The SubSpace object to copy from	Assignment operator. Copies the attribute values from *rhs* to **this*, including the cell values.
Msg	bool **makeStream2Send**();	Makes the value-headed index streams (i.e., *_mStreams2Send*), which are to be sent to the neighboring sub-cellspaces, from the updated-cell map, i.e.,

		_mUpdtCell.
	bool **loadStreamRecved**(Transition<elemType>* *pTransition* = 0, Neighborhood<elemType>* *pNbrhood* = 0); OPT_IN *pTransition* The Transition used to evaluate and update the sub-cellspace OPT_IN *pNbrhood* The Neighborhood used to evaluate and update the sub-cellspace	Loads the updated "halo" cells stored in the value-headed index stream received from the neighboring sub-cellspaces (i.e., *_vStreamRecved*). If a finalizing process is required (i.e., the *needFinalize* option of the Transition object is ON), the Transition object and Neighborhood object must be given.
	void **cleanStreamBuf**();	Cleans both the sending and received message streams
Updating Tools	bool **calcBRs**(const Transition<elemType>* *pTransition*, const Neighborhood<elemType>* *pNbrhood*, const SubInfoVect* [15] *pvSubInfos*); IN *pTransition* The Transition used to evaluate and update the sub-cellspace IN *pNbrhood* The Neighborhood used to evaluate and update the sub-cellspace IN *pvSubInfos* The set of spatial division information of the sub-cellspace and its neighboring sub-cellspaces	Calculates the critical bounding rectangles (i.e., *_mSendBRs, _vEdgeBRs,* and *_interiorBR*) based on the given Transition, Neighborhood, the spatial division information embedded in the SubSpaceInfo objects (Figure 2.14). NOTE: not only the spatial division information of **this* sub-cellspace is needed, but also the spatial division information of its neighboring sub-cellspaces must be given.
	bool **updateEdges**(Transition<elemType>* *pTransition*, Neighborhood<elemType>* *pNbrhood*);	Evaluates and updates the Edge cells of the sub-cellspace using the given Transition and Neighborhood objects

[15] The SubInfoVect class is a vector class for storing a set of SubSpaceInfo objects. For more information, see subInfoVect.h and subInfoVect.cpp

	IN *pTransition* The Transition used to evaluate and update the sub-cellspace IN *pNbrhood* The Neighborhood used to evaluate and update the sub-cellspace	
	bool **updateEdges**(vector<int>& *vIdxs2Eval*, Transition<elemType>* *pTransition*, Neighborhood<elemType>* *pNbrhood*); IN/OUT *vIdxs2Eval* The set of array indices of the cells to be evaluated IN *pTransition* The Transition used to evaluate and update the sub-cellspace IN *pNbrhood* The Neighborhood used to evaluate and update the sub-cellspace	Given a set of array indices of cells (*vIdxs2Eval*), only evaluates and updates those that are included in the Edge bounding rectangles (i.e., *_vEdgeBRs*) using the given Transition and Neighborhood objects. The evaluated cells will be removed from the index set. NOTE: if the *needFinalize* option or *needExchange* option of the Transition is ON, the updated cells will be stored in the updated-cell map, i.e., *_mUpdtCells*.
	bool **updateInterior**(Transition<elemType>* *pTransition*, Neighborhood<elemType>* *pNbrhood*); IN *pTransition* The Transition used to evaluate and update the sub-cellspace IN *pNbrhood* The Neighborhood used to evaluate and update the sub-cellspace	Evaluates and updates the Interior cells of the sub-cellspace using the given Transition and Neighborhood objects
	bool **updateInterior**(vector<int>& *vIdxs2Eval*, Transition<elemType>* *pTransition*, Neighborhood<elemType>* *pNbrhood*); IN/OUT *vIdxs2Eval* The set of array indices of the cells to be	Given a set of array indices of cells (*vIdxs2Eval*), only evaluates and updates those that are included in the Interior bounding rectangles (i.e., *_interiorBR*) using the given Transition and Neighborhood objects. The evaluated cells will be removed from the index set. NOTE: if the *needFinalize* option or *needExchange* option of the Transition is ON, the updated cells will be stored

