Bergische Universität
Wuppertal

An MPI Extension for the Use of C-XSC in Parallel Environments

Markus Grimmer

Preprint 2005/3

Wissenschaftliches Rechnen/
Softwaretechnologie

Impressum

Herausgeber:	Prof. Dr. W. Krämer, Dr. W. Hofschuster Wissenschaftliches Rechnen/Softwaretechnologie Fachbereich 7 (Mathematik) Bergische Universität Wuppertal Gaußstr. 20 D-42097 Wuppertal

Internet-Zugriff

Die Berichte sind in elektronischer Form erhältlich über die World Wide Web Seiten

`http://www.math.uni-wuppertal.de/wrswt/literatur.html`

Autoren-Kontaktadresse

Markus Grimmer
Bergische Universität Wuppertal
Gaußstr. 20
D-42097 Wuppertal

E-mail: `grimmer@math.uni-wuppertal.de`

An MPI Extension for the Use of C-XSC in Parallel Environments

Markus Grimmer

Abstract. MPI is a common interface for communication in parallel environments. In this document, two strategies to apply MPI to user-defined data types are discussed. Subsequently, two implementations of an MPI extension for the data types contained in the C++ class library C-XSC are presented. Parallel environments and the necessary steps to use C-XSC programs with MPI communication routines in this environment are shortly described focussing on the parallel supercomputer ALiCEnext [1] and its practical access. Some test results for the new implementations are given. Finally, the development of an integral equation solver in C-XSC and its parallelization using the new MPI extension are discussed.

The package of MPI communication routines for C-XSC ist going to be further enhanced and more extensive tests are being done. The source code will be available at `http://www.math.uni-wuppertal.de/wrswt/xsc/cxsc_software.html`.

Keywords: Parallel computer, MPI, C-XSC, Interval Arithmetic, ALiCEnext.

1 Introduction

Interval arithmetic is more time consuming then real arithmetic: Computations have to be done on two bounding values in each step, and in most cases much more demanding considerations have to be made to efficiently implement an algorithm with result verification. Parallelization is a typical approach to solve large problems in reasonable time. It becomes more and more important as clock frequencies are reaching physical limits while performance requirements still rise.

MPI is a well-known interface for communication in parallel environments [14] [15]. It contains communication functions for both point-to-point communication and collective communication as well as facilities for the definition of data types and virtual topologies.

Writing parallel programs involving user-defined data types is a task that requires extra program code for the correct communication of data between processes. These parts of code are often individually written for each problem, since no parallel interface for the data types in use is available.

We present two versions of an MPI extension for the C-XSC data types comprising point-to-point communication routines based on the MPI mechanisms of data type definition and packing.

2 MPI Communication Routines for C-XSC data types

2.1 MPI Mechanisms to handle user-defined data types

Firstly, MPI facilities to deal with user defined data types shall be discussed.

MPI has a number of built-in data types that correspond to basic data types in C and Fortran (`MPI_CHAR`, `MPI_INT`, `MPI_DOUBLE`, etc.). (The MPI standard relates to *C* and *Fortran* implementations [14]. Accordingly, these program languages' sets of basic data types were chosen as a base for MPI data types. (Here, we shall refer to a C implementation of MPI functions. Also, C/C++ is considered the base language for any language specific statements in this text.)

Data Type Definition MPI includes a mechanism for defining new data types. Its main idea is the definition of a *type map* for the new data type.

A *type map* T is defined as

$$T := ((type_0, disp_0), ..., (type_{n-1}, disp_{n-1}))$$

where $type_i$ represents a basic (or a previously defined) MPI data type, $disp_i$ represents a displacement (given either in bytes or as a multiple of the size of a previously defined type) and $n \in I\!N$ fixed. The vector of data types $(type_i)_{0 \leq i < n}$ is then called *type signature*.

Given a type map, there is a set of data type definition functions with different degrees of freedom with respect to the storage arrangement of data in an object of the new data type.

