



Container

Credits for the logo of Morgana: Paola Infantino
(<https://paolainfantino.com>)

The authors wish to thank L. Barbareschi, for her valuable contribution and suggestions.

Download the code at: <https://github.com/feof81/morganaPublic>.

Contents

1	Introduction	2
2	Serial containers	3
3	Oct trees	6
4	Linear databases	10
5	The maps	12
5.1	Map items	12
5.2	Map	16
6	Maps manipulation	18
6.1	Serial map manipulation	18
6.2	Map communication	22
6.3	Parallel map manipulation	26
7	Parallel vectors	32
8	Manipulators of parallel vectors	36
8.1	Serial manipulation of parallel vectors	36
8.2	Parallel vector communication	41
8.3	Parallel manipulation of parallel vectors	54
9	Traits classes	62
10	Tutorials	67

1 Introduction

The parallel distribution of data is a rather involved task in a parallel, distributed memory environment. In a serial code the *stl* library provides almost all the functionalities needed: in an ordered list the position inside the list provides the ordering and the key to access the data. On the contrary, in a parallel environment each data is characterized by a local and a global key, not all the data are present on a single process and some data may be repeated on several processes.

In *morgana*, to every single piece of data we associate:

- pid : a process identification identifier (id). This usually coincides with the process where the piece of data is stored but during some communication processes this id may represent other information such as the origin of the piece of data;
- lid : the local id. This usually coincides with the current position inside a given local vector;

- gid : the global id.

The numbering of *lids* and *gids* starts from one while the *pid* numbering starts from zero due to the fact that it follows the standard *mpi* numbering of the processes. Let us now give an overview of the package of containers, this can be roughly divided in:

- serial container support: definition of some serial containers such as *sVect* and *sArray* which are wrappers of some *stl* containers;
- octTrees: an implementation of an octal tree structure mainly used in the *geometry* package for search purposes;
- linear databases: a data handling class that interpolates the behavior of a coefficient with respect to a given parameter;
- maps and maps manipulation: it is the backbone of the parallel handling of data. It defines the structure of data distribution;
- parallel vectors and manipulators: it is the main container that stores the data and distributes it across the communicator.

In the rest of this manual we will describe in detail all these subsections.

2 Serial containers

The *simpleFormats.hpp* contains the main serial container that is *sVect*. This is simply a wrapper of the *vector* class defined in the *stl* library that contains some functions for the bounds checking and to support the parallel sending of data through data serialization. This is one of the features provided by the *boost – mpi* implementation. As the *vector* class, *sVect* is templated and only the types discussed in the *morgana dofs* manual can be used since they implement too the serialization technique.

We briefly outline the *sVect* declaration

```
template <typename T, int OFFSETVEC = 1>
class sVect : public vector<T>
{
    /*! @name Typedefs */ //@{
public:
    typedef std::vector<T> raw_container;
    typedef typename raw_container::size_type size_type;
    typedef typename raw_container::reference reference;
    typedef typename raw_container::const_reference const_reference;
    //@}

    /*! @name Parallel support */ //@{
public:
    friend class boost::serialization::access;

    UInt srlzSize; //Serialization size

template<class ARK>
```

```

void serialize(ARK & ar, const unsigned int version);
//@}

/*! @name Constructors */ //@{
public:
/*! Constructor */
explicit sVect( size_type i ) : raw_container( i ) {};

/*! Constructor */
explicit sVect( const raw_container & );

/*! Constructor */
sVect() : raw_container() {};

/*! Copy constructor */
sVect( const sVect<T, OFFSETVEC> & );

/*! Destructor */
~sVect() {}
//@}

/*! @name Operators */ //@{
public:
/*! Equality operator */
sVect<T, OFFSETVEC> & operator=( const sVect<T, OFFSETVEC> & );

/*! Get - with memory check */
T & get( size_type i );

/*! Get - with memory check */
const T & get( size_type i ) const;

/*! Get - NOT memory check */
inline reference operator() ( size_type const i );

/*! Get - NOT memory check */
inline const_reference operator() ( size_type const i ) const;

/*! Data clean */
void clean();

/*! Memory check */
inline bool bCheck( size_type const i ) const
{ return i >= OFFSETVEC && i < this->size() + OFFSETVEC ; }
//@}

/*! @name Printout */ //@{
public:
template<typename TT, int OO>
friend ostream & operator<<(ostream & f, const sVect<TT,OO> & V);
//@}
};

```

and we point out that data can be accessed either with the `()` or with the `[]` operators. In the first case the indexes start from the *OFFSETVEC* specified as the second template parameter. If not specified the default value is one. On

the contrary, the `[]` operator is inherited directly from the *vector* class and the indexes, in this case, start from one. We also point out that the class implements the *operator <<* to ease its printout. The *serialize* method has a couple of template specializations to better implement the serialization of the *Real* and *UInt* types:

```

/! Specialization for Real */
template <>
template<class ARK>
void
sVect<Real,1>::
serialize(ARK & ar, const unsigned int version)
{
    assert(version == version);

    UInt n = this->size();
    ar & n;

    if(n != this->size())
    { this->resize(n); }

    for(UInt i=1; i <= this->size(); ++i)
    {
        ar & this->get(i);
    }
}

/! Specialization for UInt */
template <>
template<class ARK>
void
sVect<UInt,1>::
serialize(ARK & ar, const unsigned int version)
{
    assert(version == version);

    UInt n = this->size();
    ar & n;

    if(n != this->size())
    { this->resize(n); }

    for(UInt i=1; i <= this->size(); ++i)
    {
        ar & this->get(i);
    }
}

```

In general all types that do not implement the serialization function must be associated to a proper specialization of the *serialize* function. The *simpleFormats.hpp* file contains also the *sArray* class

```

template <typename T, int OFFSETVEC = 1>
class sArray : public vector<T>
{
    /! @name Typedefs */ //@{
public:
    typedef vector<T> raw_container;

```

```

typedef typename raw_container::size_type size_type;
typedef typename raw_container::reference reference;
typedef typename raw_container::const_reference const_reference;
typedef typename raw_container::iterator iterator;
typedef typename raw_container::const_iterator const_iterator;
//@}

/*! @name Internal data */ //@{
private:
size_type _M_nrows;
size_type _M_ncols;
//@}

/*! @name Constructors */ //@{
public:
explicit sArray( size_type ntot );
explicit sArray();
explicit sArray( size_type nrows, size_type ncols );
~sArray() {}
//@}

/*! @name Operators */ //@{
public:
reference operator() ( size_type const i );
const_reference operator() ( size_type const i ) const;
reference operator() ( size_type const i, size_type const j );
const_reference operator() ( size_type const i, size_type const j ) const;
iterator columnIterator( size_type const col );
//@}

/*! @name Size functions */ //@{
public:
void reshape( sArray<T, OFFSETVEC>::size_type const n, size_type const m );
void clean();
bool bCheck( size_type const i, size_type const j ) const;
inline size_type nrows() const;
inline size_type ncols() const;
//@}
};

```

that implements a double-index array.

3 Oct trees

An *octTree* is a tree of connected elements, in the case treated here, *sOctTreeItems*: there is a starting element known as the root. Except the root, every *sOctTreeItem* has one father. Every element has exactly eight sons, except terminal elements, which have none, they are also known as the leaves. The *octTrees* are heavily exploited for search structures especially for geometric meshes. In our implementation each *sOctTreeItem* is linked to a piece of data, for instance a list of nodes belonging to a given space region.

Let's first describe the structure of the base element of an *octTree*:

```

/*! The single leaf of a tree */
class sOctTreeItem
{

```

```

/*! @name Internal data */ //@{
public:
  UInt father;
  UInt sons[8];
  bool isLeaf;
  UInt id;
  //@}

/*! @name Constructors */ //@{
public:
  sOctTreeItem();
  sOctTreeItem(const UInt & Id);
  sOctTreeItem(const sOctTreeItem & O);
  sOctTreeItem & operator=(const sOctTreeItem & O);
  static UInt octMap(const bool & ix, const bool & iy, const bool & iz);
  //@}

```

the *id* integer defines the identification code of the specific *sOctTreeItems*. Two *sOctTreeItem* in an *octTree* must have different *ids*. *father* designates the *id* of the father element, *isLeaf* defines whether the *sOctTreeItem* is a terminal element or, in other terms if it is a leaf. In that case it has no sons. Otherwise, if the element is not a leaf, then the array *sons* contains the *ids* of the eight sons.

Sons can be identified by a local integer spanning from 1 to 8 however there is another useful way to identify them. In fact *octTree* is mainly used to decompose the space in two halves in three dimensions i.e. $2^3 = 8$. Therefore each son can also be identified by three Boolean variables (*ix, iy, iz*). To create a correspondence between the two systems we have implemented the following static method:

```

UInt
sOctTreeItem::
octMap(const bool & ix, const bool & iy, const bool & iz)
{
  return(UInt(ix) + UInt(iy) * 2 + UInt(iz) * 4);
}

```

The *sOctTreeItems* can be ordered using the internal *id* as the key as we can see in the definition

```

/*! @name Comparison functions */ //@{
public:
  bool operator<(const sOctTreeItem & V) const;
  bool operator!=(const sOctTreeItem & V) const;
  //@}

```

and implementation of the ordering functions

```

bool
sOctTreeItem::
operator<(const sOctTreeItem & V) const
{
  return(id < V.id);
}

bool
sOctTreeItem::
operator!=(const sOctTreeItem & V) const
{

```



```

return(id != V.id);
}

```

The set functions are used to set and modify the internal identification variables of the class:

```

/*! @name Set functions */ //@{
public:
void setFather(const UInt & Father);
void setSons(const UInt VVV, const UInt FVV, const UInt VFV, const UInt FFV,
             const UInt VVF, const UInt FVF, const UInt VFF, const UInt FFF);
void setSons(const sVect<UInt> Sons);
void setIsLeaf(const bool & IsLeaf);
void setId(const UInt & Id);
//@}

```

We point out that sons can be set with the following ordering:

```

void
sOctTreeItem::
setSons(const UInt VVV, const UInt FVV, const UInt VFV, const UInt FFV,
        const UInt VVF, const UInt FVF, const UInt VFF, const UInt FFF)
{
sons[0] = VVV;
sons[1] = FVV;
sons[2] = VFV;
sons[3] = FFV;

sons[4] = VVF;
sons[5] = FVF;
sons[6] = VFF;
sons[7] = FFF;
}

```

The same data can also be extracted from the class using the following functions. We stress that, here too, the identification code of a son can be extracted either by specifying its number, ranging from 1 to 8, or by specifying the Boolean variables (ix, iy, iz):

```

/*! @name Const get Functions */ //@{
public:
const UInt & getFather() const;
const UInt & getSons(const bool & ix, const bool & iy, const bool & iz) const;
const UInt & getSons(const UInt & i) const;
const bool & setIsLeaf() const;
const UInt & getId() const;
//@}

/*! @name Get functions */ //@{
public:
UInt & getFather();
UInt & getSons(const bool & ix, const bool & iy, const bool & iz);
UInt & getSons(const UInt & i);
bool & setIsLeaf();
UInt & getId();
//@}

/*! @name Outstream operator */ //@{
public:

```

```
friend ostream & operator<<(ostream & f, const sOctTreeItem & P);
/*@}
```

Let us now pass to describe the structure of the *sOctTree* class. The letter *s* reminds that it is a local, serial class not directly involved with parallel communication processes.

```
template<typename DATA>
class sOctTree
{
    /*! @name Typedefs */ //@{
public:
    typedef std::map<sOctTreeItem,DATA>      DATAMAP;
    typedef typename DATAMAP::iterator      ITER;
    typedef typename DATAMAP::const_iterator CONST_ITER;
    //@}

    /*! @name Internal data */ //@{
public:
    UInt k;
    ITER iter;
    DATAMAP data;
    //@}

    /*! @name Constructors and functions */ //@{
public:
    sOctTree();
    sOctTree(const sOctTree & O);
    sOctTree & operator=(const sOctTree & O);
    void clear();
    //@}
```

All *sOctTreeItems* elements of the tree are contained in an *std :: map* that sorts the elements using the *id* as a key. Each element is linked in the *std :: map* to a data of type *DATA*. *iter* represents a pointer to the local position inside the *map*. The index *k* defines the maximum index *id* in the tree. The pointer can be displaced using the following functions:

```
/*! @name Tree handle */ //@{
public:
    void restart();
    void goDown(const UInt & i);
    void goDown(const bool & ix, const bool & iy, const bool & iz);
    void goUp();
    void goTo(const UInt & Id);
    //@}
```

the *restart()* method sets the pointer *iter* to the root of the tree, *goDown* descends the tree by one level letting the user choose which of the eight sons to consider either by setting $i \in [1, 8]$ or by setting three *bool ix, iy, iz*. The *goUp()* function moves the pointer to the father of the current element and *goTo* moves the pointer to the element identified by the identification integer *Id*.

```
/*! @name Set functions */ //@{
public:
    void setData(const DATA & Data);
    void setData(const UInt & Id, const DATA & Data);
    sVect<UInt> addLeaves();
    sVect<UInt> addLeaves(const UInt & Id);
```

```

sVect<UInt> addLeaves(const DATA & VVV, const DATA & FVV, const DATA & VFV, const DATA & FFV,
                    const DATA & VVF, const DATA & FVF, const DATA & VFF, const DATA & FFF);

sVect<UInt> addLeaves(const UInt & Id,
                    const DATA & VVV, const DATA & FVV, const DATA & VFV, const DATA & FFV,
                    const DATA & VVF, const DATA & FVF, const DATA & VFF, const DATA & FFF);
//@}

```

We now pass to describe the functions to handle the data coordinated with the tree. The *setData(const DATA Data)* method sets the piece of data corresponding to the element pointed by *iter*, *setData(const UInt Id, const DATA Data)* sets the data of the element corresponding to the identification integer *Id*. Then four different functions are available to add *leaves* to the tree: the first generates eight sons from a given *sOctTreeItem* pointed by *iter*. The method first checks whether the pointed element is in fact a leaf and returns an *sVect < UInt >* vector containing the *ids* of the eight new sons. The second method carries out the same procedure but the starting leaf is specified by the integer *Id*. The third *addLeaves* function is very similar, however, in the creation process it adds the corresponding data to the eight children and the last one operates on a given element of the tree identified by index *Id*.

```

/*! @name Const get functions */ //@{
public:
const sOctTreeItem & getMap() const;
const sOctTreeItem & getMap(const UInt & Id) const;
DATA          & getData();
const DATA & getData() const;
DATA          & getData(const UInt & Id);
const DATA & getData(const UInt & Id) const;
//@}

/*! @name Outstream operator */ //@{
public:
template<typename D>
friend ostream & operator<<(ostream & f, const sOctTree<D> & O);
//@}

```

Finally, there are some get functions to extract both data and the *sOctTreeItems* either from the element pointed by *iter* or by the one defined by the index *Id*.

4 Linear databases

We comment the *dataBaseLin1d.hpp* class that defines a piece-wise linear function $y = f(x)$ and is used to interpolate, given a value x , the corresponding value y where only a list of couples x_k, y_k is known.

```

/*! Data base 1d with linear interpolation */
template<typename X, typename Y>
class dataBaseLin1d
{
/*! @name Internal data */ //@{
public:
Real scaleX, scaleY;
std::map<X,Y> data;
//@}

```

```

/*! @name Constructors */ //@{
public:
dataBaseLin1d();
dataBaseLin1d(const sVect<X> & xx, const sVect<Y> & yy);
dataBaseLin1d(const std::map<X,Y> & Data);
dataBaseLin1d(const dataBaseLin1d & A);
dataBaseLin1d & operator=(const dataBaseLin1d & A);
//@}

```

The class is a template both with respect to the input data X and the output type Y . The type X is usually a *Real*. Data are placed in an *std::map* whose keys are the elements of X . Some constructors are available so that a list of x and y values can be loaded: the length of xx and yy vectors must be the same. Let us now pass to the data insertion functions:

```

/*! @name Data management */ //@{
public:
void clear();
void addPoint(const X & x, const Y & y);
void addPoint(const sVect<X> & xx, const sVect<Y> & yy);
bool replacePoint(const X & x, const Y & y);
void setScaleX(const Real & ScaleX);
void setScaleY(const Real & ScaleY);
//@}

```

the *clear* function deletes all the data, there are two *addPoint* functions: one adds only an x,y pair while one adds a sequence. The *replacePoint* method looks for a specific value x and if this is found, it substitutes the old y value with the new one given as the second argument of the function. Finally it returns *true*. If no x is found, then the method returns *false*. The functions *setScaleX* and *setScaleY* multiply by a constant X and Y data respectively.

```

/*! @name Interpolation */ //@{
public:
const X & getMinX() const;
const X & getMaxX() const;
bool isInternal(const X & x) const;
Y interp(const X & x);
//@}

```

```

/*! @name Outstream operators */ //@{
public:
template<typename XX, typename YY>
friend ostream & operator<<(ostream & f, const dataBaseLin1d<XX,YY> & M);
//@}

```

Then we pass to get functions. Methods *getMinX* and *getMaxX* return the minimum and the maximum values of the X interval, *isInternal* checks whether the variable x is internal to the interval and *interp* returns the value y corresponding to x using a piece-wise linear interpolation of data and a zero order extrapolation if x does not belongs to the interval of X data.

