Communication within the Iterative Solver Template Library (ISTL)*

Markus Blatt
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen,
Universität Heidelberg, Im Neuenheimer Feld 368, D-69120 Heidelberg,
email: Markus.Blatt@iwr.uni-heidelberg.de

April 27, 2005

Abstract

This document describes usage and interface of the classes meant for setting up the communication within a parallel programm using ISTL. As most of the communication in distributed programm occur in the same pattern it is often more efficient (and of course more easy for the programmer) to build the communication pattern once in the programm and then use multiple times (e. g. at each iteration step of an iterative solver).

Contents

1 Introduction

When using the data parallel programming model a set of processes works collectively on the same set of finite data objects. These might be elements of a finite element grid or vector entries in a linear algebra computation. Each process works on different partitions of the global data. Only for this partition it computes updated values.

In large scale parallel codes it is advisable to store the data partition in a local data structure directly in the local memory of the process. Due to data dependencies the process needs to access data in the partition of other processes, too. This can either be done by communicating these values on demand between the processes whenever they are accessed. This results in data structures that are aware of the data distribution. Or by augmenting the partition of the process such that it additionally includes the data values that the other values depend on. Note that now the partitioning is not disjoint any more but overlapping. Of course the values other processes compute for need to be updated using communication at so called synchronisation points of the algorithm

In the latter case the data structures do not need to know anything about the data distribution. This demands more effort from the parallel algorithm designer to make

*Part of the Distributed and Unified Numerics Environment (DUNE) which is available from the site http://www.dune-project.org/

sure that the data used for computations is valid, i.e. contains an updated value if another process computes the data for it. Still it allows for fewer synchronisation points in the algorithms as even in collective operations all input data may already be updated from other processes due to a previous operation. Between the necessary synchronisation points one can take advantage of the fast local memory access.

Consider representing a random access container x on a set of processes $\mathcal{P} = \{0, \ldots, P-1\}$. It is represented by individual pieces x^p, where x^p is the piece stored on process p of the P processes participating in the calculation. Although the global representation of the container is not available on any process, a process p needs to know how the entries of its local piece x^p correspond to the entries of the global container x, which would be used in a sequential program.

2 Communication Software Components

From an abstract point of view a random access container $x : I \to K$ provides a mapping from an index set $I \subset \mathbb{N}_0$ onto a set of objects K. Note that we do not require I to be consecutive. The piece x_p of the container x stored on process p is a mapping $x_p : I_p \to K$, where $I_p \subset I$. Due to efficiency the entries of x_p should be stored consecutively in memory. This means that for the local computation the data must be addressable by a consecutive index starting from 0.

When using adaptive discretisation methods there might be the need to reorder the indices after adding and/or deleting some of the discretisation points. Therefore this index does not need to be persistent and can easily be changed. We will call this index *local index*.

For the communication phases of our algorithms these locally stored entries must also be addressable by a global identifier. It is used to store the received values at and to retrieve the values to be sent from the correct local position in the consecutive memory chunk. To ease the addition and removal of discretisation points this global identifier has to be persistent but does not need to be consecutive. We will call this global identifier *global index*.

2.1 ParallelIndexSet

Let $I \subset \mathbb{N}_0$ be an arbitrary, not necessarily consecutive, index set identifying all discretisation points of the computation. Furthermore, let

$$(I_p)_{p\in\mathcal{P}}, \quad \bigcup_{p\in\mathcal{P}} I_p = I$$

be an overlapping decomposition of the global index set I into the sets of indices I_p corresponding to the global indices of the values stored locally in the chunk of process p.

Then the class

```
template<typename TG, typename TL>  class ParallelIndexSet;
```

realises the one to one mapping

$$\gamma_p \; : \; I_p \longrightarrow I_p^{\mathrm{loc}} := [0, n_p)$$

of the globally unique index onto the local index.

The template parameter `TG` is the type of the global index and `TL` is the type of the local index. The only prerequisite of `TG` is that objects of this type are comparable

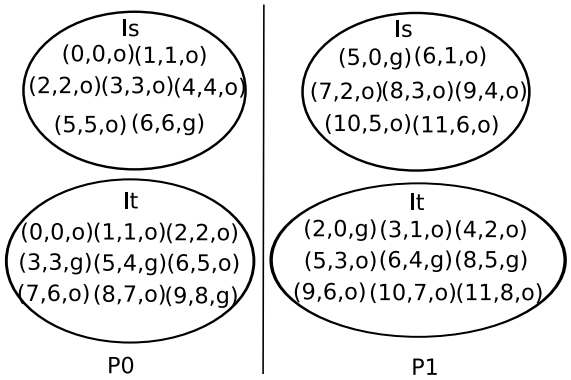

Figure 1: Index sets for array redistribution

using the less-than-operator `<`. Not that this prerequisite still allows attaching further information to the global index or even using this information as the global index. The type `TL` has to be convertible to `std::size_t` as it is used to address array elements.