	evaluated IN *pTransition* The Transition used to evaluate and update the sub-cellspace IN *pNbrhood* The Neighborhood used to evaluate and update the sub-cellspace	in the updated-cell map, i.e., *_mUpdtCells*.
	bool **updateAll**(Transition<elemType>* *pTransition*, Neighborhood<elemType>* *pNbrhood*); IN *pTransition* The Transition used to evaluate and update the sub-cellspace IN *pNbrhood* The Neighborhood used to evaluate and update the sub-cellspace	Evaluates and updates all the cells of the sub-cellspace. NOTE: if the *needFinalize* option or *needExchange* option of the Transition is ON, the updated cells will be stored in the updated-cell map, i.e., *_mUpdtCells*.
	bool **updateAll**(vector<int>& *vIdxs2Eval*, Transition<elemType>* *pTransition*, Neighborhood<elemType>* *pNbrhood*); IN/OUT *vIdxs2Eval* The set of array indices of the cells to be evaluated IN *pTransition* The Transition used to evaluate and update the sub-cellspace IN *pNbrhood* The Neighborhood used to evaluate and update the sub-cellspace	Given a set of array indices of cells (*vIdxs2Eval*), evaluates and updates those that are within the sub-cellspace using the given Transition and Neighborhood objects. The evaluated cells will be removed from the index set. NOTE: if the *needFinalize* option or *needExchange* option of the Transition is ON, the updated cells will be stored in the updated-cell map, i.e., *_mUpdtCells*.

Table 2.11 Attributes of pRPL::Layer [16]

Name	Type	Description
_pPRPrc	PRProcess*	The process that holds the layer
_name	string	The name of the layer
_pCellSpace	CellSpace<elemType>*	The global (whole) cellspace. NOTE: usually this pointer is NULL on slave processes, and points to a CellSpace object on the master process before the data is decomposed and distributed to the processes, and after the data is gather from the processes. One special case is that when a whole cellspace is required on every process, this pointer on every process points to the whole cellspace (see Layer2 in Figure 2.15).
_pNbrhood	Neighborhood<elemType>*	The Neighborhood object used to decompose the cellspace, and evaluate the cellspace.
_vpSubSpcs	vector< SubSpace<elemType>* >	The set of SubSpace objects held by the process (see Layer0 and Layer1 in Figure 2.15)
_mGatherTypes	map<int, MPI_Datatype>	The data types used to gather cell values from the SubSpace objects, each type corresponds to a SubSpace object

[16] The Layer class is a class template. By giving a data type when declaring a Layer object, the user also determines the data type for the CellSpace, SubSpace, and Neighborhood objects, which are stored in the Layer object.

Table 2.12 Methods of pRPL::Layer

	Name	Description
Constructors & Destructor	**Layer**(); **Layer**(PRProcess& *prPrc*, const string *layerName* = "Untitled"); IN *prPrc* The process that holds the Layer OPT_IN *layerName* The name of the Layer	Creates an empty Layer object Creates a Layer object held by the process *prPrc*, and name it.
	Layer(const Layer<elemType>& *rhs*); IN *rhs* The Layer object to copy from	Copy constructor. Creates a Layer object which is a copy of *rhs*, including the whole cellspace and the local sub-cellspaces.
	~Layer();	Destroys a Layer object
Operators	Layer<elemType>& **operator=**(const Layer<elemType>& *rhs*); IN *rhs* The Layer object to copy from	Assignment operator. Copies all the attribute values to **this* from *rhs*, including the whole cellspace and the local sub-cellspaces.
	const SubSpace<elemType>* **operator[]**(int *iSubSpc*) const; SubSpace<elemType>* **operator[]**(int *iSubSpc*); IN *iSubSpc* The index of a SubSpace object in the vector, i.e., *_vpSubSpcs*	Returns a pointer pointing to a local sub-cellspace.
Attribute Information	const string& **name**() const;	Returns the name of the Layer object
	void **name**(const string& *layerName*);	Sets the name of the Layer object
	PRProcess *const **prPrc**() const;	Returns a pointer pointing to the process that holds the Layer object
	int **id**() const;	Returns the ID of the process that holds the Layer object

	const string **title**() const;	Returns the title of the Layer object. The title of a Layer consists of the name of the Layer and the ID of the process that holds the Layer object
	bool **isMaster**() const;	Returns true if the process that holds the Layer object is the master of the process group
	bool **hasCellSpace**() const;	Returns true if the Layer object stores a CellSpace object
	bool **hasNbrhood**() const;	Returns true if the Layer object stores a Neighborhood object
	const CellSpace<elemType>* **cellSpace**() const; CellSpace<elemType> ***cellSpace**();	Returns a pointer pointing to the CellSpace object stored in the Layer object
	const Neighborhood<elemType> ***nbrhood**() const; Neighborhood<elemType> ***nbrhood**();	Returns a pointer pointing to the Neighborhood object stored in the Layer object
Decomposition & Distribution	bool **newCellSpace**(); bool **newCellSpace**(const SpaceDims& *dims*); bool **newCellSpace**(const SpaceDims& *dims*, const elemType& *initVal*); bool **newCellSpace**(int *nRows*, int *nCols*); bool **newCellSpace**(int *nRows*, int *nCols*, const elemType& *initVal*);	Creates a CellSpace object in the Layer object.
	bool **newNbrhood**();	Creates a Neighborhood object in the Layer object