5 The maps

5.1 Map items

The key aspect of the parallel distribution of data is to keep track of the information regarding local and global ids. In *morgana* there are two main classes that store parallel distribution data i.e.: *pMapItem* and *pMapItemSharing*. They are also known as map items. The first class stores essentially only the *pid*, *lid* and *gid* indexes for a given piece of data while the second one adds two logic indexes i.e. *shared* and *owned* that specify whether the data is repeated in another process and, if it is shared, if it is the owner of that piece of data. **There is only one owner across the communicator, all others are considered copies of the owner.** For instance in Table 1 we have shown

pid	lid	gid	shared	owned	pid	lid	gid	shared	owned
0	1	1	F	T	1	1	3	T	F
0	2	2	F	T	1	2	4	T	T
0	3	3	T	T	1	3	5	F	T
0	4	4	T	F	1	4	6	F	T

Table 1: An example of map using *pMapItemSharing*. The symbol *T* means true while *F* false.

a representative distribution map involving two processes that contain six total elements. All not-shared items are owned, all shared items have only one owner across the communicator. **All *lid* and *gid* indexes start from one and are consecutive.** There exists a third type of map item, i.e. *pMapItemSedRecv*, that is dedicated to the inter-communication between processes. This class does not specify the distribution of data rather it describes how data must be passed between processes. It is inherited from the base class *pMapItem* and it adds positive integers *sid* and *rid* which indicate the *pid* from which the piece of data is coming from (*sid* - sending id) and the *pid* to which the piece of data is going (*rid* - receiving id). In this case *lid* and *gid* indexes represent the local and global identification *ids* of the data to be sent. **Since *pid* and *rid* indexes represent some process *ids* they start from one.**

Let us now pass to describe in detail the class *pMapItem*:

```
class pMapItem
{
    /*! @name Parallel support */ //@{
public:
    friend class boost::serialization::access;

    template<class ARK>
    void serialize(ARK & ar, const unsigned int version);
    //@}

    /*! @name Internal data */ //@{
public:
    static const pMapItems parallelType = pMapPlain;
    UInt lid, gid, pid, bufLid;
    //@}
```

```

    /*! @name Constructors */ //@{
    public:
    /*! Constructor */
    pMapItem();

    /*! Constructor
    \param Lid local id
    \param Gid global id*/
    pMapItem(const UInt & Lid, const UInt & Gid);

    /*! Constructor
    \param Lid local id
    \param Gid global id
    \param Pid process id*/
    pMapItem(const UInt & Lid, const UInt & Gid, const UInt & Pid);

    /*! Copy constructor */
    pMapItem(const pMapItem & M);
    //@}

    /*! @name Logic operators */ //@{
    public:
    /*! Equality operator */
    pMapItem & operator=(const pMapItem & M);

    /*! Less operator */
    bool operator<(const pMapItem & E) const;

    /*! Not equal operator */
    bool operator!=(const pMapItem & E) const;
    //@}

    /*! @name Get-Set functions */ //@{
    public:
    void setLid(const UInt & Lid);
    void setGid(const UInt & Gid);
    void setPid(const UInt & Pid);
    void setBufLid(const UInt & BufLid);
    UInt & getLid();
    UInt & getGid();
    UInt & getPid();
    UInt & getBufLid();
    const UInt & getLid() const;
    const UInt & getGid() const;
    const UInt & getPid() const;
    const UInt & getBufLid() const;
    //@}

    /*! @name Other functions */ //@{
    public:
    /*! \c lid -> \c bufLid */
    void bufferLid();

    /*! \c bufLid -> \c lid */

```

```

void restoreLid();
/*@}

/*! @name Outstream operators */ //@{
public:
friend ostream & operator<<(ostream & f, const pMapItem & M);
/*@}
};

```

This class contains the *pid*, *gid* and *lid* identifiers described above and the *bufLid* variable which represents a buffer where to store the *lid* variable. In fact, the *lid* variable can be varied during some communication phases since some data can be transferred from a process to another. In those cases, it can be useful to keep trace of the original local *id*. To be more precise, *bufferLid()* copies the value of *lid* into *bufLid*, on the contrary, the function *restoreLid()* copies the value of *bufLid* into *lid*. Moreover, *pMapItem* can be transmitted across the communicator as any degree of freedom, in particular, since it has a fixed memory dimension, the transmission of this class can be optimized by specifying it in the list contained in the file *pMpiOptimization.h*. We also point out that each map item is associated to an *enum* flag; for instance the class *pMapItem* is associated to *pMapPlain*. The complete list of the enumerative flags is contained into the *morganaPmapsItems.h* file. To aid some data transmission operations, the *pMapItem* class can be ordered by using the *gid* value. The *pMapItemSharing* and *pMapItemSendRecv* classes are inherited from *pMapItem* and they share many aspects. Each class adds peculiar set of data: in the former case two logic flags are added while in the second case some information regarding the origin and the destination of data are added. Below we include the definition of the *pMapItemShare* class:

```

class pMapItemShare : public pMapItem
{
/*! @name Parallel support */ //@{
public:
friend class boost::serialization::access;

template<class ARK>
void serialize(ARK & ar, const unsigned int version);
/*@}

/*! @name Internal data */ //@{
public:
static const pMapItems parallelType = pMapShare;
bool shared, owned;
/*@}

/*! @name Constructors and equality operator */ //@{
public:
pMapItemShare();
pMapItemShare(const UInt & Lid, const UInt & Gid);
pMapItemShare(const UInt & Lid, const UInt & Gid,
               const bool & Shared, const bool & Owned);

pMapItemShare(const UInt & Lid, const UInt & Gid, const UInt & Pid,
               const bool & Shared, const bool & Owned);

pMapItemShare(const pMapItemShare & M);
pMapItemShare & operator=(const pMapItemShare & M);

```

```

//@}

/*! @name Set-Get functions */ //@{
public:
void setShared(const bool & Shared);
void setOwned(const bool & Owned);
bool & getShared();
bool & getOwned();
const bool & getShared() const;
const bool & getOwned() const;
//@}

/*! @name Outstream operators */ //@{
public:
friend ostream & operator<<(ostream & f, const pMapItemShare & M);
//@}
};

```

With respect to the *pMapItem* class, *pMapItemSharing* has a slightly larger memory size since it contains two more boolean variables, however these information can be used to optimize the data processing by coding some more efficient manipulation algorithms. We will detail this aspect in the subsequent sections.

Now we include the definition of the *pMapItemSendRecv* class:

```

class pMapItemSendRecv : public pMapItem
{
/*! @name Parallel support */ //@{
public:
friend class boost::serialization::access;

template<class ARK>
void serialize(ARK & ar, const unsigned int version);
//@}

/*! @name Internal data */ //@{
public:
static const pMapItems parallelType = pMapSendRecv;
UInt sid, rid;
//@}

/*! @name Constructors and equality operator */ //@{
public:
pMapItemSendRecv();
pMapItemSendRecv(const UInt & Lid, const UInt & Gid);
pMapItemSendRecv(const UInt & Lid, const UInt & Gid,
                  const UInt & Sid, const UInt & Rid);

pMapItemSendRecv(const pMapItemSendRecv & M);
pMapItemSendRecv & operator=(const pMapItemSendRecv & M);
//@}

/*! @name Set-Get functions */ //@{
public:
void setSid(const UInt & Sid);
void setRid(const UInt & Rid);
UInt & getSid();
UInt & getRid();

```



```

const UInt & getSid() const;
const UInt & getRid() const;
//@}

/*! @name Logic operators */ //@{
public:
bool operator<(const pMapItemSendRecv & E) const;
bool operator!=(const pMapItemSendRecv & E) const;
//@}

/*! @name Outstream operators */ //@{
public:
friend ostream & operator<<(ostream & f, const pMapItemSendRecv & M);
//@}
};

```

which, as stated before, has two more *ids* process i.e. the *sid* and *rid*.

5.2 Map

Let us now pass to the description of the *pMap* class that represents the container which holds all the information regarding the parallel distribution of a set of data. The interface of the class is the following:

```

template<typename ITEM> class pMap
{
/*! @name Parallel support */ //@{
public:
friend class boost::serialization::access;

template<class ARK>
void serialize(ARK & ar, const unsigned int version);
//@}

public:
/*! Items of the graph */
sVect<ITEM> items;

/*! @name Constructors and equality operator */ //@{
public:
/*! Constructor */
pMap();

/*! Constructor -> initializes to normal indexing
\param n length of the graph */
pMap(const UInt & n);

/*! Copy constructor */
pMap(const pMap & G);

/*! Equality operator */
pMap & operator=(const pMap & G);

/*! Data clean */
void clear();
//@}
};

```

The class is template with respect to the *ITEM* type which can be equal either to:

- *pMapItem*;
- *pMapItemSharing*.

The class can be initialized as empty or the user can provide the *n* parameter which defines the initial length of the map. Moreover, it has been designed to be similar to an *stl* vector so that we have implemented the *resize*, *reserve* and *push.back* methods as reported below:

```

    /*! @name Access functions */ //@{
public:
    /*! Resize */
    void resize(const UInt & n);

    /*! Size */
    UInt size() const;

    /*! Get - memory checked */
    ITEM & get(UInt i);

    /*! Get - memory checked */
    const ITEM & get(UInt i) const;

    /*! Get - NOT memory checked */
    inline ITEM & operator() (UInt const i);

    /*! Get - NOT memory checked */
    inline const ITEM & operator() (UInt const i) const;

    /*! Set */
    void set(const UInt & i, const ITEM & I);

    /*! Push Back */
    void push_back(const ITEM & I);

    /*! Memory allocation */
    void reserve(const UInt & n);
    //@}

```

The *bufferLids* and *restoreLids* functions apply the same methods to the elements contained in the map:

```

    /*! @name Other functions */ //@{
public:
    /*! For each row sets \c lid -> \c bufLid */
    void bufferLids();

    /*! For each row sets \c bufLid -> \c lid */
    void restoreLids();
    //@}

    /*! @name Outstream operators */ //@{
public:
    template<typename T>

```

```
friend ostream & operator<<(ostream & f, const pMap<T> & M);
//@}
};
```

operator << completes the class with a simple print to screen interface.

6 Maps manipulation

The versatility of parallel maps lies mainly into three manipulation classes:

- *pMapManip* is a serial manipulation class without any call to *mpi* methods;
- *pMapComm* is the parallel communication class. In most cases, in *morgana*, there are no single communication calls, rather some pre-defined communication patterns are applied;
- *pMapGlobalManip* is a global map manipulation class. The implementation of its methods depends upon the specific map item class used.

We will describe them one by one.

6.1 Serial map manipulation

The *pMapManip* class requires that the pointer to the map be considered:

```
template<typename ITEM> class pMapManip
{
  /*! @name Typedefs */ //@{
public:
  typedef pMap<ITEM> PMAP;
  //@}

  /*! @name Links */ //@{
public:
  Teuchos::RCP<PMAP> map;
  //@}

  /*! @name Internal data - finder */ //@{
public:
  bool mapLoaded;
  bool finderOk;
  set<ITEM> container;
  //@}

  /*! @name Constructors and set functions */ //@{
public:
  pMapManip();
  pMapManip(const Teuchos::RCP<PMAP> & Map);
  pMapManip(PMAP & Map);
  pMapManip(const pMapManip & Pmanip);
  pMapManip operator=(const pMapManip & Pmanip);
  void setMap(const Teuchos::RCP<PMAP> & Map);
```

```
void setMap(PMAP & Map);
//@}
```

The first block of functions

```
/*! @name Search functions */ //@{
public:
void buildFinder();
void resetFinder();
bool isItem(const ITEM & Item) const;
UInt getLidItem(const ITEM & Item) const;
//@}
```

implement a search-and-find algorithm such that given a *gid* identifier, it is possible to find the corresponding *lid*. The *buildFinder()* method creates a fast search association based on the *stl :: map* with a logarithmic cost. The *resetFinder()* clears the association, *isItem(const ITEM & Item)*, given a map item, returns a boolean variable that states whether that particular item is present or not. *getLidItem* returns the *lid* corresponding to a given *gid*. The *finderOk* variable takes track of the fact that the *buildFinder()* method has been called or not.

The subsequent series of functions

```
/*! @name Set-Get and indexing functions */ //@{
public:
/*! Set \c map to normal indexing */
void setNormalIndexing();
void setNormalIndexing(PMAP & inMap) const;
void setNormalIndexing(Teuchos::RCP<PMAP> & inMap) const;

/*! Check the normal indexing */
bool checkNormalIndexing();
bool checkNormalIndexing(const PMAP & inMap) const;
bool checkNormalIndexing(const Teuchos::RCP<const PMAP> & inMap) const;

/*! Re-order the position of the items in the vector using the \c lid data */
void setIndexing();
void setIndexing(PMAP & inMap) const;
void setIndexing(Teuchos::RCP<PMAP> & inMap) const;

/*! Returns the maximum \c gid of \c vect */
UInt getMaxGid() const;
UInt getMaxGid(const PMAP & inMap) const;
UInt getMaxGid(const Teuchos::RCP<const PMAP> & inMap) const;

/*! The graphs are equal? */
bool isEqual(const PMAP & mapB);
bool isEqual(const PMAP & mapA,
             const PMAP & mapB) const;

bool isEqual(const Teuchos::RCP<const PMAP> & mapA,
             Teuchos::RCP<const PMAP> & mapB) const;
//@}
```

is mainly a set of reordering and data extraction methods. Before passing to the individual description of each function we introduce the concept of **normal indexing**. In particular we call a map *normal* if the various elements in the map:

- form an ordered list of *lids* starting from one: the first element has *lid* = 1, the second one *lid* = 2 and so on;
- there are no holes (the map is contiguous): for instance a map with 5 elements has a maximum *lid* equal to 5.

The *checkNormalIndexing()* function checks whether these properties are satisfied. There are three different implementations: the first one checks the map stored in the link *Teuchos::RCP<PMAP> map*; the second one checks the map passed as a reference and the third one checks the map passed as a reference counting pointer *RCP*, see [1].

If the current map does not satisfy the *normal indexing* criteria then there are several methods that can fix the problem. The first one is the *setNormalIndexing()* method which considers the current position of the map items correct and it renumbers the *lids* accordingly from 1 to the maximum local number of elements in the map. The complementary function is *setIndexing()* which considers correct the *lid* indexes and reorders the items accordingly. **In this last case the map must be contiguous and the lid numbering must start from one.** There are some assert checks that control these characteristics.

The *getMaxGid()* method returns the maximum *gid* on the given process where the function is called and *isEqual* verifies (on the given process) whether two maps are equal or not. Now we pass to the third block of functions:

```

/*! @name Group functions */ //@{
public:
/*! Cut \c map into a number of segments such
    that none of them exceeds \c maxSize */
void segmentationSimple(sVect<PMAP> & targetMaps,
    const UInt & maxSize) const;

void segmentationSimple(sVect<PMAP> & targetMaps,
    const UInt & maxSize,
    const PMAP & inMap) const;

void segmentationSimple(sVect<PMAP> & targetMaps,
    const UInt & maxSize,
    const Teuchos::RCP<const PMAP> & inMap) const;

/*! Cut \c map into \c segments segments.
Items are assigned to a specific segment using the \c gid key.
The maximum of all the \c gid s on all the processors is passed with \c maxGid */
void segmentationNormal(sVect<PMAP> & targetMaps,
    const UInt & segments,
    const UInt & maxGid) const;

void segmentationNormal(sVect<PMAP> & targetMaps,
    const UInt & segments,
    const UInt & maxGid,
    const PMAP & inMap) const;

void segmentationNormal(sVect<PMAP> & targetMaps,
    const UInt & segments,
    const UInt & maxGid,
    const Teuchos::RCP<const PMAP> & inMap) const;

/*! Cut \c map into \c maxPid segments using the \c pid key */
void segmentationPid(sVect<PMAP> & targetMaps,

```

```

        const UInt & maxPid) const;

void segmentationPid(sVect<PMAP> & targetMaps,
                    const UInt & maxPid,
                    const PMAP & inMap) const;

void segmentationPid(sVect<PMAP> & targetMaps,
                    const UInt & maxPid,
                    const Teuchos::RCP<const PMAP> & inMap) const;

/*! Simple merge into \c map of the \c sourceMaps. \c map is not cleared */
void mergeSimple(sVect<PMAP> & sourceMaps);
void mergeSimple(sVect<PMAP> & sourceMaps, PMAP & inMap);
void mergeSimple(sVect<PMAP> & sourceMaps, Teuchos::RCP<PMAP> & inMap);

/*! Re-order the data using a multi-set using gid*/
void orderData();
void orderData(PMAP & inMap);
void orderData(Teuchos::RCP<PMAP> & inMap);

/*! Merge into \c map of the \c sourceMaps. The repeated \c gid s
   are eliminated so that there is only one entry for each \c gid.
   \c map is not cleared*/
void unionExclusive(sVect<PMAP> & sourceMaps);
void unionExclusive(sVect<PMAP> & sourceMaps, PMAP & inMap);
void unionExclusive(sVect<PMAP> & sourceMaps, Teuchos::RCP<PMAP> & inMap);
//@}

```

These methods perform some merging and segmentation operations. The *segmentationSimple* function divides a given map in a number of sub-maps such that each sub-map does not exceed the maximum memory size specified by the *mpiMaxSize* parameter contained in the *pMpiOptimization.h* file. In other words, this method is used in those cases where the single communication message is too large with respect to the expected hardware performance. We will detail this later on.