The pairs of global and local indices are ordered by ascending global index. It is possible to access the pairs via `operator[](TG& global)` in $log(n)$ time, where n is the number of pairs in the set. In an efficient code it is advisable to access the index pairs using the provided iterators over the index pairs.

Due to the ordering, the index set can only be changed, i.e. index pairs added or deleted, in a special resize phase. By calling the functions `beginResize()` and `endResize()` the programmer indicates that the resize phase starts and ends, respectively. During the call of `endResize()` the deleted indices will be removed and the added index pairs will be sorted and merged with the existing ones.

2.2 ParallelLocalIndex

When dealing with overlapping index sets in distributed computing there often is the need to distinguish different partitions of an index set.

This is accomplished by using the class

```
template<typename TA> class ParallelLocalIndex;
```

as the type for the local index of class `ParallelIndexSet`. Here the template parameter `TA` is the type of the attributes used, e.g. an enumeration `Flags` defined by

```
enum Flags {owner, ghost};
```

where `owner` marks the indices $k \in I_p$ owned by process p and `ghost` the indices $k \notin I_p$ owned by other processes.

As an example let us look at an array distributed between two processes. In Figure 2 one can see the array a as it appears in a sequential program. Below there are two different distributions of that array. The local views s_0 and s_1 are the parts process 0 and 1 store in the case that a is divided into two blocks. The local views t_0 and t_1 are the parts of a that process 0 and 1 store in the case that a is divided into 4 blocks and process 0 stores the first and third block and process 1 the second and fourth block. The decompositions have an overlap of one and the indices have the attributes `owner` and `ghost` visualised by white and shaded cells, respectively. The index sets I_s and I_t corresponding to the decompositions s_p and t_p, $p \in \{0, 1\}$, are shown in Figure 1 as sets of triples (g, l, a). Here g is the global index, l is the local index and a is the attribute (either o for `owner` or g for `ghost`).