- `MPI_Type_contiguous` allows for replication of contigously arranged objects of one data type.
- `MPI_Type_vector` and `MPI_Type_hvector` allow for partly contiguously arranged objects of one data type with regular displacements between contiguous blocks of objects.
- `MPI_Type_indexed` allows for non-contiguously arranged blocks of objects of one data type with varying block sizes.
- `MPI_Type_struct` finally allows for non-contiguously arranged blocks of objects of different data types, again with varying block sizes.

These cover a range of possible data type definitions.

Data type definition has two advantages: The benefit for the programmer ist that you can use the data type in all MPI communication routines just like the basic data types. At the same time there is no replication of data since all data is virtually mapped using the type map of the new data type. Memory requirements are thus kept low.

The main drawback of data type definition is that the type map of the data type has to be given at definition time, including the size and number of blocks involved. This implies that no types with variable size can be defined. For each object size that one

might want to communicate a new data type has to be defined. This particularly holds for C-XSC vectors and matrices of different length as well as for C-XSC multiple precision type objects.

Moreover, types that include dynamic storage allocation cannot be mapped properly since there's no fixed relation between the storage locations of the attributes and the location of the object itself - an individual type map for every object would be needed.

Packing and Unpacking The other facility for dealing with unknown data types is data *Packing* and *Unpacking*. Packing does not yield new data types and thus it doesn't offer a way to naturally deal with all communication functions at the same time. On the other hand, it allows all kinds of data to be prepared for communication in one turn.

Packing and unpacking is done via the MPI functions `MPI_Pack` and `MPI_Unpack`. `MPI_Pack` allows the user to incrementally specify data to be packed into one object buffer. Packed data can be arbitrary (as long as it can be described in terms of existing MPI data types), i.e. an arbitrary arrangement of objects of previously defined data types can be packed. Each piece of data involved can be packed by a separate call to `MPI_Pack`. Accordingly, data packed at some stage of the process can *depend* on the value of other data that has been processed earlier. This feature allows to pack data whose size is only specified within some other part of the data. Unpacking, in turn, allows to extract and process pieces of data from the received buffer separately. Packing, in contrast to data type definition, thus *does* allow to deal with vectors and matrices of different lengths.

Packed data objects can be communicated using the data type `MPI_Packed` which has been provided to handle buffers of packed data in the communication routines. During communication, it signalizes that the concerned part of data shall not be interpreted in terms of a predefined data type, but as a unit of packed data which has to be unpacked after communication. (Still, if packed data matches the signature of any previously defined data type, the user is allowed to interpret it in terms of this data type.) Technically, a packed unit is simply considered to be a storage buffer of a certain size, the latter being communicated together with the data itself. This implies that the result of separate packing transactions to a contiguous buffer may be considered a single packing unit, too, and a single packing unit can be unpacked by more than one unpacking transaction. This is of special importance with regard to collective communication. On the other hand, this also includes the main drawback of packing, namely the allocation of additional memory for the buffer.

A communication function for a particular data type can now be implemented first packing all data belonging to an object of the desired type into one packed unit and then calling the communication routine of interest.

Point-to-point communication

Outgoing communication:	Incoming communication:
• Pack data	• Receive data
• Send data	• Unpack data

The two necessary steps can either be combined in a single or be split up into two functions. Only the packing and unpacking requires special type dependent code and has to be implemented separately for each data type to be considered. Communication can be implemented as template functions that automatically call the appropriate packing and unpacking functions based on the type information from the template parameter.

Collective communication In collective communication, things are not that easy regarding the implementation of template communication functions for new data types. The general proceeding is given below. Taking in to account that outgoing and incoming data is handled by the same function, it has to be distinguished between sending and receiving processes within the function itself.

If process is sender
    For all data items that shall be sent
        Pack data
    Send data
else
    Receive data
    For all data items that were received
        Unpack data

But in contrast to point-to-point communication more individual problems have to be solved when there is more than one sending or receiving process:

- Some communication functions subdivide data into parts each of which is sent to or received by a separate process (e.g., a *scatter* operation sends equally sized parts of one object to a number of processes (one part each)). Depending on the object's structure, these parts cannot be identified automatically by the receiving process. Take, for example, a matrix storing two lower and upper bound values and a data array for its elements. Sending the whole matrix in packed format is easily possible, first packing the bounds into the buffer to be sent, then the data. Sending the buffer in blocks (without prior modification or addition of data) would send the bounds of the original matrix to the first receiving process only. All other processes would receive no information about the matrix size and thus might not interpret the data correctly.