The *segmentationNormal* function is a little bit more involved and requires

pid	lid	gid	pid	lid	gid	pid	lid	gid
0	1	6	0	3	4	0	1	6
0	2	5	0	4	3	0	2	5
0	3	4						
0	4	3						

Table 2: From the left to the right: the original map and the two-sub maps satisfying the normal partition criteria.

the definition of the **normal segmentation** of a given map and it is a concept linked to the parallel distribution of a map. Let us proceed with an example and consider the maximum value of all the *gids* that we label *maxGid* and a map distribution on two processes, then we consider a partition of the map in two sub-maps such that all the *gids* less than $maxGid/2$ are included in the first sub-map and all the others in the second sub-map. This represents the

pid	lid	gid
0	1	6
1	2	5
0	3	4
1	4	3

pid	lid	gid
0	1	6
0	3	4

pid	lid	gid
1	2	5
1	4	3

Table 3: From the left to the right: the original map and the two sub-maps generated using the *pid* criterion.

normal segmentation and the *segmentationNormal* method provides a tool for such a partition on a single process. This algorithm will be useful for ordering purposes. An example of the normal segmentation process is included in Table 2. Let us now pass to the description of the *segmentationPid* method: in this case the map is segmented using the *pid* index as the criterion, an example is given in Table 3.

The *mergeSimple* function gets a vector and adds to it the internal map stored with the *Teuchos :: RCP < PMAP > map* link, the new map items are appended, therefore the pre-existing items are not modified or canceled. The *orderData* function re-orders the map using the *gids* as key, but contrary to the functions described before, it can handle some repetitions: therefore different map items having the same *gid* are conserved. To conclude, the *unionExclusive* function merges the *sourceMaps* instance to the internal map stored by the *Teuchos :: RCP < PMAP > map* link eliminating repetitions. Therefore different map items with the same *gid* are not repeated.

The *pMapManip* class, in most cases, is used by other interface classes such as the map communication interface *pMapComm*.

6.2 Map communication

The *pMapComm* class contains several send and receive methods. They can be roughly distinguished in two categories:

- point to point communications;
- pattern communications, i.e. the communication scheme is embedded in the function itself.

The methods of this class loosely resemble the ones usually implemented by *mpi* libraries but their implementation is slightly more involved. In general, communication phases in *Morgana* are conceived such that the single message size is maximized with the smallest number of communication instances called. This means that, in some cases, the size of messages can be quite large and, if this size is too large, the code can automatically split it into some sub-messages. We outline that this size check adds some communication overheads that can be overridden using the *-DNOCOMMBUFFER* compilation flag. In this latter case the user must be sure that the system can handle the maximum message size for the specific computational run. We stress that in almost all cases we have used the *-DNOCOMMBUFFER* flag and we disabled all the checks.

```

/*! Class for the communication of \c pMap */
template<typename ITEM> class pMapComm
{
/*! @name Typedefs */ //@{
public:
typedef pMap<ITEM> MAP;
//@}

/*! @name Internal data and comm-link */ //@{
public:
UInt maxSize;
bool commDevLoaded;
Teuchos::RCP<const communicator> commDev;
//@}

/*! @name Constructors */ //@{
public:
/*! Constructor */
pMapComm();

/*! Constructor */
pMapComm(const Teuchos::RCP<const communicator> & CommDev);

/*! Setting of the communication device */
void setCommunicator(const Teuchos::RCP<const communicator> & CommDev);
//@}

```

The constructor of the class defines an integer $maxSize = mpiMaxSize / sizeof(ITEM)$ that prescribes the maximum size of a single message. **As already stated, the $mpiMaxSize$ constant defines the maximum size in bytes of each message and this is defined in the $pMpiOptimization$ file.** As a consequence, the maximum size of a message of type $ITEM$ depends upon $sizeof(ITEM)$. If the message is larger than $maxSize$ the data to be communicated are split in some sub-messages and the piece of data is reconstructed by the receiving process. **Due to the necessity of reconstructing the message the $irecv$ functions are blocking.** This is a marked difference with respect to standard $irecv$ mpi functions that are not blocking and return before the full receive of the message. On the contrary, all send functions $isend$ are not blocking. These limitations make sense since the messages regarding the transfer of maps are, in general, very small. As we will see, for the communication of very large chunk of data some more optimized functions have been developed. The class stores internally a $Teuchos::RCP<const communicator> commDev$ link to the local communicator.

```

/*! @name Send and Recv - non-pending-comm */ //@{
public:
/*! Sending and Receiving
\param sid sending process
\param rid receiving process
\param sMap the data to send
\param rMap the data received (rMap is cleared)
*/
void sendRecv(const UInt & sid, const UInt & rid,
              const Teuchos::RCP<const MAP> & sMap,
              Teuchos::RCP<MAP> & rMap) const;

```



```

    /*! Sending and Receiving
    \param sid sending process
    \param rid receiving process
    \param sMap the data to send
    \param rMap the data received (rMap is cleared)
    */
    void sendRecv(const UInt & sid, const UInt & rid,
                  const MAP & sMap, MAP & rMap) const;

    /*! Sending
    \param sid sending process
    \param rid receiving process
    \param sMap the data to send
    */
    void send(const UInt & sid, const UInt & rid,
              const Teuchos::RCP<const MAP> & sMap) const;

    /*! Sending
    \param sid sending process
    \param rid receiving process
    \param sMap the data to send
    */
    void send(const UInt & sid, const UInt & rid,
              const MAP & sMap) const;

    /*! Not-sync sending
    \param sid sending process
    \param rid receiving process
    \param sMap the data to send
    \param reqs the sending status vector
    */
    void isend(const UInt & sid, const UInt & rid,
               const Teuchos::RCP<const MAP> & sMap,
               sVect<request> & reqs) const;

    /*! Not-sync sending
    \param sid sending process
    \param rid receiving process
    \param sMap the data to send
    \param reqs the sending status vector
    */
    void isend(const UInt & sid, const UInt & rid,
               const MAP & sMap, sVect<request> & reqs) const;

    /*! Receiving
    \param sid sending process
    \param rid receiving process
    \param rMap the data received (rMap is cleared)
    */
    void recv(const UInt & sid, const UInt & rid,
              Teuchos::RCP<MAP> & rMap) const;

    /*! Receiving
    \param sid sending process
    \param rid receiving process
    \param rMap the data received (rMap is cleared)
    */

```

```
void recv(const UInt & sid, const UInt & rid, MAP & rMap) const;
```

```
/*! Not-sync receiving
\param sid sending process
\param rid receiving process
\param rMap the data received (rMap is cleared)
*/
```

```
void irecv(const UInt & sid, const UInt & rid,
           Teuchos::RCP<MAP> & rMap) const;
```

```
/*! Not-sync receiving
\param sid sending process
\param rid receiving process
\param rMap the data received (rMap is cleared)
*/
```

```
void irecv(const UInt & sid, const UInt & rid, MAP & rMap) const;
```

Communication functions specify the sending id *sid* and the receiving id *rid* (we stress that these indexes, since they follow the standard *mpi* numbering, start from one). *sMap* specifies the map to be sent and *rMap* the folder where to place the received data. We point out that all the *rMap* is cleared before receiving data.

```
/*! Merging the received data to \c rMap
\param sid sending process
\param rid receiving process
\param sMap the data to send
\param rMap the data received (rMap is merged with the arriving data)
*/
```

```
void merge(const UInt & sid, const UInt & rid, const MAP & sMap, MAP & rMap) const;
```

```
/*! Merging the received data to \c rMap
\param sid sending process
\param rid receiving process
\param sMap the data to send
\param rMap the data received (rMap is merged with the arriving data)
*/
```

```
void merge(const UInt & sid, const UInt & rid, const Teuchos::RCP<const MAP> & sMap,
           Teuchos::RCP<MAP> & rMap) const;
```

```
//@}
```

The *merge* method, on the contrary, does not clear the *rMap* map but it merges incoming data from the *sid* process using the *merge* function of the *pMapManip* class.

```
/*! @name Pattern communications - non-pending-comm */ //@{
public:
```

```
/*! Normal data distribution (gid-based)
\param map contains the data to send, then is cleared and loaded with the new data.
The \c maxGid parameter can be used to force the vector to a different distribution
on the communicator
*/
```

```
void vectorNormal(Teuchos::RCP<MAP> & map, const UInt & maxGid = 0) const;
```

```
/*! Normal data distribution (gid-based)
\param map contains the data to send, then is cleared and loaded with the new data.
The \c maxGid parameter can be used to force the vector to a different distribution
on the communicator
```

```

*/
void vectorNormal(MAP & map, const UInt & maxGid = 0) const;

/*! Pid data distribution (pid-based)
\param map contains the data to send, then is cleared and loaded with the new data
*/
void vectorPid(Teuchos::RCP<MAP> & map) const;

/*! Pid data distribution (pid-based)
\param map contains the data to send, then is cleared and loaded with the new data
*/
void vectorPid(MAP & map) const;
//@}
};

```

Let' now pass to describe the pattern communication methods. The *vectorNormal* function, given a distributed map, performs a series of communications to achieve the **normal distribution** as described in Section 6.1. Therefore, given, for instance, a communicator with two processes, the first process will obtain the map items characterized by the *gids* ranging from 1 to *maxGid/2*. The second one will receive the ones spanning from *maxGid/2* to *maxGid*. Finally the *vectorPid* function interprets the index *pid* in all the map items as the destination of that particular map item. So every process sends the relevant data and collects all the map items sent by other processes. The *map* parameter passed to that function is cleared and the new map is stored there.

6.3 Parallel map manipulation

pMapGlobalManip performs some macro-manipulations at a global level, it can check the correctness of the global distribution of the map and provides some interfaces with respect to the corresponding *epetra* and *tpetra* maps embedded in the *trilinos* third party package. The class has two template specializations: one for *pMapItem* and one for *pMapItemShare*. We will describe the first specialization and then we'll highlight the differences of the second specialization with respect to the first one.

```

/*! Perform global manipulations, retrieve informations and checks the \c pMap.
   Specialized version for \c pMapItem */
template<> class pMapGlobalManip<pMapItem>
{
/*! @name Typedefs */ //@{
public:
typedef pMapItem ITEM;
typedef pMap<ITEM> MAP;

typedef int ORDINALTYPE;
typedef Tpetra::DefaultPlatform::DefaultPlatformType PLATFORM;
typedef Tpetra::DefaultPlatform::DefaultPlatformType::NodeType NODE;
typedef Tpetra::Map<ORDINALTYPE,ORDINALTYPE,NODE> TPETRA_MAP;
//@}

/*! @name Internal data and links */ //@{
public:
bool commDevLoaded;

```

```

Teuchos::RCP<const communicator> commDev;
//@}

/*! @name Constructors and set functions */ //@{
public:
pMapGlobalManip();
pMapGlobalManip(const Teuchos::RCP<const communicator> & CommDev);
pMapGlobalManip(const communicator & CommDev);
void setCommunicator(const Teuchos::RCP<const communicator> & CommDev);
void setCommunicator(const communicator & CommDev);
//@}

```

This class defines, with the *typedef* command, some types related to the *tpetra* maps such as the fact we are using the *int* type, *ORDINALTYPE*, to define the indexes and *NODE*, i.e. the type of parallel distribution nodes. *pMapGlobalManip* stores also a *Teuchos :: RCP < constcommunicator > commDev* link to the local communicator. Let us now pass to analyze methods one by one:

```

/*! @name Data retriving functions */ //@{
public:
/*! Returns the maximum \c gid on all the processors */
UInt sizeG(const Teuchos::RCP<const MAP> & Map) const;

/*! Returns the maximum \c gid on all the processors */
UInt sizeG(const MAP & Map) const;
//@}

```

These functions returns the maximum *gid* of all the map items across the communicator.

```

/*! @name Global manipulations */ //@{
public:
/*! Checks that there is a consecutive numeration
of the Gids and that all the gids are positive */
bool check(const Teuchos::RCP<const MAP> & Map) const;

/*! Checks that there is a consecutive numeration
of the Gids and that all the gids are positive */
bool check(const MAP & Map) const;

/*! Modify the map eliminating all the repetitions */
void destroyOverlap(Teuchos::RCP<MAP> & Map) const;

/*! Modify the map eliminating all the repetitions */
void destroyOverlap(MAP & Map) const;

```

The method *check* verifies that:

- all the indexes are contiguous;
- the first index start with one.

The *destroyOverlap* function detects repeated map items with the same *gid* across the communicator and destroys these common copies such that only one map item for each *gid* is present. The repetition of map items across the communicator is also called: *overlap*. The algorithm applied is the following:

- map is transformed to achieve the *normal distribution*, therefore the map items having the same *gid* are now on the same process;
- every process eliminates the map items having the same *gid*. Only the one with the smallest *pid* survives;
- map items are transmitted back to the original processes;
- the *lid* identifiers are updated.

There are also several functions that provide an import and export interface with the maps of *Epetra* and *Tpetra*:

```

/! Export the map to an \c Epetra_Map
\param Map the input map
\param epetraMap the output map
\param base the base of the output map */
void exportEpetraMap(const Teuchos::RCP<const MAP> & Map,
Teuchos::RCP<Epetra_Map> & epetraMap,
Epetra_MpiComm & epetraComm,
const UInt & base) const;

/! Export the map to an \c Epetra_Map
\param Map the input map
\param epetraMap the output map
\param base the base of the output map */
void exportEpetraMap(const MAP & Map,
Teuchos::RCP<Epetra_Map> & epetraMap,
Epetra_MpiComm & epetraComm,
const UInt & base) const;

/! Export the map to an \c Tpetra_Map
\param Map the input map
\param tpetraMap the output map */
void exportTpetraMap(const Teuchos::RCP<const MAP> & Map,
Teuchos::RCP<const TPETRA_MAP> & tpetraMap) const;

/! Export the map to an \c Tpetra_Map
\param Map the input map
\param tpetraMap the output map */
void exportTpetraMap(const MAP & Map,
Teuchos::RCP<const TPETRA_MAP> & tpetraMap) const;

/! Import the map from an \c Epetra_Map
\param EpetraMap the input map
\param Map the output map*/
void importEpetraMap(const Teuchos::RCP<const Epetra_Map> & EpetraMap,
Teuchos::RCP<MAP> & Map);

/! Import the map from an \c Epetra_Map
\param EpetraMap the input map
\param Map the output map*/
void importEpetraMap(const Epetra_Map & EpetraMap,
Teuchos::RCP<MAP> & Map);

/! Import the map from an \c Tpetra_Map
\param TpetraMap the input map

```

```

\param Map the output map*/
void importTpetraMap(const Teuchos::RCP<const TPETRA_MAP> & TpetraMap,
Teuchos::RCP<MAP>                                     & Map);
//@}

```

There are some functions that support the communicator split in order to pass some data from the original communicator to a split one and vice versa. In the first case we say that the map is reduced to one sub-communicator. The *isActive* flag identifies the active sub communicator so that only the processes active in the new sub-communicator pass to *NewMap* their map items. The *pids* are then updated according to the new communicator.

```

/*! @name Communicator manipulations */ //@{
public:
/*! Copies the map on a smaller communicator,
all the pids not included in the new comm will be truncated */
void reduceCommunicator(const bool      & isActive,
const Teuchos::RCP<const communicator> & OldCommDev,
const Teuchos::RCP<const MAP>          & OldMap,
const Teuchos::RCP<const communicator> & NewCommDev,
Teuchos::RCP<MAP>                      & NewMap);

/*! Copies the map on a smaller communicator,
all the pids not included in the new comm will be truncated */
void reduceCommunicator(const bool & isActive,
const communicator & OldCommDev,
const MAP          & OldMap,
const communicator & NewCommDev,
MAP                & NewMap);

/*! Copies the map on a bigger communicator,
all the pids not included are void */
void expandCommunicator(const bool      & isActive,
const Teuchos::RCP<const communicator> & OldCommDev,
const Teuchos::RCP<const MAP>          & OldMap,
const Teuchos::RCP<const communicator> & NewCommDev,
Teuchos::RCP<MAP>                      & NewMap);

/*! Copies the map on a bigger communicator,
all the pids not included are void */
void expandCommunicator(const bool & isActive,
const communicator & OldCommDev,
const MAP          & OldMap,
const communicator & NewCommDev,
MAP                & NewMap);
//@}
};

```

In Figure 1 we have depicted an example of the reduction of a map from a reference communicator *oldComm* to a new communicator *newComm*. The original map is distributed on four processes and the map it is empty for *pid* = 2, 3. The original communicator is split in two sub-communicators, one of which is meant as active with *isActive* = *true*. The maps associated to *pid* = 0, 1 onto *oldComm* are copied to the new communicator just updating the *pid* flags since they, in this case, change: in particular $0 \rightarrow 1$ and $1 \rightarrow 0$. A similar scheme is used to expand a map from a smaller communicator to a larger one such as

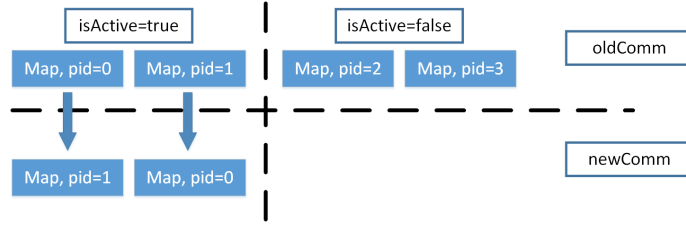


Figure 1: A scheme representing the reduction of a map from a communicator to another.