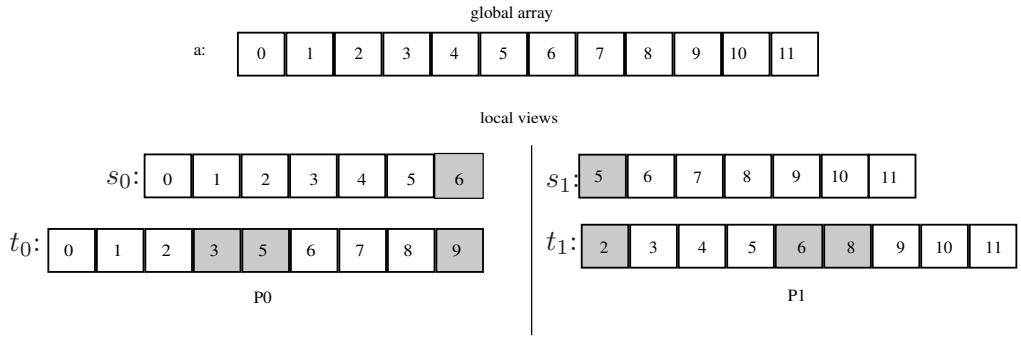

Figure 2: Redistributed array

The following code snippet demonstrates how to set up the index set I_s on process 0:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
using namespace Dune;
// shortcut for index set type
typedef ParallelLocalIndex<Flags> LocalIndex;
typedef ParallelIndexSet<int, LocalIndex > PIndexSet;
sis.beginResize();
if(rank==0) {
  sis.add(5, LocalIndex(0, ghost));
```

2.3 Remote Indices

To set up communication between the processes every process needs to know which indices are also known to other processes and which attributes are attached to them on the remote side. There are scenarios where data is exchanged between different index sets, e.g. if the data is agglomerated on lesser processes or redistributed. Therefore communication is allowed to occur between different decompositions of the same index set.

Let $I \subset \mathbb{N}$ be the global index set and

$$(I_p^s)_{p\in\mathcal{P}}, \quad \bigcup_{p\in\mathcal{P}} I_p^s = I, \quad \text{and} \quad (I_p^t)_{p\in\mathcal{P}}, \quad \bigcup_{p\in\mathcal{P}} I_p^t = I$$

be two overlapping decompositions of the same index set I. Then an instance of class `RemoteIndices` on process $p \in \mathcal{P}$ stores the sets of triples

$$r_{p\to q}^s = \{(g,(l,a),b) \,|\, g \in I_q^s \wedge g \in I_p^t, l = \gamma_p^s(g), a = \alpha_p^s(l), b = \alpha_q^t(\gamma_q^t(g))\} \tag{1}$$

and

$$r_{p\to q}^t = \{(g,(l,a),b) \,|\, g \in I_q^s \wedge g \in I_p^t, l = \gamma_p^t(g), a = \alpha_p^t(l), b = \alpha_p^s(\gamma_p^s(g))\}, \tag{2}$$

for all $q \in \mathcal{P}$. Here α_p^s and α_p^t denote the mapping of local indices on process p onto attributes for the index set I_p^s and I_p^t as realised by `ParallelLocalIndex`. Note that the sets $r_{p\to q}^s$ and $r_{p\to q}^t$ will only be nonempty if the processes p and q manage overlapping index sets.

For our example in Figure 2 and Figure 1 the interface between I_s and I_t on process 0 is:

$$r^s_{0\to 0} = \{(0,(0,o),o),(1,(1,o),o),(2,(2,o),o),(3,(3,o),g),(5,(5,o),g),(6,(6,g),o)\}$$
$$r^t_{0\to 0} = \{(0,(0,o),o),(1,(1,o),o),(2,(2,o),o),(3,(3,g),o),(5,(4,g),o),(6,(5,o),g)\}$$
$$r^s_{0\to 1} = \{(2(2,o),g),(3,(3,o),o),(4,(4,o),o),(5,(5,o),o),(6,(6,g),g)\}$$
$$r^t_{0\to 1} = \{(5,(4,g),g),(6,(5,o),o),(7,(6,o),o),(8,(7,o),o),(9,(8,g),o)\}$$

This information can either be calculated automatically by communicating all indices in a ring or set up by hand if the user has this information available. Assuming that `sis` is the index set I_s and `tis` the index set I_t set up as described in the previous subsection and `comm` is an MPI communicator then the simple call

```
    tis.add(g, LocalIndex(l++, flag));
  }
```

on all processes automatically calculates this information and stores it in `riRedist`. For a parallel calculation on the local views s_0 and s_1 calling

```
std::cout<<rank<<"␣isxset:␣"<<sis<<std::endl;
```

on all processes builds the necessary information in `riS`.

2.4 Communication Interface

With the information provided by class `RemoteIndices` the user can set up arbitrary communication interfaces. These interfaces are realised in `template<typename T> class Interface`, where the template parameter `T` is the custom type of the `ParallelIndexSet` representing the index sets. Using the attributes attached to the indices by `ParallelLocalIndex` the user can select subsets of the indices for exchanging data, e.g. send data from indices marked as `owner` to indices marked as `ghost`.

Basically the interface on process p manages two sets for each process q it shares common indices with:

$$i^s_{p\to q} = \{l|(g,(l,a),b) \in r^s_{p\to q} | a \in A_s \wedge b \in A_t\}$$

and

$$i^t_{p\to q} = \{l|(g,(l,a),b) \in r^t_{p\to q} | a \in A_t \wedge b \in A_s\}\,,$$

where A_s and A_t are the attributes marking the indices where the source and target of the communication will be, respectively.

In our example these sets on process 0 will be stored for communication if $A_s = \{o\}$ and $A_t = \{o, g\}$:

$$i^s_{0\to 0} = \{0,1,3,5\} \qquad i^t_{0\to 0} = \{0,1,3,4\}$$
$$i^s_{0\to 1} = \{2,3,4,5\} \qquad i^t_{0\to 1} = \{5,6,7,8\}\,.$$

The following code snippet would build the interface above in `infRedist` as well as the interface `infS` to communicate between indices marked as `owner` and `ghost` on the local array views s_0 and s_1:

```
riRedist.rebuild<true>();

std::vector<int> v;
```

```
RemoteIndices<PIndexSet> riS(sis,sis, comm, v, true);
riS.rebuild<false>();

std::cout<<std::endl<<"begin"<<rank<<"␣riS="<<riS<<"␣end"<<rank<<std::en

Combine<EnumItem<Flags,ghost>,EnumItem<Flags,owner>,Flags> ghostFlags;
```

2.5 Communicator

Using the classes from the previous sections all information about the communication is available and we are set to communicate data values of arbitrary container types. The only prerequisite for the container type is that its values are addressable via `operator[](size_t index)`. This should be safe to assume.

An important feature of our communicators is that we are not only able to send one data item per index, but also different numbers of data elements (of the same type) for each index. This is supported in a generic way by the traits class `template<class V> struct CommPolicy` describing the container type `V`. The `typedef IndexedType` is the atomic type to be communicated and `typedef IndexedTypeFlag` is either `SizeOne` if there is only one data item per index or `VariableSize` if the number of data items per index is variable.

The default implementation works for all array-like containers which provide only one data item per index. For all other containers the user has to provide its own custom specialisation.

The class `template<class T> class BufferedCommunicator` performs the actual communication. The template parameter `T` describes the type of the parallel index set. It uses the information about the communication interface provided by an object of class `Interface` to set up communication buffers for a container containing a specific data type. It is also responsible for gathering the data before and scattering the data after the communication step. The strict separation of the interface description from the actual buffering and communication allows for reusing the interface information with various different container and data types.

Before the communication can start one has to call the `build` method with the data source and target containers as well as the communication interface as arguments. Assuming `s` and `t` as arrays s_i and t_i, respectively, then