- An even more sophisticated problem is that subdivision of data is not a straightforward process, but can follow various strategies that highly depend on the kind of problem to be solved or the (physical or virtual) topology of processes involved. In other words, subdivision of data cannot be automated or done by a one-fits-all subdivision function.

Both tasks involve *individual* adaptation.

It is possible to provide implementations for a particular subdivision strategy or for a particular way to interpret received data as part of an object, but these will not serve the purpose of a general interface.

There is one collective communication function in MPI which does not involve data subdivision and therefore can be implemented as a template without deciding for a data subdivisioni strategy first, which is `MPI_Bcast` that sends a complete data object to all processes.

All other important functions, especially the *scatter, gather, allgather* and *alltoall* functions [14] involve data subdivision (or assembly, respectively).

2.2 C-XSC Data Types and MPI

Now, the internal structure of data storage in C-XSC objects will be briefly described. For a more extensive description of C-XSC elements, see [6] [8] [9] [10].

Scalar Data Types First, there are the scalar data types `real`, `interval`, `complex` and `cinterval`. `interval` and `complex` objects contain two `real`s each, a `cinterval` contains two intervals. These are the only data types that can be implemented as MPI data types without restrictions since they don't have variable lengths and no dynamic memory is involved.

Vector and Matrix Data Types For every scalar data type there is an appropriate vector and matrix data type in C-XSC. These contain `int` values for *lower and upper index bounds* (i.e. one pair of bounds for vectors, two pairs for matrices), an extra element for the *size* in each dimension and a dynomically allocated C array of entries of the associated base type (i.e. `real` for an `rmatrix`, etc.) which is always a one dimensional array.

Multiple Precision Data Types Multiple precision data types in C-XSC are also known as *staggered* data types. A multiple precision variable is given as a vector of values of the base type. The value of the variable is determined by the exact sum of its components [10]. There are corresponding multiple precision types `l_real` to `l_imatrix` for all `real`-valued "short" (i.e. non-multiple precision) types, but only `l_complex` on the complex side.

Internally, components are stored as a C array of values of the base type. (Again, the array is dynamically allocated; the same holds for the data types given below.) The difference to vector types is that there is only one additional value for the length of the internal array (*"staggered precision"*) and no lower and upper bounds.

Dotprecision Data Types Dotprecision types have been designed to store the exact result of a *scalar product*, i.e. the exact sum of a number of products of scalar values. There are versions for all of the four base types. The size of a dotprecision object (or dotprecision *accumulator*), is system dependent.

If e_{min} and e_{max} contain the minimum and maximum values for the exponent in the base arithmetic, l denotes the mantissa length and g an implementation-dependent number of guard digits, then the length L of a *real* dotprecision accumulator (as described in [10]) is given by

$$L = g + 2e_{max} + 2|e_{min}| + 2l.$$

Internally, the storage of a dotprecision variable is represented by a C array of long integers, but the relevant value is only the length of the storage buffer itself (in bytes). This size is internally (but accessibly) computed and stored in the C-XSC variable `BUFFERSIZE`, which can be used with regard to MPI communication.

Remark When preparing objects of a data type for communication with MPI, it is important to understand which technical properties can be relied on.

For a matrix or vector as well as a multiple precision number, memory for the entries (or components) is allocated dynamically during construction of an object, while storage for the index bounds is not. Thus, memory occupied by an object is not necessarily contiguous and one object cannot necessarily be mapped to a buffer the same way as a second object of the same type and size. Bounds therefore have to be packed separately from the entries, while all entries (as components of one single dynamic array) can be packed in one turn.

Moreover, for vectors or matrices of more complex structures (e.g. of staggered values), it might even be necessary to pack all entries of a vector separately.