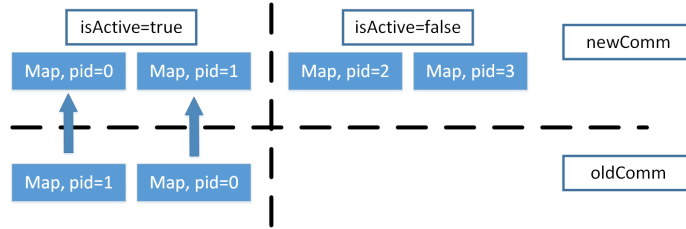


Figure 2: A scheme representing the expansion of a map from a communicator to another.

depicted in Figure 2. Let us now pass to describe the specialization of the class for *pMapItemShare*: in this case some parts of the internal implementation of the functions change and some more methods are present. In particular the *check* function checks all the instances already discussed for the *pMapItem* case but also assures that:

- there is only one *owned* map item across the communicator;
- all the map items with the same *gid* are shared.

Moreover, the implementation of the *destroyOverlap* method is simpler: it is a local function that eliminates in each process the local *shared* but not *owned* map items.

Some more functions are present, for instance the block:

```

/*! @name Data retriving functions */ //@{
public:
/*! Returns the maximum \c gid on all the processors */
UInt sizeG(const Teuchos::RCP<const MAP> & Map) const;

/*! Returns the maximum \c gid on all the processors */
UInt sizeG(const MAP & Map) const;

/*! Returns the number of locally shared items */
UInt sharedL(const Teuchos::RCP<const MAP> & Map) const;

/*! Returns the number of locally shared items */
UInt sharedL(const MAP & Map) const;

```

```

/*! Returns the number of globally shared items */
UInt sharedG(const Teuchos::RCP<const MAP> & Map) const;

/*! Returns the number of globally shared items */
UInt sharedG(const MAP & Map) const;

/*! Returns the number of locally owned items */
UInt ownedL(const Teuchos::RCP<const MAP> & Map) const;

/*! Returns the number of locally owned items */
UInt ownedL(const MAP & Map) const;
//@}

```

contains some functions that return the maximum *gid* across the communicator, the local and global number of *shared* map items and the local number of *owned* map items. Let us now pass to the last block of functions:

```

/*! Every owned and shared element updates all the other shared and not owned items.
This function creates the communication patten for this update */
void createSendRecvMap(const Teuchos::RCP<const MAP> & Map,
                      Teuchos::RCP<SENDRECV> & mapSend,
                      Teuchos::RCP<SENDRECV> & mapRecv) const;

/*! Every owned and shared element updates all the other shared and not owned items.
This function creates the communication patten for this update */
void createSendRecvMap(const MAP & Map,
                      SENDRECV & mapSend,
                      SENDRECV & mapRecv) const;

/*! Fix the owning-sharing, based on the gids.
Assumes that the \c gids are correct and updates the
owning-sharing structure. Multiple gids are fixed so
that only one has the ownership, the ones that are
already ok are retained */
void updateOwningSharing(Teuchos::RCP<MAP> & Map) const;

/*! Fix the owning-sharing, based on the gids.
Assumes that the \c gids are correct and updates the
owning-sharing structure. Multiple gids are fixed so
that only one has the ownership, the ones that are
already ok are retained */
void updateOwningSharing(MAP & Map) const;

/*! Fix the sharing, based on the \c gids.
Assumes that both the \c gids and the \c owned flag are correct and
updates the \c shared flag only */
void updateSharing(Teuchos::RCP<MAP> & Map) const;

/*! Fix the sharing, based on the \c gids.
Assumes that both the \c gids and the \c owned flag are correct and
updates the \c shared flag only */
void updateSharing(MAP & Map) const;

```

The *createSendRecvMap* function, given a *Map* map, creates a couple of maps (*mapSend* and *mapRecv*) such that each *shared* and *owned* map item generates a sequence of *pMapItemSendRecv* to connect it with other *shared* not *owned* map items. For instance considering the map included in Table 1

<i>pid</i>	<i>lid</i>	<i>gid</i>	<i>sid</i>	<i>rid</i>
0	3	3	0	1
1	2	4	1	0

Table 4: Send map.

<i>pid</i>	<i>lid</i>	<i>gid</i>	<i>sid</i>	<i>rid</i>
0	4	4	1	0
1	1	3	0	1

Table 5: Receiving map.

the corresponding *mapSend* map would be the one displayed in Table 4 and the *mapRecv* would be the one included in Table 5. This function is particularly useful and it is used to generate some involved communication patterns. We will see this functionality several times when we will pass to describe the *pVect* class. The *updateOwningSharing* function assumes that *gids* are correct and checks whether repeated *gids* of map items are associated to *shared = true* and there is only one *owned = true* element among a pool of shared map items. Finally the *updateSharing* function is similar but fixes only the *shared* flags, the *owned* flags must already be correct.

7 Parallel vectors

The *pVect* class is the main data container used in *morgana* and can be thought as a natural extension of the *sVect* (i.e. *std::vector*) to a parallel environment.

```

/*! Parallel vector data type, global and local id access support.
  Contains a finder device to find the global ids. */
template<typename DATA, typename MAP> class pVect
{
    /*! @name Parallel support */ //@{
public:
    typedef sVect<DATA>          CONTAINER_DATA;
    typedef pMap<MAP>            CONTAINER_MAP;
    typedef Teuchos::RCP<CONTAINER_DATA> RCP_CONTAINER_DATA;
    typedef Teuchos::RCP<CONTAINER_MAP> RCP_CONTAINER_MAP;
    typedef pMapManip<MAP>      MANIPULATOR;
    //@}

    /*! @name Parallel support */ //@{
public:
    friend class boost::serialization::access;

    template<class ARK>
    void serialize(ARK & ar, const unsigned int version);
    //@}

    /*! @name Internal data */ //@{

```

```

public:
Teuchos::RCP<CONTAINER_DATA> data;
Teuchos::RCP<CONTAINER_MAP> map;
Teuchos::RCP<MANIPULATOR> mapManip;
bool startupOk;
//@}

```

The class is template with respect to the type of data used and the type of parallel map. The parallel maps that, by now, can be used are *pMapItem* and *pMapItemShare* while the data type can be equal to any of the *dofs* types found in the package *morganaDofs*. The class contains:

- a serial vector *sVect* that actually contains all the data;
- a parallel map;
- a serial manipulator class: i.e. *pMapManip*.

The parallel map contains all the information regarding the parallel distribution of data and the serial manipulator is mainly used to find locally some specific data associated to a given *gid*. In fact, the piece of data in *sVect* is sorted by its *lid* flag and to access the same datum using the *gids* requires a sort algorithm. The *pVect* class can be serialized to send data to other processes in the communicator.

The class contains an internal system to handle the insertion of new data and update the search table contained in *pMapManip*. In principle, every new addition should cause an update, however this process could cause a significant computational burden therefore only some insert functions trigger the update. Some other functions only register, with an internal flag, that the piece of data has changed and it requires a subsequent, manual, update of the search table. The updated status of that table is encoded in the flag *startupOk*. Any access to data that requires to use the search table when *startupOk = false* will cause an *assert* error.

```

/! @name Constructors and operators */ //@{
public:
/! Constructor - \c startupOk = false */
pVect();

/! Constructor - \c startupOk = false */
pVect(const UInt & n);

/! Constructor - \c startupOk = true,
apply normal indexing, finder build */
pVect(const pVect & Vect);

/! Constructor - \c startupOk = true,
apply normal indexing, finder build */
pVect(const CONTAINER_MAP & Map,
      const CONTAINER_DATA & Data);

/! Constructor - \c startupOk = true,
apply normal indexing, finder build */
pVect(const Teuchos::RCP<CONTAINER_MAP> & Map,
      const Teuchos::RCP<CONTAINER_DATA> & Data);

```

```

/*! Equality operator - \c startupOk = true,
apply normal indexing, finder build */
pVect operator=(const pVect & Vect);
//@}

```

There is a number of constructors depending on what kind of data is available at the time of creation of the *pVect*. If the data and the parallel map are available, then the find table is updated just after the creation of the class. Then we pass to the block of data insertion and data management:

```

/*! @name Set and Add functions */ //@{
public:
/*! Set data - \c startupOk = true, apply normal indexing, finder build */
void setData(const CONTAINER_MAP & Map, const CONTAINER_DATA & Data);

/*! Add data block - \c startupOk = true, apply normal indexing, finder build */
void setData(const Teuchos::RCP<CONTAINER_MAP> & Map,
             const Teuchos::RCP<CONTAINER_DATA> & Data);

/*! Add data block - \c startupOk = true, apply normal indexing, finder re-building */
void addData(const CONTAINER_MAP & Map,
             const CONTAINER_DATA & Data);

/*! Add data block - \c startupOk = true, apply normal indexing, finder re-building */
void addData(const Teuchos::RCP<CONTAINER_MAP> & Map,
             const Teuchos::RCP<CONTAINER_DATA> & Data);

/*! Cancel all the finder informations, \c startupOk = false */
void resetFinder();

/*! Re-build the finder, enforce normal indexing, \c startupOk = true */
void updateFinder();

/*! Buffering of all the lids */
void bufferLids();

/*! Restore the buffered lids */
void restoreLids();

/*! Clearing of all the informations - \c startupOk = false */
void clear();
//@}

```

The *setData* and *addData* functions are meant to add large chunks of data, therefore they force the update of the find table and set *startupOk = true*. The *resetFinder()* method deletes the find table and *updateFinder()* forces its update. *bufferLids()* and *restoreLids()* call the same function for each map item present in *Teuchos :: RCP < CONTAINER_MAP > map*. Finally *clear()* deletes all the data.

```

/*! @name Get - Set block functions */ //@{
public:
CONTAINER_DATA      & getDataRef();
CONTAINER_MAP       & getMapRef();
RCP_CONTAINER_DATA  & getDataRcp();
RCP_CONTAINER_MAP   & getMapRcp();
const RCP_CONTAINER_DATA & getDataRcp() const;
const RCP_CONTAINER_MAP & getMapRcp() const;

```

```

const CONTAINER_DATA      & getDataRef() const;
const CONTAINER_MAP      & getMapRef() const;
void setMap(const CONTAINER_MAP & RowMap);
void setData(const CONTAINER_DATA & Data);
//@}

```

Several access functions are available to download the internal structures of the class. The last two set functions, i.e. *setMap* and *setData*, modify the corresponding structures without updating the search algorithm and without checking the coherence between the *map* and the *data* vector i.e. their length must be equal.

```

/#!/ @name Get pedantic functions */ //@{
public:
UInt sizeL();
DATA & getL(const UInt & lid);
const DATA & getL(const UInt & lid) const;
DATA & getG(const UInt & gid);
const DATA & getG(const UInt & gid) const;
bool isG(const UInt & gid) const;
MAP & getMapL(const UInt & lid);
const MAP & getMapL(const UInt & lid) const;
MAP & getMapG(const UInt & gid);
const MAP & getMapG(const UInt & gid) const;
DATA & getDataL(const UInt & lid);
const DATA & getDataL(const UInt & lid) const;
DATA & getDataG(const UInt & gid);
const DATA & getDataG(const UInt & gid) const;
//@}

```

The single map items and the data can be accessed also using the *lid* or *gid* keys. If the *gid* key is used the code checks with an *assert* command whether the search table in *mapManip* is updated by verifying that *startupOk = true*. Obviously accessing the data using the *lid* key is more performance effective since no search table must be queried.

```

/#!/ @name Get interface functions */ //@{
public:
UInt size() const;
DATA & get(const UInt & lid);
const DATA & get(const UInt & lid) const;
DATA & operator() (const UInt & lid);
const DATA & operator() (const UInt & lid) const;
//@}

```

This last block of functions has been implemented to make the *pVect* more compliant with a standard *sVect* by accessing its size or data as if it were an *std::vector*.

```

/#!/ @name PushBack functions */ //@{
public:
/#!/ Add one item
\param mapItem map for the item
\param dataItem the datum to be added
\param updateFinder if true the gid finder is updated,
on the contrary is necessary to use the \c updateFinder
method before accessing to data (\c startupOk = false) */
void push_back(const MAP & mapItem, const DATA & dataItem,
               bool updateFinder = false);

```

```

    /*! Add one item
    \param mapItem map for the item
    \param dataItem the datum to be added
    \param updateFinder if true the gid finder is updated,
    on the contrary is necessary to use the \c updateFinder
    method before accessing to data (\c startupOk = false) */
    void push_back(const DATA & dataItem, const MAP & mapItem,
                  bool updateFinder = false);

    /*! Resize */
    void resize(const UInt & n);

    /*! Memory allocation */
    void reserve(const UInt & n);
    //@}

```

Also these last methods are very similar to the ones implemented for the *std::vector* class however the *push_back* one is slightly different. If the method is called using only one parameter, then the search table is not updated and this would trigger: *startupOK = false*. Otherwise, the user may specify explicitly, with the second parameter, whether the search table must be updated or not.

8 Manipulators of parallel vectors

8.1 Serial manipulation of parallel vectors

The *pVect* class has its own serial manipulation class i.e. *pVectManip*. Its functionality is similar to the one we have seen for the *map* class, however, some details are different as we will outline.

```

    /*! Serial manipulation class for \c pVect */
    template<typename DATA, typename MAPITEM> class pVectManip
    {
    /*! @name Typedefs */ //@{
    public:
    typedef pMap<pMapItemSendRecv> SENDRECV;
    typedef pVect<DATA,MAPITEM> PVECT;
    typedef sVect<DATA> DATAVECT;
    typedef pMap<MAPITEM> PMAP;
    //@}

    /*! @name Internal data - finder */ //@{
    public:
    Teuchos::RCP<PVECT> vect;
    bool vectLoaded;
    //@}

    /*! @name Constructors and set functions */ //@{
    public:
    pVectManip();
    pVectManip(const Teuchos::RCP<PVECT> & Vect);
    pVectManip(PVECT & Vect);
    void setVect(const Teuchos::RCP<PVECT> & Vect);
    void setVect(PVECT & Vect);
    }

```

```
//@}
```

The class stores internally a *Teuchos* :: *RCP* < *PVECT* > *vect* link to the parallel vector to be manipulated. All the methods described below give the possibility to work on a specific *pVect* passed to the function or on the internally stored link *vect*.

```

/*! @name Set-Get functions */ //@{
public:
/*! Set \c vect to normal indexing */
void setNormalIndexing();
void setNormalIndexing(PVECT & inVect) const;
void setNormalIndexing(Teuchos::RCP<PVECT> & inVect) const;

/*! Re-order the position of the items in the vector using the \c lid data */
void setIndexing();
void setIndexing(PVECT & inVect) const;
void setIndexing(Teuchos::RCP<PVECT> & inVect) const;

/*! Returns the maximum \c gid of \c vect */
UInt getMaxGid();
UInt getMaxGid(const PVECT & inVect) const;
UInt getMaxGid(const Teuchos::RCP<const PVECT> & inVect) const;
//@}

```

The first method we are going to describe is *setNormalIndexing()* which calls the same method described for *pMapManip*. The *setIndexing()* method changes the position of the map items and the data in *pVect* using the *lid* index as the correct position of the data inside the vector. *getMaxGid()* returns the maximum *gid* to the local process.

```

/*! @name Group functions */ //@{
public:
/*! Cut \c vect into \c maxSize segments */
void segmentationSimple(sVect<PVECT> & targetVecs,
                      const UInt & maxSize) const;

void segmentationSimple(sVect<PVECT> & targetVecs,
                      const UInt & maxSize,
                      const PVECT & inVect) const;

void segmentationSimple(sVect<PVECT> & targetVecs,
                      const UInt & maxSize,
                      const Teuchos::RCP<const PVECT> & inVect) const;

/*! Cut \c vect into \c segments segments.
Items are assigned to a specific segment using the \c gid key.
The maximum of all the \c gid s on all the processors
is passed with \c maxGid */
void segmentationNormal(sVect<PVECT> & targetVecs,
                      const UInt & segments,
                      const UInt & maxGid) const;

void segmentationNormal(sVect<PVECT> & targetVecs,
                      const UInt & segments,
                      const UInt & maxGid,
                      const PVECT & inVect) const;