	bool **newNbrhood**(const vector<CellCoord>& *vNbrCoords*, double *weight* = 1.0); bool **newNbrhood**(const vector<CellCoord>& *vNbrCoords*, const vector<double>& *vNbrWeights*); template<class elemType2> bool **newNbrhood**(const Neighborhood<elemType2>& *rhs*);	
	bool **smplDcmpDstrbt**(DomDcmpMethod *dcmpMethod*, int *nSubSpcs1*, int *nSubSpcs2* = 1, bool *dstrbtData* = true); IN *dcmpMethod* The decomposition method used to divide the global cellspace IN *nSubSpcs1* The number of sub-cellspaces when row-wise or column-wise decomposition is used. Or the number of rows of sub-cellspaces when block-wise decomposition is used OPT_IN *nSubSpcs2* Equals to 1 if row-wise or column-wise decomposition is used. The number of columns of sub-cellspaces when block-wise decomposition is used. OPT_IN *dstrbtData* Whether distribute cell values from the	First, decomposes the global (whole) CellSpace object on the master process using a simple decomposition method (e.g., row-wise, column-wise, and block-wise). Second, maps the sub-cellspaces onto the processes, and creates corresponding SubSpace objects on the processes for storing the sub-cellspaces. Third, distributes the cell values from the master process to the processes that hold the sub-cellspaces, if *dstrbtData* is true.

	master process to the processes holding the sub-cellspaces	
	bool **quadDcmpDstrbt**(Transition<elemType>& *transition*, int *maxNumLeaves*, int *minWorkload* = 0, bool *dstrbtData* = true); IN *transition* The Transition object used to estimate the workload of a sub-cellspace IN *maxNumLeaves* The maximum number of quad-tree leaves. The quad-tree will keep growing until the *maxNumLeaves* is reached. OPT_IN *minWorkload* The minimum workload of a sub-cellspace. The quad-tree will keep growing until the *minWorkload* is reached. OPT_IN *dstrbtData* Whether distribute cell values from the master process to the processes holding the sub-cellspaces	First, decomposes the global (whole) CellSpace object on the master process using the quad-tree decomposition method Second, maps the sub-cellspaces onto the processes, and creates corresponding SubSpace objects on the processes for storing the sub-cellspaces. Third, creates the SubSpace objects on the processes to accommodate the cell values. Four, distributes the cell values from the master process to the processes that hold the sub-cellspaces, if *dstrbtData* is true.

	bool **distribute**(bool *dstrbtData* = true);
	
	OPT_IN *dstrbtData* Whether distribute cell values from the master process to the processes holding the sub-cellspaces
		Creates the SubSpace objects on the processes to accommodate the cell values.
		Distributes the cell values from the master process to the processes that hold the sub-cellspaces, if *dstrbtData* is true.
		NOTE: this method can only be called after the decomposition and mapping are done. It is especially useful when multiple Layers are to be decomposed and distributed.
		`Exampe:`
		`// layer1 and layer2 are held by the same process`

		`layer1. smplDcmpDstrbt(SMPL_ROW, 10); // decomposing and distributing`
		`layer2.distribute(); // distributing using the same spatial divisions and mapping scheme`

	bool **broadcast**(bool *bcastData* = true); OPT_IN *dstrbtData* Whether distribute cell values from the master process to the processes holding the sub-cellspaces	Creates a CellSpace object for storing the global cellspace on every slave process. Distributes the cell values from the master process if *dstrbtData* is true.
Local Information	int **nLclSubSpcs**() const;	Returns the number of local sub-cellspaces stored in the Layer object
	const SubSpace<elemType>* **findSubSpc**(int *subID*) const; SubSpace<elemType>* **findSubSpc**(int *subID*); IN *subID* The ID of the sub-cellspace	Returns a pointer to the sub-cellspace whose ID matches *subID*
	double **sizeRatio**() const;	Returns the ratio between the sum of the sizes of the sub-cellspaces stored in the Layer object and the size of the global (whole) cellspace
Updating	bool **update**(Transition<elemType>& *transition*); IN transition The Transition object used to update the sub-cellspaces	Updates the sub-cellspaces stored in the Layer object using the given Transition object
	bool **update**(map<int, vector<int> >& *mLclIdxs2Eval*, Transition<elemType>& *transition*); IN *mLclIdxs2Eval* The map of local array indices of cells for sub-cellspaces. The key field of the map is the ID of a sub-cellspace, and the value	Given a map of local array indices of cells for the sub-cellspaces, updates them using the given Transition object

