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`

September 27, 2007

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

2 Index Sets

During distributed computations every discretization point needs to be indentified uniquely by every process regardless of where it is actually stored. In most scenarios it not advisable to store all the data needed for the computation on every process as memory is often a limiting factor in scientific computing. Therefore the data will distributed between the processes and each process will store only the data corresponding to its own part of the distribution. Due to the efficiency of the local commnunication is it normally best practice to hold the locally stored data in consecutive memory chunks.

This means that for the local computation the data must be adressable by a consecutive index starting from 0. When using adaptive discretization methods there might be need to reorder the indices after adding and/or deleting some of the discretization points. Therefore this index does not have to be persistent. Further on we will call this index *local index*.

For the communication phases of our algorithms these locally stored indices must also be adressable by a global identifier to be able to store the received values tagged with

[*]Part of the Distributed and Unified Numerics Environment (DUNE) which is available from the site `http://www.dune.uni-hd.de/`

the global identifiers at the correct local index in the consecutive local memory chunk. To ease the addition and removal of discretization points this global identifier has to be persistent. Further on we will call this global identifier *global index*.

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

Further more let $(I_p)_{p \in [0,P)}$, $\bigcup_{p=0}^{P-1} I_p = I$ be an overlapping decompostion of the global index set I into the sets of indices I_p corresponding to the discretization points stored locally on process p.

Then the

```
template<typename TG, typename TL, int N>
class IndexSet;
```

realizes the one to one mapping

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

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, that has to be convertible to `std::size_t`, and the parameter `N` is used internally to specify the chunk size of the array list.

The pairs of global and local index are ordered by ascending global index. Thus it it possible to access the pairs via `operator[](TG& global)` in $log(n)$ time, where n is the number of pairs in the set.

Due to the ordering the index set can only be changed, i. e. indices 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 indices will be merged with the existing ones.

To be able to attach further information to the index the only prerequesite for the type of the local index is that it is convertible to `std::size_t` as it it meant for adressing array elements.

ParallelLocalIndex When dealing with overlapping index sets in distributed computing there often is the need to distinguish different part of the index set, e. g. mark some of the indices as owned by the process and others as owned by another process.

This can easily be done by using the class

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

where the template parameter `TA` is the type of the attributes used, e. g. `enum{owner, overlap}`.

As the programmer often knows in advance which indices might also be present on other processes there is the possiblity to mark the index as public.

Usage Examples Let us look at a short example on how to build an index set. The code in Listing 1 sets up an index set with 8 components on two processes. This index set might be used to access as distributed field or `std::vector` as sketched in Figure 1.

The process 0 stores in his /em local field a_0 the values corresponding to the global indices $I_0 = \{0, 2, 6, 3, 5\}$, in that order, of the global field a and process I the entries corresponding to the values at the indices $I_1 = \{0, 1, 7, 5, 4\}$.

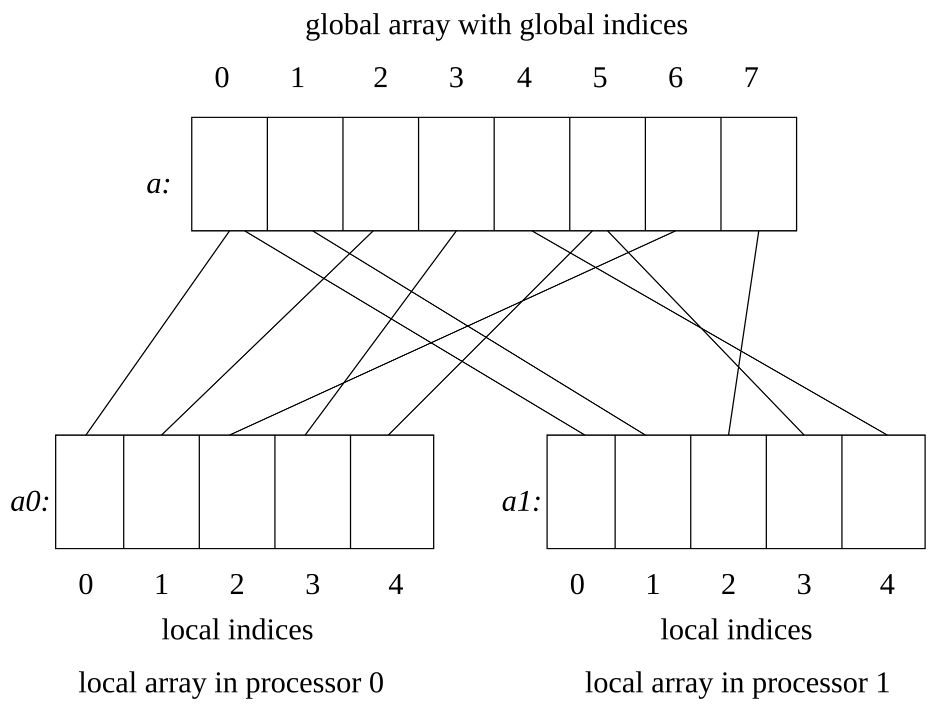
global array with global indices
0 1 2 3 4 5 6 7
a:
a0:
a1:
0 1 2 3 4
0 1 2 3 4
local indices
local indices
local array in processor 0
local array in processor 1

