The International Journal of Parallel, Emergent and Distributed Systems
Vol. 00, No. 00, August 2008, 1–10

RESEARCH ARTICLE

C++ Components Describing Parallel Domain Decomposition and Communication

Markus Blatt* and Peter Bastian

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen (IWR),
Ruprechts-Karls-Universität Heidelberg, Im Neuenheimer Feld 368, D-69120 Heidelberg,
Germany

(*received August 1, 2008*)

Large scale parallel codes require the data to be decomposed between the set of processes active in the computation. This data decomposition implies recurring communication schemes.

The paper introduces generic template classes in C++ for describing the data decomposition. The aim is to store the data in arbitrary existent efficient sequential data structures. Each entry in the sequential data structure corresponds to an entry in the virtual global view of the container. Once the decomposition is setup the needed communication schemes can be created automatically and can be used to communicate values from containers of various types. Even containers with a varying number of values associated with an entry are possible.

The framework abstracts the decomposition information and the communication in the client code from the eventual parallel paradigm choice. A prototype based on MPI is presented. It relieves the user from specifying information that is already known at compile time.

Keywords: domain decomposition methods, data parallel programming, object-oriented programming, generic programming, finite element methods, iterative solvers, MPI

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 sure that the data used for computations is valid, i.e. contains an updated value if

*Corresponding author. Email: Markus.Blatt@iwr.uni-heidelberg.de

ISSN: 1744-5760 print/ISSN 1744-5779 online

DOI: 10.1080/1744576YYxxxxxxxx
http://www.informaworld.com

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.

In Sections 2 to 3 of this paper we present software components that are able to describe this relation between the local data structures and global data distribution. These allow us to precompute the communication interfaces for synchronising arbitrary data. Thus they can be used to easily trigger synchronisation in parallel algorithms. Finally we will compare the performance of our approach to directly using MPI in Section 4.

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 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`).

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

```
// shortcut for index set type
typedef ParallelLocalIndex<Flags> LocalIndex;
typedef ParallelIndexSet<int, LocalIndex > PIndexSet;
PIndexSet sis;
sis.beginResize();
```

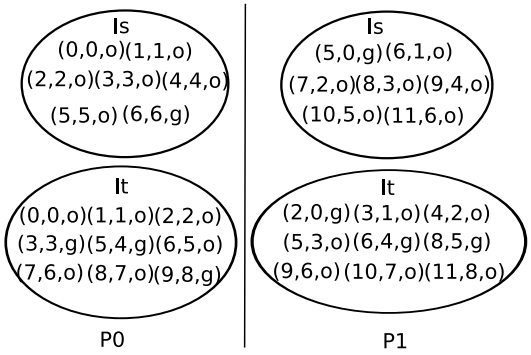


Figure 1. Index sets for array redistribution

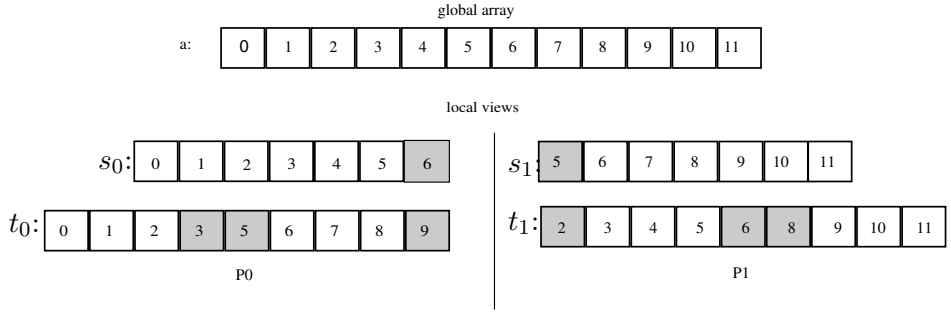


Figure 2. Redistributed array

```
for(int i=0; i<6; i++)
  sis.add(i, LocalIndex(i, owner));
sis.add(6, LocalIndex(6, ghost));
sis.endResize();
```

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

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

and

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

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 $r^s_{p\to q}$ and $r^t_{p\to q}$ 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

```
RemoteIndices<PIndexSet> riRedist(sis, tis, comm);
riRedist.rebuild<true>();
```

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

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

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:

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

Interface<PIndexSet> infRedist;
Interface<PIndexSet> infS;

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

```
BufferedCommunicator<PIndexSet> bComm;
BufferedCommunicator<PIndexSet> bCommRedist;
bComm.build(s, s, infS);
bCommRedist.build(s, t, infRedist);
```

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.

```
bCommRedist.forward<CopyData<Container> >(s,t);
 // calculate on the redistributed array
doCalculations(t);
bCommRedist.backward<AddData<Container> >(s,t);
```

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:

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

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

  static void scatter(T& v, double 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.