	field is a vector of local array indices of cells to be evaluated IN *transition* The Transition object used to update the sub-cellspaces	
	bool **update**(vector<int>& *vGlbIdxs2Eval*, Transition<elemType>& *transition*); IN *vGlbIdxs2Eval* The vector of global array indices of cells to be evaluated IN *transition* The Transition object used to update the sub-cellspaces	Given a set of global array indices of cells, updates them using the given Transition object
	bool **buildGatherTypes**();	Makes a set of data types for gathering cell values from processes to the master process
	void **freeGatherTypes**();	Releases the data types created for gathering cell values. NOTE: it should be called after the **gatherCellSpace** method.
	bool **gatherCellSpace**();	Gathers the cell values from the processes that hold the sub-cellspaces to the master process that holds the global cellspace

Table 2.13 Attributes of pRPL::Transition

Name	Type	Description
_pCellSpace	CellSpace<elemType>*	The CellSpace on which the Transition is to be applied
_pNbrhood	Neighborhood<elemType>*	The Neighborhood to be used for the Transition
_onlyUpdtCtrCell	bool	True if the Transition is centralized, i.e., only updating the central cell of the Neighborhood
_needFinalize	bool	True if the Transition requires a finalizing process
_needExchange	bool	True if the Transition requires data exchange between sub-cellspaces
_edgesFirst	bool	True if the Transition processes the Edge cells before starting the non-blocking data exchange between sub-cellspaces

Table 2.14 Methods of pRPL::Transition

	Name	Description
Constructors & Destructor	**Transition**(bool *onlyUpdtCtrCell* = true, bool *needFinalize* = true, bool *needExchange* = true, bool *edgesFirst* = true); OPT_IN *onlyUpdtCtrCell* Option OPT_IN *needFinalize* Option. NOTE: the *finalize* method must be overloaded by the user if this option is true. OPT_IN *needExchange* Option OPT_IN *edgesFirst* Option. NOTE: It can NOT be true if *needExchange* is false	Creates a Transition object, and sets the options
	virtual **~Transition**() {}	Destroys a Transition object
Option Setting	bool **onlyUpdtCtrCell**() const;	Returns true if the Transition is a centralized algorithm
	bool **needFinalize**() const	Returns true if the Transition includes a finalizing process, i.e., the **finalize** method is customized and overloaded
	bool **needExchange**() const;	Returns true if the Transition requires data exchange between sub-cellspaces
	bool **edgesFirst**() const;	Returns true if the Transition processes the Edge cells before starting the non-blocking data exchange between sub-cellspaces
	virtual bool **cellSpace**(CellSpace<elemType>* *pCellSpc*); IN *pCellSpc* The CellSpace object on which the	Specifies a CellSpace (or SubSpace) object for the Transition, and this CellSpace (or SubSpace) will be evaluated and updated by using this Transition. NOTE: when multiple cellspaces are used, this method must be customized and

	Transition is to be applied	overloaded (see section 2.5.2 and example 2.5)
	virtual bool **nbrhood**(Neighborhood<elemType>* *pNbrhood*); IN *pNbrhood* The Neighborhood to be used to evaluated and update the specified CellSpace	Specifies a Neighborhood object for the Transition, and this Neighborhood will be used to evaluate and update the CellSpace (or SubSpace) specified using the **cellSpace** method
	virtual bool **evaluate**(vector< pair<int, elemType> >& *vUpdtedCells*, const CellCoord& *coord*); OUT *vUpdtedCells* The vector of index-value pairs of the updated cells. If the Transition is a non-centralized algorithm, there may be multiple updated cells within a neighborhood. NOTE: The index is the local array index of the cell. IN *coord* The local row-column coordinate of the cell to be evaluated, i.e., the central cell of the neighborhood	Implements the user-defined algorithm for evaluating and updating the cellspace. It must be customized and overloaded by users (see section 2.5.3 and example 2.6).
	virtual bool **finalize**(const elemType& *val*, const CellCoord& *coord*); IN *val* The temporary value of the updated cell IN *coord* The local row-column coordinate of the updated cell	Implements the finalizing process of the algorithm to finally update the value of the updated cell. It must be customized and overloaded if the *needfinalize* option is ON (true) (see section 2.5.4 and example 2.7).
	virtual int **workload**(const CoordBR& *workBR*); IN *workBR* The region in the cellspace of which the	Calculates the workload of the given region (defined by *workBR*) in the cellspace. It must be overloaded if the quad-tree decomposition is to be used (see section 2.5.5 and example 2.8).

	workload is to be calculated	