```

```

void segmentationNormal(sVect<PVECT> & targetVects,
                        const UInt    & segments,
                        const UInt    & maxPid,
                        const Teuchos::RCP<const PVECT> & inVect) const;

```

The *segmentationSimple* function is equivalent of what we have seen for *maps*. It is used to split a parallel vector into some sub-vectors such that each sub-vector does not exceed a given maximum dimension determined by, possibly present, communication constraints. The *segmentationNormal* method, given a parallel vector, generates a number of *sVect* < *PVECT* > *targetVects* sub-vectors applying the *normal* distribution described in Section 8.1. The data are distributed accordingly to map items.

```

/#!/ Cut \c vect into \c maxPid segments using the \c pid key */
void segmentationPid(sVect<PVECT> & targetVects,
                    const UInt    & maxPid) const;

void segmentationPid(sVect<PVECT> & targetVects,
                    const UInt    & maxPid,
                    const PVECT  & inVect) const;

void segmentationPid(sVect<PVECT> & targetVects,
                    const UInt    & maxPid,
                    const Teuchos::RCP<const PVECT> & inVect) const;

/#!/ Cut \c vect into \c maxPid segments
using the sending-id (sid) in \c mapSend as a key */
void segmentationMap(sVect<PVECT> & targetVects,
                    const SENDRECV & mapSend,
                    const UInt    & maxPid) const;

void segmentationMap(sVect<PVECT> & targetVects,
                    const SENDRECV & mapSend,
                    const UInt    & maxPid,
                    const PVECT  & inVect) const;

void segmentationMap(sVect<PVECT> & targetVects,
                    const SENDRECV & mapSend,
                    const UInt    & maxPid,
                    const Teuchos::RCP<const PVECT> & inVect) const;

/#!/ Cut \c vect into \c maxPid segments using the sending-id (sid)
in \c mapSend as a key,
* recursive, high performance, version \c targetVects
should be already allocated and the map must not change */
void segmentationMapR(sVect<PVECT> & targetVects,
                    const SENDRECV & mapSend,
                    const UInt    & maxPid) const;

void segmentationMapR(sVect<PVECT> & targetVects,
                    const SENDRECV & mapSend,
                    const UInt    & maxPid,
                    const PVECT  & inVect) const;

void segmentationMapR(sVect<PVECT> & targetVects,
                    const SENDRECV & mapSend,
                    const UInt    & maxPid,
                    const Teuchos::RCP<const PVECT> & inVect) const;

```

Another function, very similar to the one described for the maps, is *segmentationPid* that divides the vector and the map according to the *pid* indexes. Let us now pass to describe the *segmentationMap* method: this function receives a *pVect* and a map of type *SENDRECV* that specifies the elements of the vector to be sent and to which process through a list of proper *sid*. The *sVect* < *PVECT* > *targetVects* list of parallel vectors has a length equal to *maxPid*, i.e. the *size* of the communicator. The *i*-th sub-vector in *targetVects* contains the elements to be sent to the *i*-th process. This function is, in many cases, used in conjunction with the *createSendRecvMap* function of *pMapGlobalManip* which creates the *sendMap*.

The *segmentationMapR* function represents a slight variant of the former method and it is optimized for all the cases where the piece of data is sent in a repetitive manner. We refer, for instance, to a time-advancing method that, at every step, updates some data. In this case the function assumes that the list of *targetVects* sub-vectors is already correctly sized and it updates only the data, not the corresponding map items. Therefore, the sub-vectors must already have a correct map that never changes.

```

/#!/ Cut \c vect into \c maxPid segments using the \c data themselves as a key.
The maximum and minimum data \c maxData and \c minData should be provided */
void segmentationData(sVect<PVECT> & targetVects,
                    const UInt & maxPid,
                    const DATA & minData,
                    const DATA & maxData) const;

void segmentationData(sVect<PVECT> & targetVects,
                    const UInt & maxPid,
                    const DATA & minData,
                    const DATA & maxData,
                    const PVECT & inVect) const;

void segmentationData(sVect<PVECT> & targetVects,
                    const UInt & maxPid,
                    const DATA & minData,
                    const DATA & maxData,
                    const Teuchos::RCP<const PVECT> & inVect) const;

/#!/ Re-order the data using a multi-set, \c key = data */
void orderData();
void orderData(PVECT & inVect);
void orderData(Teuchos::RCP<PVECT> & inVect);

/#!/ Re-order the data using a multi-set, \c key = gid */
void orderGid();
void orderGid(PVECT & inVect);
void orderGid(Teuchos::RCP<PVECT> & inVect);

```

The *segmentationData* method is one of the most involved methods. It can split a *pVect* in a list of sub-vectors *sVect* < *PVECT* > *targetVects* using data as key for the sorting process. Every dof inside *Morgana* can be sorted, so the process is well defined: in particular dofs must be considered in the *traitsSegmentationUtility.hpp* file we are going to describe shortly. This latter file provides a utility that, given a maximum and a minimum value of a given list of dofs, cuts it in a given number of sub-vectors. This is a particularly useful functionality, once implemented in a parallel environment, since the same dofs are sent to the same process. The *segmentationData* function loads the

maximum and minimum global values of a global distributed $pVect$ and cuts it in $maxPid$ sub-vectors called $targetVects$. Let us review an example: we consider two processors each having its own list as pointed out in Table 6.

The $dataSegmentationUtility(const UInt n, const DATA minData, const$

pid	0	1
	(1,0,0)	(1, 0, 0)
	(4,0,0)	(0, 0, 0)

Table 6: Distribution of a small list of points using two processes: the original list.

$DATA maxData)$ method is then called using the following parameters:

- $n = 2$;
- $minData = (0, 0, 0)$;
- $maxData = (4, 0, 0)$.

and generates an $orderMap$ vector having three elements $m_1 = (0, 0, 0)$, $m_2 = (2, 0, 0)$ and $m_3 = (4, 0, 0)$. The first element m_1 corresponds to the smallest point, the third one m_3 to the greatest and m_2 corresponds to a pivot middle value. The values between m_1 and m_2 are sent to the first process while the values between m_2 and m_3 are sent to the second one. Therefore the first process, $pid = 0$, sets $inVect(1) = \{(1, 0, 0)\}$ and $inVect(2) = \{(4, 0, 0)\}$ while the second process, $pid = 1$, sets $inVect(1) = \{(0, 0, 0), (1, 0, 0)\}$ and $inVect(2) = \{\}$. For completeness, we anticipate that the $segmentationData$ function used in conjunction with a communication process would lead to the data partitioning depicted in Table 7. Let us now pass to the $orderData()$ function: it re-orders

pid	0	1
	(0, 0, 0)	(4, 0, 0)
	(1, 0, 0)	-
	(1, 0, 0)	-

Table 7: Distribution of a small list of points using two processes: the modified list.

the data - map elements pairs by using data as keys. A similar method is $orderGid()$ which re-orders data using the gid indexes as keys.

```

/*! Simple merge into \c vect of the \c sourceMaps.
\c vect is not cleared */
void mergeSimple(sVect<PVECT> & sourceVects);

void mergeSimple(sVect<PVECT> & sourceVects,
                 PVECT      & outVect);

void mergeSimple(sVect<PVECT>      & sourceVects,
                 Teuchos::RCP<PVECT> & outVect);

/*! Simple merge into \c vect of the \c sourceMaps.
```

```

\c vect is not cleared Recursive version the map
is not changed and the outVect must have already
been properly sized */
void mergeSimpleR(sVect<PVECT> & sourceVects);

void mergeSimpleR(sVect<PVECT> & sourceVects,
                  PVECT      & outVect);

void mergeSimpleR(sVect<PVECT>      & sourceVects,
                  Teuchos::RCP<PVECT> & outVect);

```

The *mergeSimple* method simply adds *sourceVects* to the current *vect* just adding data and map items to the current ones. The *mergeSimpleR* method is slightly different: it assumes that the vector has the same size of the combined sum of the sizes of the *sourceVects*, then it copies the data of the sub-vectors into *vect* beginning with the first *sourceVects*. The map items are not modified. This version of the method is used in the *pVectComm* class to aid the communications in cases where the same set of data is sent several times.

```

/*! Merge into \c vect of the \c sourceMaps.
The repeated \c gid s are eliminated so that there
is only one entry for each \c gid.
\c vect is cleared*/
void unionExclusive(sVect<PVECT> & sourceVects);

void unionExclusive(sVect<PVECT> & sourceVects,
                    PVECT      & outVect);

void unionExclusive(sVect<PVECT>      & sourceVects,
                    Teuchos::RCP<PVECT> & outVect);

/*! Given a vector eliminates the elements with the same \c gid */
void unionExclusive();
void unionExclusive(PVECT & outVect);
void unionExclusive(Teuchos::RCP<PVECT> & outVect);
//@}

```

Finally, *unionExclusive* creates, given a set of *sourceVects* a vector without any repetition of *gids*. It merges the maps of the various sub-vectors and applies the *unionExclusive* method of *pMapManip*. Then, given the new map, it creates a new vector matching the corresponding data linked to the new map. The version *unionExclusive* () just eliminates repeated *gid* elements in the map and the corresponding data.

8.2 Parallel vector communication

The *pVectComm* class is similar to the *pMapComm* class used for *pMap*, however, *pVectComm* is highly critical for the performances of the code and, due to this reason, is much more involved and complex. We start its description recalling just a few information regarding the *mpi* message passing scheme. For instance, this is based on a set of point-to-point message passing algorithms that identifies each message with a sending code *sid* and a message *tag*. Messages that have different couples of *sid/tag* are, in fact, different messages. The same couple can be re-used in the code but one must wait the corresponding message

to be received. If the couple *sid/tag* is used for a message while the same couple has been used previously and not yet received this will lead to an exception. We call this event a conflicting communication overlap.

Another concept we would like to recall is that the *mpi* methods can be categorized as *blocking* or *non – blocking* and this concept is very interesting when used for receiving functions. The *blocking* functions do not return until the data is received and made available on the output buffer. On the other hand, the *non – blocking* functions return immediately and a proper flag is given to inquire when the message has been properly received. The time between the call of the receive function and the actual gathering of data can be used to perform some computing operations thus masking, in part, the cost of passing data around the communicator. In the following we will describe how these concepts are implemented in this code.

The parallel communication system of *Morgana* is rather involved, in particular the methods of *pVectComm* can be classified in several different ways according to their characteristics. In particular one possible way of grouping the functions is:

- single point to point communications (P2P-COMM): they involve a single communication from a process *sid* to a receiving process *rid*;
- mapped communications (MAP-COMM): they involve a series of communications based on a *pMapSendRecv* communication map. Each data exchange between two process is managed by a single communication but, potentially, each process may communicate with all the others;
- pattern communications (PTT-COMM): the scheme of communications is implicitly defined by the scope of the function and this is opaque to the user. Also in this case only one message is passed between two processes but, potentially, these are all-to-all communications.

Another criterion is:

- pending communications: these functions are of the *non – blocking* type;
- non-pending communications: these functions are of the *blocking* type;
- recursive communications: are a type of pending methods further optimized for all the cases where a recursive sending of data is required. In this case the data size and shape must not change.

It is not straightforward to allow the possibility to handle complex MAP-COMM using a non-pending implementation still guaranteeing that there is no conflicting communication overlap. Things may get even more complicated since big messages may be segmented and thus it is not only necessary to assure that the messages arrive but it is also necessary to recompose the original chunk of data. *Morgana* solves these problems organizing the *tags* into groups also known as *channels*. The size of a channel is defined in the *pMpiOptimization.hpp* file by the variable *mpiLayer* = 100 therefore, by now, each channel reserves 100 possible tags. **All the non-pending communications use the first channel therefore are implemented in such a way that they use tags up to 99.**

The current implementation uses tags up to 2 therefore we expect that there is ample margins also for future implementations of the code. **We stress that all non-pending communications are blocking therefore each single method closes the receive phase of the communication and, thus, it is impossible to incur in a conflicting communication overlap.** Pending communications (recursive communications are included) are slightly more complicated: if only one pending communication is used at a time (i.e. the lines of code between the send and the receive phase not contain any reference to other non-pending communications) no problem arises. If more than one non-pending communication is used at the same time each communication must use a different channel so that different *tags* are used for each individual communication phase. We will include some examples after having presented the various functions of the class.

We start by presenting some structures used for non-pending communications:

```

/*! Class for exchanging data for pending comm - Parallel Vector PendingS */
template<typename DATA, typename MAPITEM>
class pVectPendings
{
public:
    typedef pVect<DATA,MAPITEM> PVECT;

    public :
    sVect<request> reqs;
    sVect<PVECT>   bufVects;
};

/*! Class for exchanging data for recursive pending comm - Parallel Vector Recursives */
template<typename DATA, typename MAPITEM>
class pVectRecursives
{
public :
    sVect<request>      reqs;
    boost::mpi::content bufCont;
};

/*! Class for recursive pVect update */
template<typename DATA, typename MAPITEM>
class pVectUpdateRecursive
{
public:
    typedef pVect<DATA,MAPITEM>          PVECT;
    typedef pVectRecursives<DATA,MAPITEM> PVRs;

    public:
    PVECT      rVect;
    sVect<PVECT> sendSegments;
    sVect<PVECT> recvSegments;
    sVect<PVRs>  sendPvrs;
    sVect<PVRs>  recvPvrs;
};

```

The *pVectPendings* class is used to check the end of a receive phase and to reconstruct the data segments to form a unique *pVect*. In fact *sVect* < *request* > *reqs* is the vector containing a *boost* structure that can be used to detect the end of the receive phase and *sVect* < *PVECT* > *bufVects* are the data segments. The *pVectRecursives* class is used to keep track of a single recursive communication and contains a vector of *boost* structures called *reqs*

and the type `boost::mpi::content bufCont` which stores the general structure and size of the data to be sent. If a `pVect` needs to be updated several times it is necessary to handle a number of single communications to different `pids`, in this case we used the `pVectUpdateRecursive` class which contains a number of `pVectRecursive` `< DATA, MAPITEM >` classes to keep track of the sent and received data in `sendPvrs` and `recvPvrs`. The chunk of data to be exchanged is stored in `sendSegments` and `recvSegments` vectors.

Let now pass to the main `pVectComm` class:

```
template<typename DATA, typename MAPITEM> class pVectComm
{
    /*! @name Typedefs */ //@{
public:
    typedef pVect<DATA,MAPITEM>          PVECT;
    typedef sVect<DATA>                  DATAVECT;
    typedef pMap<MAPITEM>                PMAP;
    typedef pMap<pMapItemSendRecv>       SENDRECV;
    typedef pVectPendings<DATA,MAPITEM>  PVPS;
    typedef pVectRecursive<DATA,MAPITEM> PVRS;
    //@}

    /*! @name Internal data and comm-link */ //@{
public:
    UInt maxSize;
    bool commDevLoaded;
    Teuchos::RCP<const communicator> commDev;
    //@}

    /*! @name Constructors */ //@{
public:
    pVectComm();
    pVectComm(const Teuchos::RCP<const communicator> & CommDev);
    void setCommunicator(const Teuchos::RCP<const communicator> & CommDev);
    //@}
}
```

it defines a sequence of types such as the `SENDRECV` send and receive maps and the structures used for pending and recursive communications such as `PVPS` and `PVRS`. The class contains a link `commDev` to the communicator. The following is a list of functions that are used for point-to point communications with a non-pending logic. To be more precise, the `isend` functions are non-blocking but both the `recv` and `irecv` functions are blocking. The `merge` function merges data to the current `rVect` while all other functions clear that buffer. The usage and internal logic are very similar to what described for the `pMapComm` class.

```
/*! @name Send and Recv - non-pending-comm */ //@{
public:
    void sendRecv(const UInt & sid,
                  const UInt & rid,
                  const Teuchos::RCP<const PVECT> & sendVect,
                  Teuchos::RCP<PVECT> & rVect) const;

    void sendRecv(const UInt & sid,
                  const UInt & rid,
                  const PVECT & sendVect,
                  PVECT & rVect) const;
}
```

```

void send(const UInt & sid,
         const UInt & rid,
         const Teuchos::RCP<const PVECT> & sendVect) const;

void send(const UInt & sid,
         const UInt & rid,
         const PVECT & sendVect) const;

void isend(const UInt & sid,
          const UInt & rid,
          const Teuchos::RCP<const PVECT> & sendVect,
          sVect<request> & reqs) const;

void isend(const UInt & sid,
          const UInt & rid,
          const PVECT & sendVect,
          sVect<request> & reqs) const;

void recv(const UInt & sid,
         const UInt & rid,
         Teuchos::RCP<PVECT> & rVect) const;

void recv(const UInt & sid,
         const UInt & rid,
         PVECT & rVect) const;

void irecv(const UInt & sid,
          const UInt & rid,
          Teuchos::RCP<PVECT> & rVect) const;

void irecv(const UInt & sid,
          const UInt & rid,
          PVECT & rVect) const;

void merge(const UInt & sid,
          const UInt & rid,
          const PVECT & sendVect,
          PVECT & rVect) const;

void merge(const UInt & sid,
          const UInt & rid,
          const Teuchos::RCP<const PVECT> & sendVect,
          Teuchos::RCP<PVECT> & rVect) const;
//@}