Figure 1: A Distributed Field

Listing 1: Build an Index Set

```
// $Id: buildindexset.hh 335 2005-10-05 14:45:11Z mblatt $
#ifndef BUILDINDEXSET_HH
#define BUILDINDEXSET_HH


#include<dune/istl/indexset.hh>
#include<dune/istl/plocalindex.hh>
#include"mpi.h"

/**
 * @brief Flag for marking the indices.
 */
enum Flag{owner, overlap};

// The type of local index we use
typedef Dune::ParallelLocalIndex<Flag> LocalIndex;

/**
 * @brief Add indices to the example index set.
 * @param indexSet The index set to build.
 */
template<class TG, int N>
void build(Dune::ParallelIndexSet<TG,LocalIndex,N>& indexSet)
{
  //
  // The number of processes
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // The rank of our process
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // Indicate that we add or remove indices.
  indexSet.beginResize();

  if(rank==0){
    indexSet.add(0, LocalIndex(0,overlap,true));
    indexSet.add(2, LocalIndex(1,owner,true));
    indexSet.add(6, LocalIndex(2,owner,true));
    indexSet.add(3, LocalIndex(3,owner,true));
    indexSet.add(5, LocalIndex(4,owner,true));
  }

  if(rank==1){
    indexSet.add(0, LocalIndex(0,owner,true));
    indexSet.add(1, LocalIndex(1,owner,true));
    indexSet.add(7, LocalIndex(2,owner,true));
    indexSet.add(5, LocalIndex(3,overlap,true));
    indexSet.add(4, LocalIndex(4,owner,true));
  }

  // Modification is over
  indexSet.endResize();
}
#endif
```

Due to the complexity of `operator[](TG& global)` it is always advisable to use iterators, obtained by calling `begin()` and `end()` respectively, to access the index pairs of the set.

Listing 2 demonstrates their usage. First the maximum local index $i_{\max}$ of the set is computed, and the the local indices are renumbered. Due to the ordering the local index with the smallest corresponding global index becomes $i_{\max}$ and the rest is numbered consecutively decreasingly with increasing global index. Let n be the number of index pairs in the set than local index corresponding to the largets global index becomes $i_{\max} - n$.

Listing 2: Usage of Index Set Iterators

```
// $Id: reverse.hh 335 2005-10-05 14:45:11Z mblatt $
#ifndef REVERSE_HH
#define REVERSE_HH

#include<dune/istl/plocalindex.hh>
#include<dune/istl/indexset.hh>

/**
 * @brief Reverse the local indices of an index set.
 *
 * Let the index set have N entries than the index 0 will become N-1,
 * 1 become N-2, ..., and N-1 will become 0.
 * @param indexSet The index set to reverse.
 */
template<typename TG, typename TL, int N>
void reverseLocalIndex(Dune::ParallelIndexSet<TG,TL,N>& indexSet)
{
  // reverse the local indices
  typedef typename Dune::ParallelIndexSet<TG,TL,N>::iterator iterator;

  iterator end = indexSet.end();
  size_t maxLocal = 0;

  // find the maximal local index
  for(iterator index = indexSet.begin(); index != end; ++index){
    // Get the local index
    LocalIndex& local = index->local();
    maxLocal = std::max(maxLocal, local.local());
  }

  for(iterator index = indexSet.begin(); index != end; ++index){
    // Get the local index
    LocalIndex& local = index->local();
    local = maxLocal--;
  }

}
#endif
```

3 Remote Indices

4 Communication Interface

5 Communicator