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* inout, int len, int root)`	Broadcast an array from root to all other processes
`template<typename T> int gather (T* in, T* out, int len, int root)`	Gather arrays at a root process
`template<typename BinaryFunction, typename Type> int allreduce(Type* in, Type* out, int len)`	Combine values from all processes on all processes. Combine function is given with `BinaryFunction`

Table 1. Collective Communication Functions

The collective communication classes were developed before the release of Boost.MPI [8]. 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 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

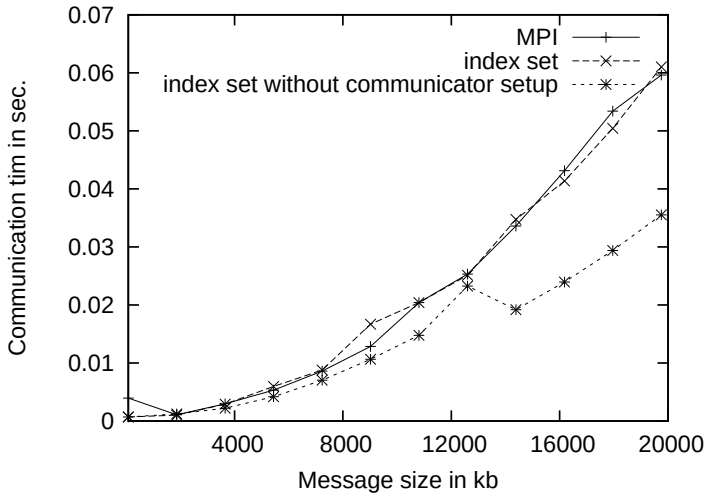
4. Performance Analysis

The performance of the library was compared to direct usage of MPI on the cluster "Helics II" consisting of 156 nodes with two dual core AMD Opteron 2220 2.8 GHz processors interconnected by a 10G Myrinet high speed interconnect.

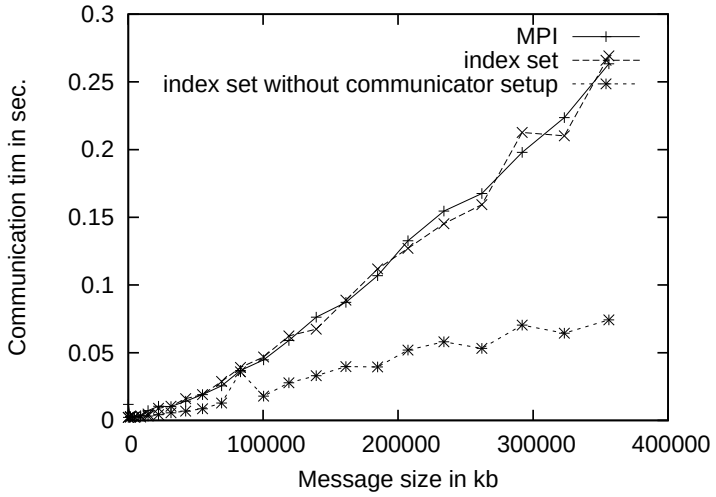
The test case simulates a parallel finite element computation on a structured parallel tensor product grid in two and three dimension, respectively, with one cell overlap. In each communication step all processes exchange data with their 4 and 6 neighours, respectively, in a forward communication. Now all cells in the ghost cells have consistent data. After this communication step the data at the ghost indices is consistent with the corresponding data owned by other processes. Now each process adds a random value to all data items and initiates a backward communication. In Figure 3 the average time for this operation is depicted for growing message sizes (implied by growing grids).