```

The following block describes nearly the same methods but with *pending* characteristics, in particular, every function of the previous block is split into two functions indicated by the letters *I* and *O*. The first is the one which starts the process and writes down some information in the *pvps* variable. The same variable must be passed to the *O* function which checks and returns only when the communication phase ends. In particular, the *recv* functions make available the received vector in the variable *rVect*. The variable *channel* identifies the communication channel i.e. it specifies implicitly the *tags* used in the communication.

```

/*! @name Send and Recv - pending-comm */ //@{
public:

```

```

void sendI(const UInt          & sid,
           const UInt          & rid,
           const Teuchos::RCP<PVECT> & sendVect,
           PVPS                & pvps,
           const UInt          & channel = 2) const;

void sendO(const UInt          & sid,
           const UInt          & rid,
           const Teuchos::RCP<PVECT> & sendVect,
           PVPS                & pvps,
           const UInt          & channel = 2) const;

void sendI(const UInt & sid,
           const UInt & rid,
           const PVECT & sendVect,
           PVPS & pvps,
           const UInt & channel = 2) const;

void sendO(const UInt & sid,
           const UInt & rid,
           const PVECT & sendVect,
           PVPS & pvps,
           const UInt & channel = 2) const;

void recvI(const UInt          & sid,
           const UInt          & rid,
           Teuchos::RCP<PVECT> & rVect,
           PVPS                & pvps,
           const UInt          & channel = 2) const;

void recvO(const UInt          & sid,
           const UInt          & rid,
           Teuchos::RCP<PVECT> & rVect,
           PVPS                & pvps,
           const UInt          & channel = 2) const;

void recvI(const UInt & sid,
           const UInt & rid,
           PVECT & rVect,
           PVPS & pvps,
           const UInt & channel = 2) const;

void recvO(const UInt & sid,
           const UInt & rid,
           PVECT & rVect,
           PVPS & pvps,
           const UInt & channel = 2) const;

void sendRecvI(const UInt          & sid,
               const UInt          & rid,
               const Teuchos::RCP<const PVECT> & sendVect,
               Teuchos::RCP<PVECT> & rVect,
               PVPS                & pvps,
               const UInt          & channel = 2) const;

void sendRecvO(const UInt          & sid,
               const UInt          & rid,

```

```

        const Teuchos::RCP<const PVECT> & sendVect,
            Teuchos::RCP<PVECT>          & rVect,
            PVPS                          & pvps,
        const UInt                        & channel = 2) const;

void sendRecvI(const UInt & sid,
               const UInt & rid,
               const PVECT & sendVect,
               PVECT & rVect,
               PVPS & pvps,
               const UInt & channel = 2) const;

void sendRecvO(const UInt & sid,
               const UInt & rid,
               const PVECT & sendVect,
               PVECT & rVect,
               PVPS & pvps,
               const UInt & channel = 2) const;

void mergeI(const UInt & sid,
            const UInt & rid,
            const PVECT & sendVect,
            PVECT & rVect,
            PVPS & pvps,
            const UInt & channel = 2) const;

void mergeO(const UInt & sid,
            const UInt & rid,
            const PVECT & sendVect,
            PVECT & rVect,
            PVPS & pvps,
            const UInt & channel = 2) const;

void mergeI(const UInt & sid,
            const UInt & rid,
            const Teuchos::RCP<const PVECT> & sendVect,
            Teuchos::RCP<PVECT> & rVect,
            PVPS & pvps,
            const UInt & channel = 2) const;

void mergeO(const UInt & sid,
            const UInt & rid,
            const Teuchos::RCP<const PVECT> & sendVect,
            Teuchos::RCP<PVECT> & rVect,
            PVPS & pvps,
            const UInt & channel = 2) const;
//0}

```

When the same *pVect* is passed several times between some processes, but the overall length and the *map* of the parallel vector do not change (only the numeric values inside the *data sVect* in the *pVect* change), then the communication scheme can be highly optimized. In this case, each task (i.e. send or receive a message) is divided into three methods identified by the *RR*, *RI* and *RO* flags. Three functions are linked by the *pvrs* variable (and in some cases also by *reqsRR*) which is used to exchange information among them. The methods characterized by the flag *RR* must be called only once and they setup the proper environment for the communication: for instance they exchange the

maps of the vector and its size. *RI* methods start the communication phase, either receive or send, and the *RO* phase completes it making all data available. Some computations can be put between the *RI* and the *RO* methods. We will discuss some examples in Section 10.

```

/! @name Send and Recv - recursive-comm */ //@{
public:
void sendRR(const UInt          & sid,
            const UInt          & rid,
            const Teuchos::RCP<PVECT> & sendVect,
            PVRs                & pvr,
            sVect<request>        & reqsRR,
            const UInt          & channel = 2) const;

void sendRR(const UInt          & sid,
            const UInt          & rid,
            const PVECT        & sendVect,
            PVRs                & pvr,
            sVect<request>        & reqsRR,
            const UInt          & channel = 2) const;

void recvRR(const UInt          & sid,
            const UInt          & rid,
            Teuchos::RCP<PVECT> & rVect,
            PVRs                & pvr,
            const UInt          & channel = 2) const;

void recvRR(const UInt & sid,
            const UInt & rid,
            PVECT & rVect,
            PVRs & pvr,
            const UInt & channel = 2) const;

void sendRI(const UInt          & sid,
            const UInt          & rid,
            const Teuchos::RCP<PVECT> & sendVect,
            PVRs                & pvr,
            const UInt          & channel = 2) const;

void sendRI(const UInt & sid,
            const UInt & rid,
            const PVECT & sendVect,
            PVRs & pvr,
            const UInt & channel = 2) const;

void sendRO(const UInt          & sid,
            const UInt          & rid,
            const Teuchos::RCP<PVECT> & sendVect,
            PVRs                & pvr,
            const UInt          & channel = 2) const;

void sendRO(const UInt & sid,
            const UInt & rid,
            const PVECT & sendVect,
            PVRs & pvr,
            const UInt & channel = 2) const;

void recvRI(const UInt          & sid,

```

```

        const UInt          & rid,
        Teuchos::RCP<PVECT> & rVect,
        PVRs                & pvr,
        const UInt          & channel = 2) const;

void recvRI(const UInt & sid,
            const UInt & rid,
            PVECT & rVect,
            PVRs & pvr,
            const UInt & channel = 2) const;

void recvRO(const UInt          & sid,
            const UInt          & rid,
            Teuchos::RCP<PVECT> & rVect,
            PVRs                & pvr,
            const UInt          & channel = 2) const;

void recvRO(const UInt & sid,
            const UInt & rid,
            PVECT & rVect,
            PVRs & pvr,
            const UInt & channel = 2) const;
//@}

```

Let us now pass to the mapped communications (MAP-COMM). We first start with the *non – pending* variants:

```

/*! @name Mapped communications - non-pending-comm */ //@{
public:
void sendRecv(const Teuchos::RCP<SENDRECV>    & mapSend,
              const Teuchos::RCP<const PVECT> & sendVect,
              Teuchos::RCP<PVECT>            & rVect) const;

void sendRecv(const SENDRECV & mapSend,
              const PVECT    & sendVect,
              PVECT          & rVect) const;

void sendRecv(const Teuchos::RCP<const SENDRECV> & mapSend,
              const Teuchos::RCP<const SENDRECV> & mapRecv,
              const Teuchos::RCP<const PVECT>    & sendVect,
              Teuchos::RCP<PVECT>                & rVect) const;

void sendRecv(const SENDRECV & mapSend,
              const SENDRECV & mapRecv,
              const PVECT    & sendVect,
              PVECT          & rVect) const;
//@}

```

The user provides a *sendVect* reference and a *mapSend* map which contain which data must be sent and to which process, as discussed for the corresponding functions of *pMapComm*. The data sent by all other processes are accumulated in the *rVect* vector. Some versions of the function may include a *mapRecv* map which contains what kind of data and from which process a given chunk of data should be expect. This extra information is used to optimize the communications since when no data must be exchanged no communication path is established. On the contrary, functions having only the *sendVect* map use an all-to-all approach sending void messages when no data must be exchanged between two specific processes.

Some *pending-comm* versions have been developed as well, they have two associated functions one indicated by the letter *I* and the other one by the letter *O*. The first one starts the sending, or receiving process while the second checks the completion of the process. If only one *pending-comm* communication is used at a time, then the user shall not provide any *channel* parameter. Otherwise the user must specify a different *channel* for each concurring *pending-comm* communication performed in parallel.

```

/! @name Mapped communications - pending-comm */ //@{
public:
void sendRecvI(const Teuchos::RCP<SENDRECV> & mapSend,
               const Teuchos::RCP<const PVECT> & sendVect,
               Teuchos::RCP<PVECT> & rVect,
               sVect<PVECT> & commSegments,
               sVect<PVPS> & sendPvps,
               sVect<PVPS> & recvPvps,
               const UInt & channel = 2) const;

void sendRecvO(const Teuchos::RCP<SENDRECV> & mapSend,
               const Teuchos::RCP<const PVECT> & sendVect,
               Teuchos::RCP<PVECT> & rVect,
               sVect<PVECT> & commSegments,
               sVect<PVPS> & sendPvps,
               sVect<PVPS> & recvPvps,
               const UInt & channel = 2) const;

void sendRecvI(const SENDRECV & mapSend,
               const PVECT & sendVect,
               PVECT & rVect,
               sVect<PVECT> & commSegments,
               sVect<PVPS> & sendPvps,
               sVect<PVPS> & recvPvps,
               const UInt & channel = 2) const;

void sendRecvO(const SENDRECV & mapSend,
               const PVECT & sendVect,
               PVECT & rVect,
               sVect<PVECT> & commSegments,
               sVect<PVPS> & sendPvps,
               sVect<PVPS> & recvPvps,
               const UInt & channel = 2) const;

void sendRecvI(const Teuchos::RCP<const SENDRECV> & mapSend,
               const Teuchos::RCP<const SENDRECV> & mapRecv,
               const Teuchos::RCP<const PVECT> & sendVect,
               Teuchos::RCP<PVECT> & rVect,
               sVect<PVECT> & commSegments,
               sVect<PVPS> & sendPvps,
               sVect<PVPS> & recvPvps,
               const UInt & channel = 2) const;

void sendRecvO(const Teuchos::RCP<const SENDRECV> & mapSend,
               const Teuchos::RCP<const SENDRECV> & mapRecv,
               const Teuchos::RCP<const PVECT> & sendVect,
               Teuchos::RCP<PVECT> & rVect,
               sVect<PVECT> & commSegments,
               sVect<PVPS> & sendPvps,
               sVect<PVPS> & recvPvps,

```

```

        const UInt                                     & channel = 2) const;

void sendRecvI(const SENDRECV      & mapSend,
               const SENDRECV      & mapRecv,
               const PVECT         & sendVect,
               PVECT               & rVect,
               sVect<PVECT>         & commSegments,
               sVect<PVPS>         & sendPvps,
               sVect<PVPS>         & recvPvps,
               const UInt          & channel = 2) const;

void sendRecv0(const SENDRECV      & mapSend,
               const SENDRECV      & mapRecv,
               const PVECT         & sendVect,
               PVECT               & rVect,
               sVect<PVECT>         & commSegments,
               sVect<PVPS>         & sendPvps,
               sVect<PVPS>         & recvPvps,
               const UInt          & channel = 2) const;

/*@}

```

The recursive version of these functions has three coordinated methods as described above. The *RR* method is used to initialize the communication process, the *RI* version is used to start each recursive communication and the *RO* method is used to check its conclusion. Some buffers, such as *sendSegments*, *recvSegments*, *sendPvrs* and *recvPvrs* must be passed among the three functions as they store some data regarding the communication status.

```

/*! @name Mapped communications - recursive-comm */ //@{
public:
void sendRecvRR(const Teuchos::RCP<SENDRECV>      & mapSend,
                const Teuchos::RCP<const PVECT>    & sendVect,
                Teuchos::RCP<PVECT>                & rVect,
                sVect<PVECT>                        & sendSegments,
                sVect<PVECT>                        & recvSegments,
                sVect<PVRs>                         & sendPvrs,
                sVect<PVRs>                         & recvPvrs,
                const UInt                          & channel = 2) const;

void sendRecvRR(const SENDRECV      & mapSend,
                const PVECT         & sendVect,
                PVECT               & rVect,
                sVect<PVECT>         & sendSegments,
                sVect<PVECT>         & recvSegments,
                sVect<PVRs>          & sendPvrs,
                sVect<PVRs>          & recvPvrs,
                const UInt          & channel = 2) const;

void sendRecvRI(const Teuchos::RCP<SENDRECV>      & mapSend,
                const Teuchos::RCP<const PVECT>    & sendVect,
                Teuchos::RCP<PVECT>                & rVect,
                sVect<PVECT>                        & sendSegments,
                sVect<PVECT>                        & recvSegments,
                sVect<PVRs>                         & sendPvrs,
                sVect<PVRs>                         & recvPvrs,
                const UInt                          & channel = 2) const;

void sendRecvRI(const SENDRECV      & mapSend,

```

```

const PVECT      & sendVect,
PVECT           & rVect,
sVect<PVECT>    & sendSegments,
sVect<PVECT>    & recvSegments,
sVect<PVRs>     & sendPvrs,
sVect<PVRs>     & recvPvrs,
const UInt      & channel = 2) const;

void sendRecvR0(const Teuchos::RCP<SENDRECV> & mapSend,
const Teuchos::RCP<const PVECT> & sendVect,
Teuchos::RCP<PVECT> & rVect,
sVect<PVECT> & sendSegments,
sVect<PVECT> & recvSegments,
sVect<PVRs> & sendPvrs,
sVect<PVRs> & recvPvrs,
const UInt & channel = 2) const;

void sendRecvR0(const SENDRECV & mapSend,
const PVECT & sendVect,
PVECT & rVect,
sVect<PVECT> & sendSegments,
sVect<PVECT> & recvSegments,
sVect<PVRs> & sendPvrs,
sVect<PVRs> & recvPvrs,
const UInt & channel = 2) const;

void sendRecvRR(const Teuchos::RCP<SENDRECV> & mapSend,
const Teuchos::RCP<SENDRECV> & mapRecv,
const Teuchos::RCP<const PVECT> & sendVect,
Teuchos::RCP<PVECT> & rVect,
sVect<PVECT> & sendSegments,
sVect<PVECT> & recvSegments,
sVect<PVRs> & sendPvrs,
sVect<PVRs> & recvPvrs,
const UInt & channel = 2) const;

void sendRecvRR(const SENDRECV & mapSend,
const SENDRECV & mapRecv,
const PVECT & sendVect,
PVECT & rVect,
sVect<PVECT> & sendSegments,
sVect<PVECT> & recvSegments,
sVect<PVRs> & sendPvrs,
sVect<PVRs> & recvPvrs,
const UInt & channel = 2) const;

void sendRecvRI(const Teuchos::RCP<SENDRECV> & mapSend,
const Teuchos::RCP<SENDRECV> & mapRecv,
const Teuchos::RCP<const PVECT> & sendVect,
Teuchos::RCP<PVECT> & rVect,
sVect<PVECT> & sendSegments,
sVect<PVECT> & recvSegments,
sVect<PVRs> & sendPvrs,
sVect<PVRs> & recvPvrs,
const UInt & channel = 2) const;

void sendRecvRI(const SENDRECV & mapSend,

```

```

        const SENDRECV      & mapRecv,
        const PVECT         & sendVect,
        PVECT              & rVect,
        sVect<PVECT>        & sendSegments,
        sVect<PVECT>        & recvSegments,
        sVect<PVRs>         & sendPvrs,
        sVect<PVRs>         & recvPvrs,
        const UInt          & channel = 2) const;

void sendRecvRO(const Teuchos::RCP<SENDRECV>      & mapSend,
               const Teuchos::RCP<SENDRECV>      & mapRecv,
               const Teuchos::RCP<const PVECT>    & sendVect,
               Teuchos::RCP<PVECT>              & rVect,
               sVect<PVECT>                    & sendSegments,
               sVect<PVECT>                    & recvSegments,
               sVect<PVRs>                    & sendPvrs,
               sVect<PVRs>                    & recvPvrs,
               const UInt                      & channel = 2) const;

void sendRecvRO(const SENDRECV      & mapSend,
               const SENDRECV      & mapRecv,
               const PVECT         & sendVect,
               PVECT              & rVect,
               sVect<PVECT>        & sendSegments,
               sVect<PVECT>        & recvSegments,
               sVect<PVRs>         & sendPvrs,
               sVect<PVRs>         & recvPvrs,
               const UInt          & channel = 2) const;
//@}

```

Let now pass to describe the *pattern* communication methods, they have been implemented only in the *non – pending* form since they are almost never used for recursive communications.

```

/*! @name Pattern communications - non-pending-comm */ //@{
public:
void vectorNormal(Teuchos::RCP<PVECT> & Vect,
                 const UInt & MaxGid = 0);

void vectorNormal(PVECT & Vect,
                 const UInt & MaxGid = 0);

void vectorPid(Teuchos::RCP<PVECT> & Vect);
void vectorPid(PVECT & Vect);

void vectorData(Teuchos::RCP<PVECT> & Vect);
void vectorData(PVECT & Vect);

void vectorData(Teuchos::RCP<PVECT> & Vect,
                const DATA & dataMin,
                const DATA & dataMax);

void vectorData(PVECT & Vect,
                const DATA & dataMin,
                const DATA & dataMax);
//@}

```

The *vectorNormal* method applies the same logic described for the *pMapComm* class: the data are divided according to the map values and distributed across

the communicator as described, for instance, in Table 2. The *vectorPid* function sends data to the processor outlined by the *pid* flag of the *map* of the *pVect Vect* variable. The *vectorData* detects maximum and minimum data values across the communicator and distributes them according to the partition defined by the *traitsSegmentationUtility* functions. The number of partitions corresponds to the maximum number of processes in the communicator. The maximum and minimum values can also be passed to the function by the user.