```
infRedist.build(riRedist, ownerFlags, allFlags);
infS.build(riS, ownerFlags, ghostFlags);
```

demonstrates how to set up the communicator `bCommRedist` for the array redistribution and `bComm` for a parallel calculation on the local views s_i. The `build` function calculates the size of the messages to send to other processes and allocates buffers for the send and receive actions. The representatives `s` and `t` are needed to get the number of data values at each index in the case of variable numbers of data items per index. Note that, due to the generic programming techniques used, the compiler knows if the number of data points is constant for each index and will apply a specialised algorithm for calculating the message size without querying neither `s` nor `t`. Clean up of allocated resources is done either by calling the method `free()` or automatically in the destructor.

The actual communication takes place if one of the methods `forward` and `backward` is called. In our case in `bCommRedist` the `forward` method sends data from the local views s_i to the local views t_i according to the interface information and the `backward` method in the opposite direction.

The following code snippet first redistributes the local views s_i of the global array to the local views t_i and performs some calculation on this representation. Afterwards the result is communicated backwards.

```
Container s(sis.size(),3), t(tis.size());

s[sis.size()-1]=-1;
```

Note that both methods have a different template parameter, either `CopyData` or `AddData`. These are policies for gathering and scattering the data items. The former just copies the data from and to the location. The latter copies from the source location but adds the received data items to the target entries. Assuming our data is stored in simple C-arrays `AddData` could be implemented like this:

```
enum Flags { owner, ghost };

template<typename T>
struct AddData {
  typedef typename T::value_type IndexedType;

  static const IndexedType& gather(const T& v, int i){
    return v[i];
  }

  static void scatter(T& v, const IndexedType& item, int i){
    v[i]+=item;
```

Note that arbitrary manipulations can be applied to the communicated data in both methods.

For containers with multiple data items associated with one index the methods `gather` and `scatter` must have an additional integer argument specifying the sub-index.

3 Collective Communication

While communicating entries of array-like structures is a prominent task in scientific computing codes one must not neglect collective communication operations, like gathering and scattering data from and to all processes, respectively, or waiting for other processes. An abstraction for these operations is crucial for decoupling the communication from the parallel programming paradigm used.

Therefore we designed `template<class T> class CollectiveCommunication` which provides information of the underlying parallel programming paradigm as well as the collective communication operations as known from MPI. See Table 1 for a list of all functions.

Currently there is a default implementation for sequential programs as well as a specialisation working with MPI. This approach allows for running parallel programs sequentially without any parallel overhead simply by choosing the sequential specialisation at compile time. Note that the interface is far more convenient to use than the C++ interface of MPI. The latter is a simple wrapper around the C implementation without taking advantage of the power of generic programming.

The collective communication classes were developed before the release of Boost.MPI [Gregor and Troyer(2006)]. In contrast to Boost.MPI it was never meant as a full generic implementation of all MPI functions. Instead it is restricted to the most basic subset

Function	Description
`int rank()`	Get the rank of the process
`int size()`	Get the number of processes
`template<typename T> T sum (T& in)`	Compute global sum
`template<typename T> T prod (T& in)`	Compute global product
`template<typename T> T min (T& in)`	Compute global minimum
`template<typename T> T max (T& in)`	Compute global maximum
`void barrier()`	Wait for all processes.
`template<typename T> int broadcast (T* in, int len, int root)`	Broadcast array from root to all other processes
`template<typename T> int gather (T* in, T* out, int len, int root)`	Gather array on root process
`template<typename BinaryFunction, typename Type> int allreduce(Type* in, Type* out, in`	Combine values from all processes on all processes. Combine function is given with `BinaryFunction`

Table 1: Collective Communication Functions

of collective operations needed to implement finite element methods and iterative solver using the previously described components. This lean interface should make it possible to easily port this approach to thread based parallelisation as well as other parallelisation paradigms. This would allow code to easily switch between different paradigms

References

[Gregor and Troyer(2006)] D. Gregor and M. Troyer. Boost.MPI. `http://www.boost.org/`, 2006.