The version labelled "MPI" uses a custom `MPI_Datatype` based on `MPI_Type_hindexed` modelling our interface information on the sending and receiving side. The version labelled "index set" in the graphs uses the software components as described above. At each communication step the size of the buffers is calculated and the buffers are allocated. After the communication they are freed again. In the third version the buffers are allocated only once for each message size and reused at all communication steps.

We see that the presented approach poses no performance penalty on parallel code. In contrast due to the added flexibility of using persistent communication buffers it can even outperform raw MPI code.



(a) 2D example



(b) 3D example

Figure 3. Performance Test

5. Related Work and Conclusion

In contrast to the presented template based approach the PROMOTER programming model [7] is realised as a language extension to C++ together with a library which abstracts the communication schemes. The data partitioning and distribution is done at the language level. Unfortunately this does not allow adaptively changing or redistributing the data as needed for finite element computations on adaptively refined meshes.

The TACO (topologies and collections) framework [9] overcomes this problem. It uses global object pointers underneath and lets the user specify the data distribution at runtime using distributed linked objects. This allows dynamically adding new objects at runtime. Still this means that all parts of a simulation software, e.g. linear algebra and grids, need to either all use TACO directly or at least use the same data distribution. Especially when using third party software components resembling the data distribution with TACO might be cumbersome.

The Janus framework [6] follows a similar approach to the presented one. The basic abstraction is a mapping of a finite set of distributed objects onto consecutive global indices starting at 0. This abstraction is called a domain. In the parallel case

the domain is distributed onto p, the number of processes, mutually disjoint subdomains. In addition each object is mapped onto a local consecutive index starting at 0 on each process. This results in a strided distribution of the collection of objects and according global indices. Adaptively adding new objects calls for renumbering both local and global indices. Furthermore due to the mutually disjoint distribution all operations dependent on objects of other subdomains, e.g. sparse matrix vector products, require communication.

In the presented approach the subdomains need to be augmented to overlapping subdomains according to the data dependencies. This allows for minimising the communication phases. E.g. in Krylov solvers not every matrix vector product requires communication using this data distribution. Such overlapping subdomains are either provided by parallel grid managers or directly by the user. As in Janus each object is identified by a global index. This is mapped to a local index and an attribute used for identifying different partitions of the local domain and to set up communication. Once this mapping is set up the user can define communication interfaces based on the attributes and perform communications for arbitrary data types. Even if MPI is used underneath the user is relieved from directly using MPI routines and setting up communication by hand. Furthermore he can take advantage of the better performance offered by the presented approach if he has to deal with often recurring communication schemes.

The presented approach was used to implement the data-parallel iterative solver template library (ISTL module) [3, 4] of the distributed and unified numerics environment (DUNE) [1, 2, 5]. Currently it is used to develop an efficient parallel adaptive grid manager.

It is licenced under the GNU General Public License (GPL) with "runtime exception". Is available as part of the ISTL model of DUNE which is available for download from the DUNE project home page [5].

References

[1] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, *A generic grid interface for parallel and adaptive scientific computing. part i: abstract framework*, Computing 82 (2008), pp. 103–119.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander, *A generic grid interface for parallel and adaptive scientific computing. part ii: implementation and test in dune*, Computing 82 (2008), pp. 121–138.

[3] M. Blatt and P. Bastian, *The Iterative Solver Template Library*, in *Applied Parallel Computing. State of the Art in Scientific Computing*, Lecture Notes in Computer Science, Vol. 4699, Springer, 2007, pp. 666–675.

[4] M. Blatt and P. Bastian, *On the generic parallelisation of iterative solvers for the finite element method*, Int. J. Computational Science and Engineering 4 (2008), pp. 56–69.

[5] DUNE, `http://www.dune-project.org/`.

[6] J. Gerlach, *Domain engineering and generic programming for parallel scientific computing*, Ph.D. thesis, TU Berlin, 2002.

[7] W. Giloi, M. Kessler, and A. Schramm, *PROMOTER: A High Level, Object-Parallel Programming Language*, in *Proceedings of the International Conference on High Performance Computing*, December, New Dehli, India, 1995.

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

[9] J. Nolte, M. Sato, and Y. Ishikawa, *TACO – Dynamic Distributed Collections with Templates and Topologies*, in *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Springer-Verlag, London, UK, 2000, pp. 1071–1080.