8.3 Parallel manipulation of parallel vectors

Let us now conclude the overview of the classes devoted to management of parallel vectors management by describing the *pVectGlobalManip* class. This has two specializations for *pMapItem* and *pMapItemSharing* map items. Let's start with the first specialization

```

/*! Performs global manipulations, retrieve informations and checks the \c pVect.
   Specialized version for \c pMapItem */
template<typename DATA> class pVectGlobalManip<DATA,pMapItem>
{
/*! @name Typedefs */ //@{
public:
typedef pVect<DATA,pMapItem>    PVECT;
typedef sVect<DATA>             DATAVECT;
typedef pMap<pMapItem>          PMAP;
typedef pMap<pMapItemSendRecv> SENDRECV;
//@}

/*! @name Internal data and links */ //@{
public:
bool commDevLoaded;
Teuchos::RCP<const communicator> commDev;
//@}

/*! @name Constructors and set functions */ //@{
public:
pVectGlobalManip();
pVectGlobalManip(const Teuchos::RCP<const communicator> & CommDev);
pVectGlobalManip(const communicator & CommDev);
void setCommunicator(const Teuchos::RCP<const communicator> & CommDev);
void setCommunicator(const communicator & CommDev);
//@}

/*! @name Data retrieving functions */ //@{
public:
/*! Returns the maximum \c gid on all the processors */
UInt sizeG(const Teuchos::RCP<const PVECT> & Vect) const;

/*! Returns the maximum \c gid on all the processors */
UInt sizeG(const PVECT & Vect) const;

```

The first part of the class includes the constructors, a link to the communicator and the redefinition of some types such as the *PVECT* parallel vector and send receive map type labeled *SENDRECV*.

```

    /*! Performs a check on the \c pMap contained into \c Vect, see \c pMapGlobalManip */
    bool check(const Teuchos::RCP<const PVECT> & Vect) const;

    /*! Performs a check on the \c pMap contained into \c Vect, see \c pMapGlobalManip */
    bool check(const PVECT & Vect) const;

    /*! Check that items with the same \c gid have the same data */
    bool checkConsistency(const Teuchos::RCP<const PVECT> & Vect) const;

    /*! Check that items with the same \c gid have the same data */
    bool checkConsistency(const PVECT & Vect) const;

    /*! Computes the minimum and maximum data */
    void dataMinMax(const Teuchos::RCP<const PVECT> & Vect,
        DATA & DataMin,
        DATA & DataMax);

    /*! Computes the minimum and maximum data */
    void dataMinMax(const PVECT & Vect,
        DATA & DataMin,
        DATA & DataMax);
    //@}

```

Regarding the second block of functions, the method *sizeG* returns the maximum *gid* in the map contained in the *pVect*, the *check* method applies the corresponding method, described in Section 6.3 for the *pMapGlobalManip* class, on the internal *map*. The *checkConsistency* function verifies that data having the same *gid* have the same value: for this reason data must be one of the *dofs* implemented in *Morgana* as all of them implement the \neq operator. Finally, the *dataMinMax* method returns the maximum and minimum *dof* in the global parallel vector. The following block contains some global numbering functions and map exchange functions.

```

    /*! @name Global manipulations */ //@{
    public:
    /*! Assigns a unique \c gid to data.
       If two data are equal the same gid is assigned to both.
       Data should implement the inequality and less operator. */
    void buildGlobalNumbering(Teuchos::RCP<PVECT> & Vect) const;

    /*! Assigns a unique \c gid to data.
       If two data are equal the same gid is assigned to both.
       Data should implement the inequality and less operator. */
    void buildGlobalNumbering(PVECT & Vect) const;

    /*! A higher performance method of \c buildGlobalNumbering.
       Local elements are marked with
       the \c isLocal vector and they are not broadcasted.*/
    void buildGlobalNumbering(Teuchos::RCP<PVECT> & Vect,
        const Teuchos::RCP<const sVect<bool> > & isLocal) const;

    /*! A higher performance method of \c buildGlobalNumbering.
       Local elements are marked with
       the \c isLocal vector and they are not broadcasted*/
    void buildGlobalNumbering(
        PVECT & Vect,
        const sVect<bool> & isLocal) const;

    /*! Creates a new vector whose new map is \c NewMap

```



```

    The data contained in \c Vect are communicated
    to satisfy the new map*/
    void changeMap(      Teuchos::RCP<PVECT>      & Vect,
                     const Teuchos::RCP<const PMAP> & NewMap) const;

    /*! Creates a new vector whose new map is \c NewMap
    The data contained in \c Vect are communicated
    to satisfy the new map*/
    void changeMap(      PVECT & Vect,
                     const PMAP & NewMap) const;

    /*! Distributes the vector using the \c pid logic.
    The repeated \c gid are eliminated */
    void vectorNormalExclusive(Teuchos::RCP<PVECT> & Vect);

    /*! Distributes the vector using the \c pid logic.
    The repeated \c gid are eliminated */
    void vectorNormalExclusive(PVECT & Vect);

    /*! Linear-epetra data distribution
    \param Vect contains the data to send,
    then is cleared and loaded with the new data
    */
    Epetra_Map vectorLinear(Teuchos::RCP<PVECT> & Vect);

    /*! Linear-epetra data distribution
    \param Vect contains the data to send,
    then is cleared and loaded with the new data
    */
    Epetra_Map vectorLinear(PVECT & Vect);
    //@}

```

Let us start with the *buildGlobalNumbering* function: this is used to build a global *gid* map (if the *pMapItemSharing* map item is used, then also the ownership structure is updated) based on the values of data. If two data have the same value then they will have the same *gid*. This function is particularly useful when dealing with geometrical items as two different processes may create the same geometrical entity (such as a point or a face) and it must be possible to recognize that the two items are in fact the same thing and they must be labeled with the same *gid*. The algorithm is based on the *vectorData* methods that send the same data to the same process so it is easier to recognize equal items and number them accordingly. Items are then sent to the original process. A higher performance version has been implemented as well i.e. *buildGlobalNumbering* (*Teuchos :: RCP < PVECT > Vect, const Teuchos :: RCP < constVect < bool >> isLocal*). The *isLocal* vector specifies which elements of the vector are to be considered local, these are not involved in the communication process and they are assumed as not repeated. This functionality reduces the communication burden but adds some complications because the user must carefully select the items that are considered to be non-repeated. The *changeMap* redistributes the data of the vector using the *NewMap* as the criterion to send data across the communicator. The *vectorNormalExclusive* function redistributes data using the *normal* distribution and eliminating the repeated elements having equal *gids*.

```

    /*! @name Communicator manipulations */ //@{
    public:
    /*! Copies the map on a smaller communicator,
    all the pids not included in the new comm will be truncated */

```

```

void reduceCommunicator(const bool                                & isActive,
                       const Teuchos::RCP<const communicator> & OldCommDev,
                       const Teuchos::RCP<const PVECT>         & OldVect,
                       const Teuchos::RCP<const communicator> & NewCommDev,
                       Teuchos::RCP<PVECT>                     & NewVect);

/*! Copies the map on a smaller communicator, all the pids not included in the new comm will be truncated */
void reduceCommunicator(const bool                                & isActive,
                       const communicator & OldCommDev,
                       const PVECT         & OldVect,
                       const communicator & NewCommDev,
                       PVECT               & NewVect);

/*! Copies the map on a bigger communicator, all the pids not included are void */
void expandCommunicator(const bool                                & isActive,
                       const Teuchos::RCP<const communicator> & OldCommDev,
                       const Teuchos::RCP<const PVECT>         & OldVect,
                       const Teuchos::RCP<const communicator> & NewCommDev,
                       Teuchos::RCP<PVECT>                     & NewVect);

/*! Copies the map on a bigger communicator, all the pids not included are void */
void expandCommunicator(const bool                                & isActive,
                       const communicator & OldCommDev,
                       const PVECT         & OldVect,
                       const communicator & NewCommDev,
                       PVECT               & NewVect);

//@}

```

The expand and reduce communicator functions are used in the same way as described for the *pMapGlobalManip* class in Section 6.3. They pass the map and data to the new reduced, or expanded, communicator. Let us now pass to describe some more functions that are available when using the *pMapItemSharing* specialization

```

/*! Assigns a unique \c gid to data and constructs the sharing/owning data structure.
Only items marked as \c shared = true are assumed to be shared.
All items that are found to be not shared are marked as not shared.
If two data are equal the same gid is assigned to both.
Data should implement the inequality and less operator.*/
void buildSharingOwnership(Teuchos::RCP<PVECT> & Vect) const;

/*! Assigns a unique \c gid to data and constructs the sharing/owning data structure.
Only items marked as \c shared = true are assumed to be shared.
All items that are found to be not shared are marked as not shared.
If two data are equal the same gid is assigned to both.
Data should implement the inequality and less operator.*/
void buildSharingOwnership(PVECT & Vect) const;

/*! Sends and overwrites data of the shared-and-owned
to all other shared-and-notOwned items.
The communication map \c mapSend should be provided.
See \c pMapGlobalManip to construct the commMap. */
void updateData(      Teuchos::RCP<PVECT>         & Vect,
                 const Teuchos::RCP<const SENDRECV> & mapSend) const;

/*! Sends and overwrites data of the shared-and-owne
d to all other shared-and-notOwned items.
The communication map \c mapSend should be provided.

```

```

See \c pMapGlobalManip to construct the commMap. */
void updateData(          PVECT & Vect,
                        const SENDRECV & mapSend) const;


/*! Sends and overwrites data of the shared-and-owned
to all other shared-and-notOwned items.
The communication map \c mapSend should be provided.
See \c pMapGlobalManip to construct the commMap. */
void updateData(          Teuchos::RCP<PVECT>          & Vect,
                        const Teuchos::RCP<const SENDRECV> & mapSend,
                        const Teuchos::RCP<const SENDRECV> & mapRecv) const;

/*! Sends and overwrites data of the shared-and-owned
to all other shared-and-notOwned items.
The communication map \c mapSend should be provided.
See \c pMapGlobalManip to construct the commMap. */
void updateData(          PVECT & Vect,
                        const SENDRECV & mapSend,
                        const SENDRECV & mapRecv) const;

```

This and the following two blocks contain all the functions used to update the shared data using the owned piece of data as shown in Figure 3. This particular block contains only the blocking versions and it is possible to prescribe only the *mapSend* sending map or both the sending and *mapRecv* receiving map. The version that uses both maps is slightly more efficient. Some more efficient,

map					data
pid	lid	gid	shared	owned	
0	1	1	F	T	7
0	2	2	F	T	12
0	3	3	T	T	-1
0	4	4	T	F	5
1	1	3	T	F	-3
1	2	4	T	T	8
1	3	5	F	T	3
1	4	6	F	T	2



map					data
pid	lid	gid	shared	owned	
0	1	1	F	T	7
0	2	2	F	T	12
0	3	3	T	T	-1
0	4	4	T	F	8
1	1	3	T	F	-1
1	2	4	T	T	8
1	3	5	F	T	3
1	4	6	F	T	2

Figure 3: A parallel vector before (up) and after (down) the call to the *updateData* method.

non-blocking, versions of these kind of methods are available.

```

/*! @name Non-pending update */ //@{
public:
/*! Sends and overwrites data of the shared-and-owned
   to all other shared-and-notOwned items.
   The communication map \c mapSend should be provided.
   See \c pMapGlobalManip to construct the commMap. */
void updateDataI(      Teuchos::RCP<PVECT>          & Vect,
                     const Teuchos::RCP<const SENDRECV> & mapSend,
                     sVect<PVECT>                    & commSegments,
                     sVect<PVPS>                      & sendPvps,
                     sVect<PVPS>                      & recvPvps,
                     const UInt                        & channel = 2) const;

/*! Sends and overwrites data of the shared-and-owned
   to all other shared-and-notOwned items.
   The communication map \c mapSend should be provided.
   See \c pMapGlobalManip to construct the commMap. */
void updateDataO(      Teuchos::RCP<PVECT>          & Vect,
                     const Teuchos::RCP<const SENDRECV> & mapSend,
                     sVect<PVECT>                    & commSegments,
                     sVect<PVPS>                      & sendPvps,
                     sVect<PVPS>                      & recvPvps,
                     const UInt                        & channel = 2) const;

/*! Sends and overwrites data of the shared-and-owned
   to all other shared-and-notOwned items.
   The communication map \c mapSend should be provided.
   See \c pMapGlobalManip to construct the commMap. */
void updateDataI(      PVECT          & Vect,
                     const SENDRECV    & mapSend,
                     sVect<PVECT> & commSegments,
                     sVect<PVPS> & sendPvps,
                     sVect<PVPS> & recvPvps,
                     const UInt      & channel = 2) const;

/*! Sends and overwrites data of the shared-and-owned
   to all other shared-and-notOwned items.
   The communication map \c mapSend should be provided.
   See \c pMapGlobalManip to construct the commMap. */
void updateDataO(      PVECT          & Vect,
                     const SENDRECV    & mapSend,
                     sVect<PVECT> & commSegments,
                     sVect<PVPS> & sendPvps,
                     sVect<PVPS> & recvPvps,
                     const UInt      & channel = 2) const;

/*! Sends and overwrites data of the shared-and-owned
   to all other shared-and-notOwned items.
   The communication map \c mapSend should be provided.
   See \c pMapGlobalManip to construct the commMap. */
void updateDataI(      Teuchos::RCP<PVECT>          & Vect,
                     const Teuchos::RCP<const SENDRECV> & mapSend,
                     const Teuchos::RCP<const SENDRECV> & mapRecv,
                     sVect<PVECT>                    & commSegments,
                     sVect<PVPS>                      & sendPvps,
                     sVect<PVPS>                      & recvPvps,

```

```

        const UInt                                & channel = 2) const;

    /*! Sends and overwrites data of the shared-and-owned
       to all other shared-and-notOwned items.
       The communication map \c mapSend should be provided.
       See \c pMapGlobalManip to construct the commMap. */
    void updateDataO(        Teuchos::RCP<PVECT>          & Vect,
                            const Teuchos::RCP<const SENDRECV> & mapSend,
                            const Teuchos::RCP<const SENDRECV> & mapRecv,
                            sVect<PVECT>                  & commSegments,
                            sVect<PVPS>                    & sendPvps,
                            sVect<PVPS>                    & recvPvps,
                            const UInt                      & channel = 2) const;

    /*! Sends and overwrites data of the shared-and-owned
       to all other shared-and-notOwned items.
       The communication map \c mapSend should be provided.
       See \c pMapGlobalManip to construct the commMap. */
    void updateDataI(        PVECT          & Vect,
                            const SENDRECV   & mapSend,
                            const SENDRECV   & mapRecv,
                            sVect<PVECT>    & commSegments,
                            sVect<PVPS>     & sendPvps,
                            sVect<PVPS>     & recvPvps,
                            const UInt      & channel = 2) const;

    /*! Sends and overwrites data of the shared-and-owned
       to all other shared-and-notOwned items.
       The communication map \c mapSend should be provided.
       See \c pMapGlobalManip to construct the commMap. */
    void updateDataO(        PVECT          & Vect,
                            const SENDRECV   & mapSend,
                            const SENDRECV   & mapRecv,
                            sVect<PVECT>    & commSegments,
                            sVect<PVPS>     & sendPvps,
                            sVect<PVPS>     & recvPvps,
                            const UInt      & channel = 2) const;

    //@}

```

The structure is very similar to the one we have seen for the *pMapComm* class described in Section 8.2. There are two functions labeled with the letters *I* and *O*. The first one starts the communication phase while the second one closes it and updates the vector: some computations can be placed between the two functions. The functions have some buffers such as *commSegments*, *sendPvps* and *recvPvps* that must be passed from one function to the other as they are needed to check the communication phase. The channel determines the *tags* used in the communication phase: if no several *updateData* functions are called at the same time, then the default value can be used. For more information see the more detailed description regarding the communication channels included in Section 8.2.

```

    /*! @name Recursive update */ //@{
public:
    void updateDataRR(        Teuchos::RCP<PVECT>          & Vect,
                            const Teuchos::RCP<const SENDRECV> & mapSend,
                                                PVUR & commBuffer,
                            const UInt & channel = 2) const;

```

```

void updateDataRR(    PVECT    & Vect,
                     const SENDRECV & mapSend,
                     PVUR      & commBuffer,
                     const UInt    & channel = 2) const;

void updateDataRI(    Teuchos::RCP<PVECT>          & Vect,
                     const Teuchos::RCP<const SENDRECV> & mapSend,
                                     PVUR & commBuffer,
                                     const UInt & channel = 2) const;

void updateDataRI(    PVECT    & Vect,
                     const SENDRECV & mapSend,
                     PVUR      & commBuffer,
                     const UInt    & channel = 2) const;

void updateDataRO(    Teuchos::RCP<PVECT>          & Vect,
                     const Teuchos::RCP<const SENDRECV> & mapSend,
                                     PVUR & commBuffer,
                                     const UInt & channel = 2) const;

void updateDataRO(    PVECT    & Vect,
                     const SENDRECV & mapSend,
                     PVUR      & commBuffer,
                     const UInt    & channel = 2) const;

void updateDataRR(    Teuchos::RCP<PVECT>          & Vect,
                     const Teuchos::RCP<const SENDRECV> & mapSend,
                     const Teuchos::RCP<const SENDRECV> & mapRecv,
                                     PVUR & commBuffer,
                                     const UInt & channel = 2) const;

void updateDataRR(    PVECT    & Vect,
                     const SENDRECV & mapSend,
                     const SENDRECV & mapRecv,
                     PVUR      & commBuffer,
                     const UInt    & channel = 2) const;

void updateDataRI(    Teuchos::RCP<PVECT>          & Vect,
                     const Teuchos::RCP<const SENDRECV> & mapSend,
                     const Teuchos::RCP<const SENDRECV> & mapRecv,
                                     PVUR & commBuffer,
                                     const UInt & channel = 2) const;

void updateDataRI(    PVECT    & Vect,
                     const SENDRECV & mapSend,
                     const SENDRECV & mapRecv,
                     PVUR      & commBuffer,
                     const UInt    & channel = 2) const;

void updateDataRO(    Teuchos::RCP<PVECT>          & Vect,
                     const Teuchos::RCP<const SENDRECV> & mapSend,
                     const Teuchos::RCP<const SENDRECV> & mapRecv,
                                     PVUR & commBuffer,
                                     const UInt & channel = 2) const;

void updateDataRO(    PVECT    & Vect,
                     const SENDRECV & mapSend,

```

```

const SENDRECV & mapRecv,
      PVUR      & commBuffer,
const UInt     & channel = 2) const;
//@}

```

Finally, there is also a recursive set of methods to use in all cases where the update phase must be repeated several times: for instance these functions are useful to update the ghost cells at each time step of a time-evolving algorithm. In this case the vector cannot change its map and its length, only the values of the *dofs* may change. *RR* methods must be called only once at the beginning of the update phase to set up the *commBuffer* communication buffer. Then each update phase is composed by the *RI* method which starts the communication and by the *RO* method which ends it and updates the vector. Only the values corresponding to *shared* items may change, therefore all other values (*not* – *shared*) can be modified freely between the *RI* and *RO* functions.

9 Traits classes

In this package there is only one trait class i.e. *traitsSegmentationUtility*. This class makes it possible to divide a given vector its maximum and minimum values and the number of segments to be considered. The first part of the file contains several implementations of the *linearCombination* function:

```

//-----
// DOFS THAT CAN BE SEGMENTED
//-----
/*! Trait for UInt */
inline UInt linearCombination(const UInt & G1,
                             const UInt & G2,
                             const Real & a)
{
return(UInt( Real(G1) + Real(G2 - G1) * a ));
}

/*! Trait for Real */
inline Real linearCombination(const Real & G1,
                             const Real & G2,
                             const Real & a)
{
return(G1 * (1.0 - a) + G2 * a);
}

/*! Trait for point2d */
inline point2d linearCombination(const point2d & G1,
                                const point2d & G2,
                                const Real & a)
{
point2d G;

G.X[0] = G1.X[0] * (1.0 - a) + G2.X[0] * a;
G.X[1] = G1.X[1] * (1.0 - a) + G2.X[1] * a;

return(G);
}

```

```

    /*! Trait for point3d */
    inline point3d linearCombination(const point3d & G1,
                                    const point3d & G2,
                                    const Real    & a)
    {
        point3d G;

        G.X[0] = G1.X[0] * (1.0 - a) + G2.X[0] * a;
        G.X[1] = G1.X[1] * (1.0 - a) + G2.X[1] * a;
        G.X[2] = G1.X[2] * (1.0 - a) + G2.X[2] * a;

        return(G);
    }

    /*! Trait for tensor2d */
    inline tensor2d linearCombination(const tensor2d & G1,
                                      const tensor2d & G2,
                                      const Real    & a)
    {
        tensor2d G;

        G.T[0][0] = G1.T[0][0] * (1-a) + G2.T[0][0] * a;
        G.T[0][1] = G1.T[0][1] * (1-a) + G2.T[0][1] * a;

        G.T[1][0] = G1.T[1][0] * (1-a) + G2.T[1][0] * a;
        G.T[1][1] = G1.T[1][1] * (1-a) + G2.T[1][1] * a;

        return(G);
    }

    /*! Trait for tenso3d */
    inline tensor3d linearCombination(const tensor3d & G1,
                                      const tensor3d & G2,
                                      const Real    & a)
    {
        tensor3d G;

        G.T[0][0] = G1.T[0][0] * (1-a) + G2.T[0][0] * a;
        G.T[0][1] = G1.T[0][1] * (1-a) + G2.T[0][1] * a;
        G.T[0][2] = G1.T[0][2] * (1-a) + G2.T[0][2] * a;

        G.T[1][0] = G1.T[1][0] * (1-a) + G2.T[1][0] * a;
        G.T[1][1] = G1.T[1][1] * (1-a) + G2.T[1][1] * a;
        G.T[1][2] = G1.T[1][2] * (1-a) + G2.T[1][2] * a;

        G.T[2][0] = G1.T[2][0] * (1-a) + G2.T[2][0] * a;
        G.T[2][1] = G1.T[2][1] * (1-a) + G2.T[2][1] * a;
        G.T[2][2] = G1.T[2][2] * (1-a) + G2.T[2][2] * a;

        return(G);
    }

    /*! Trait for stateVector */
    inline stateVector linearCombination(const stateVector & G1,
                                         const stateVector & G2,
                                         const Real    & a)

```



```

{
assert(G1.Length() == G2.Length());

stateVector G(G1.Length());

for(int i=1; i <= G1.Length(); ++i)
{
G.operator()(i) = G1.operator()(i) * (1.0-a) + G2.operator()(i) * a;
}

return(G);
}

/*! Trait for staticVector */
template<size_t N>
staticVector<N> linearCombination(const staticVector<N> & G1,
                                const staticVector<N> & G2,
                                const Real          & a)
{
staticVector<N> G;

for(UInt i=1; i <= N; ++i)
{
G.operator()(i) = G1.operator()(i) * (1.0-a) + G2.operator()(i) * a;
}

return(G);
}

/*! Trait for staticMatrix */
inline stateMatrix linearCombination(const stateMatrix & G1,
                                    const stateMatrix & G2,
                                    const Real          & a)
{
assert(G1.RowDim() == G2.RowDim());
assert(G1.ColDim() == G2.ColDim());

stateMatrix G( G1.RowDim(), G1.ColDim());

for(int i=1; i <= G1.RowDim(); ++i)
{
for(int j=1; j <= G1.ColDim(); ++j)
{
G.operator()(i,j) = G1.operator()(i,j) * (1.0 - a) + G2.operator()(i,j) * a;
}
}

return(G);
}

/*! Trait for pGraphItem */
inline pGraphItem linearCombination(const pGraphItem & G1,
                                    const pGraphItem & G2,
                                    const Real          & a)
{
//Asserts
assert(G1.nodesOrdered);

```

```

assert(G2.nodesOrdered);
assert(G1.connected.size() == G2.connected.size());
assert(G1.orderedNodes.size() == G2.orderedNodes.size());
assert(G1.orderedNodes.size() == G1.connected.size());
assert(G1.getNodesOrdered());
assert(G2.getNodesOrdered());

//Resizing
pGraphItem G;
G.connected.resize(G1.connected.size());

//Computing
for(UInt i=1; i <= G1.connected.size(); ++i)
{
G.connected(i) = Real(G1.getSorted(i)) * (1.0 - a) + Real(G2.getSorted(i)) * a;
}

//Updating and return
G.updateSorting();
return(G);
}

/*! Trait for geoElement */
template<typename GEOSHAPE>
geoElement<GEOSHAPE> linearCombination(const geoElement<GEOSHAPE> & G1,
                                       const geoElement<GEOSHAPE> & G2,
                                       const Real & a)
{
//Asserts
assert(G1.nodesOrdered);
assert(G2.nodesOrdered);
assert(G1.connected.size() == G2.connected.size());
assert(G1.orderedNodes.size() == G2.orderedNodes.size());
assert(G1.orderedNodes.size() == G1.connected.size());
assert(G1.getNodesOrdered());
assert(G2.getNodesOrdered());

//Resizing
geoElement<GEOSHAPE> G;
G.connected.resize(G1.connected.size());

//Computing
for(UInt i=1; i <= G1.connected.size(); ++i)
{
G.connected(i) = Real(G1.getSorted(i)) * (1.0 - a) + Real(G2.getSorted(i)) * a;
}

//Updating and return
G.updateSorting();
return(G);
}

```

This function implements a very simple convex linear combination: all the *dofs* have their own specialization but also some other types such as:

- *pGraphItem*;
- *geoElement < GEOSHAPE >*;

are involved. These other types will be introduced in the *morganaGraph* and *morganaGeometry* parts of the manual. The *linearCombination* function requires two types P_1 and P_2 , a real number a and it returns $(1-a)P_1 + aP_2$. In general all specializations are quite simple, however **all types that involve the use of integer types require a special treatment due to rounding**. To be more precise every linear combination must satisfy $P_1 \leq (1-a)P_1 + aP_2 \leq P_2$ where $a \in [0, 1]$.

The *linearCombination* function is then exploited by the *dataSegmentationUtility* class

```

/! This is the only trait class in the \c morganaContainer package.
This trait specifies how to segment a vector containing
some data into pieces. This is mainly used for segmentation
on the processors vectors that has not a \c gid identificator.

For instance all functions that generate a global numbering
usually use this class. We have implemented some functions for the
various types of data that can be segmented: they are mainly all
the \c morganaDofs and the \c pGraphItem classes. The second one
are used much more often than the \c morganaDofs. The numbering
of locally created geometric faces is an example of usage of this class.

Given a vector of ordinalable data this class divides it into a number of sub-vectors.
The data type should satisfy:  $P_1 \leq (1-a)P_1 + aP_2 \leq P_2$  where  $a$  is a real  $[0,1]$  */
template<typename DATA> class dataSegmentationUtility
{
/! @name Typedefs */ //@{
public:
typedef std::pair<DATA,UInt>          PAIR;
typedef std::map<DATA,UInt>          STLMAP;
typedef typename STLMAP::const_iterator  ITERATOR;
//@}

/! @name Internal data */ //@{
public:
  UInt N;
  DATA  MaxData;
  STLMAP orderMap;
  //@}

/! @name Functions */ //@{
public:
  dataSegmentationUtility(const UInt & n, const DATA & minData, const DATA & maxData);
  UInt index(const DATA & value) const;

template<typename D>
friend ostream & operator<<(ostream & f, const dataSegmentationUtility<D> & M);
  //@}
};

```

The *dataSegmentationUtility* constructor requires the *maxData* maximum value and the *minData* minimum value of a given interval and the number n of sub-intervals to build. For instance if $n = 2$, then the class generates two intervals $[minData, (minData+maxData)/2]$ and $[(minData+maxData)/2, maxData]$. Once initialized the class returns, for a given *DATA value*, the corresponding index of the sub-interval to which *value* belongs.

10 Tutorials

Many tutorials and examples can be found in the */tests/morganaContainer* folder so here we will provide a very brief description of some of the files there contained. The *sOctTreeTest1.cpp* file represents a very simple example for the construction of an *octTree*. In *databaseLin1dTest1.cpp* a few data are added to a linear database and its interpolation method is checked.

There are several examples regarding parallel maps: in *pMapTest2.cpp* a very simple example of how to create a parallel map is introduced along with the handling of a very simple communication phase. The serial manipulation of parallel maps is discussed, for instance, in *pMapManipTest3.cpp*, where the *segmentationNormal* function is used, and in *pMapManipTest5.cpp*, where the *setNormalIndexing* method is used. The parallel communication of maps is tested, for instance, in *pMapCommTest1.cpp* where a simple *sendRecv* communication path is used to transfer a map from a communicator to another one. In *pMapCommTest4.cpp* the *vectorNormal* method is used to distribute the map on the communicator according to the specific *gid*. Finally, the *vectorPid* method is implemented in *pMapCommTest6.cpp* file. The parallel manipulation of maps includes several tests since there are many complex functionalities implemented. For instance, in *pMapGlobalManipTest1.cpp*, *pMapGlobalManipTest4.cpp* and *pMapGlobalManipTest6.cpp* the *check* method is used to verify whether the map is consistent or not. The user is encouraged to modify some aspects of the map and see whether errors in the map lead to its detection. In particular, all three *mapItems* types are covered. The interface between *morgana* maps and *trilinos* maps is covered in a number of examples such as *pMapGlobalManipTest2.cpp*, *pMapGlobalManipTest11.cpp* and *pMapGlobalManipTest12.cpp*. The statistics of a map are extracted in *pMapGlobalManipTest3.cpp*. A very important functionality of the parallel maps is the possibility of creating communication patterns through the *sendRecv* type of maps, see, for instance, *pMapGlobalManipTest5.cpp*. Finally, the creation and fixing of the ownership structures is investigated in *pMapGlobalManipTest9.cpp*, *pMapGlobalManipTest10.cpp* and the shift of communicator is detailed in *pMapGlobalManipTest13.cpp* and *pMapGlobalManipTest14.cpp*.

Let us now pass to the tutorials of parallel vectors: some simple examples are discussed in *pVectTest1.cpp*, *pVectTest2.cpp*. The segmentation of a vector in several sub-vectors, in a serial environment, is implemented in *pVectManipTest1.cpp*, *pVectManipTest9.cpp* for the *segmentationNormal* function, in *pVectManipTest2.cpp*, *pVectManipTest10.cpp* for the *segmentationSimple* method, in *pVectManipTest3.cpp*, *pVectManipTest11.cpp*, *pVectManipTest12.cpp* for the *segmentationPid* function and in *pVectManipTest7.cpp*, *pVectManipTest15.cpp* for the *segmentationData* function. Merging methods are included in *pVectManipTest4.cpp* for the *mergeSimple* function and in *pVectManipTest5.cpp*, *pVectManipTest8.cpp*, *pVectManipTest13.cpp* for the *unionExclusive* method. The *setIndexing* function is discussed in *pVectManipTest6.cpp*, *pVectManipTest14.cpp*.

Now passing to parallel communication of *pVect*, we have implemented some tests both for the *pending* and *non – pending* paradigm. Regarding the former

paradigm, in *pVectCommTest1.cpp* a simple case of *sendRecv* communication is implemented, in *pVectCommTest4.cpp* a simple communication pattern is carried out using *pMapItemSendRecv* items. In *pVectCommTest5.cpp* the *vectorNormal* function is used and in *pVectCommTest6.cpp* the *vectorData* method is implemented. There are also some examples of communications using the *non-pending* paradigm among which we point out that in *pVectPendingTest2.cpp* an example of communication is carried out and in *pVectPendingTest3.cpp* it is underlined how to perform some computations between the start and the end of a communication phase. In this case the computation block is simply a dummy research of prime numbers. In *pVectPendingTest9.cpp* the recursive communication system is exploited.

The global manipulation of a parallel vector is tested, for instance, in *pVectGlobalManipTest1.cpp* and *pVectGlobalManipTest3.cpp* where the *buildGlobalNumbering* and *buildSharingOwnership* methods are used, respectively. In *pVectGlobalManipTest4.cpp* a more complex test is performed: given a vector, a communication update map is created using *pMapItemSendRecv*. This map is then used to update the data in the vector so that the shared but not-owned elements are updated by the corresponding shared and owned ones. To conclude, in *pVectGlobalManipTest7.cpp*, *pVectGlobalManipTest11.cpp*, *pVectGlobalManipTest12.cpp* and *pVectGlobalManipTest14.cpp* the *reduceCommunicator*, *vectorNormalExclusive*, *checkConsistency* and *destroyOverlap* methods are tested, respectively.

References

- [1] Trilinos project: <http://trilinos.sandia.gov/>.