Still, packing allows to implement communication functions for a data type regardless of the size and memory structure of a particular object. Data type definition, though possibly yielding faster results due to the absence of memory replication, can only be applied to single objects. It should be considered for individual programming purposes, but does not serve to create MPI equivalents of most C-XSC data types.

2.3 Two implementations

Finally, the structure and functional range of two implementations shall be presented.

C-XSC Data Types Covered All C-XSC data types are covered [1]. For the basic types, new MPI data types have been defined:

C-XSC class	New MPI Data Type
`real`	`MPI_CXSC_REAL`
`interval`	`MPI_CXSC_INTERVAL`
`complex`	`MPI_CXSC_COMPLEX`
`cinterval`	`MPI_CXSC_CINTERVAL`

(From a technical point of view, MPI data types are values of the type `MPI_Datatype`.)

To define these new data types, the assumption has been made that attributes of objects of these C-XSC classes are stored *contiguously*.

A function

```
MPI_Define_CXSC_Types()
```

[1] except for the `int` versions of vectors and matrices which can be added

is included to define and *commit* [14] the data types. For all other C-XSC types, the packing mechanism is used, based on the types defined above.

The two available versions differ in the way that has been described in the foregoing chapter as well as in the number of implemented communication functions.

Implementation 1 The first version implements overloaded communication functions with integrated packing/unpacking.

The following communication functions are available:

- `MPI_Send`, `MPI_ISend`
- `MPI_Recv`

Here is a sample interface:

```
int MPI_Send (rvector&, int, int, MPI_Comm);
int MPI_Isend(rvector&, int, int, MPI_Comm, MPI_Request*);
int MPI_Recv (rvector&, int, int, MPI_Comm, MPI_Status* );
```

The corresponding implementation of `MPI_Send` and `MPI_Recv` is given below.

```
int MPI_Send(rvector& rv, int i1, int i2, MPI_Comm MC)
{
  int pos=0;
  char sendbuffer[MPI_CXSC_BUFFERLEN];

  int err;

  int lb=Lb(rv);
  int ub=Ub(rv);
  if (!MPI_CXSC_TYPES_DEFINED)
     if (err=MPI_Define_CXSC_Types()!=MPI_SUCCESS)
        return err;
  MPI_Pack(&lb,1,MPI_INT,sendbuffer,MPI_CXSC_BUFFERLEN,
           &pos,MC);
  MPI_Pack(&ub,1,MPI_INT,sendbuffer,MPI_CXSC_BUFFERLEN,
           &pos,MC);
  MPI_Pack(&rv[lb],ub-lb+1,MPI_CXSC_REAL,sendbuffer,
           MPI_CXSC_BUFFERLEN,&pos,MC);
  err=MPI_Send(sendbuffer, pos, MPI_PACKED, i1, i2, MC);
  return err;
}

int MPI_Recv(rvector& rv, int i1, int i2, MPI_Comm MC,
             MPI_Status* MS)
{
  int pos=0;
  char* recbuffer[MPI_CXSC_BUFFERLEN];
```

```
  int err;
  int vlen, lb, ub;

  if (!MPI_CXSC_TYPES_DEFINED)
     if (err=MPI_Define_CXSC_Types()!=MPI_SUCCESS)
        return err;

  if (err=MPI_Recv(recbuffer, MPI_CXSC_BUFFERLEN,
              MPI_PACKED, i1, i2, MC, MS)!= MPI_SUCCESS)
    return err;

  MPI_Unpack(recbuffer, MPI_CXSC_BUFFERLEN, &pos, &lb,
             1, MPI_INT, MC);
  MPI_Unpack(recbuffer, MPI_CXSC_BUFFERLEN, &pos, &ub,
             1, MPI_INT, MC);
  vlen=ub-lb+1;
  Resize(rv,vlen);
  SetLb(rv,lb);
  MPI_Unpack(recbuffer, MPI_CXSC_BUFFERLEN, &pos, &rv[lb],
             vlen, MPI_CXSC_REAL, MC);
  return err;
}
```

In cases where the bounds are known by both the sending and receiving processes, only the data might be sent eliminating the need for buffering. In terms of a general interface, though, this cannot be assumed. In contrast, assuming matching sizes can result in deficient memory management and thus in erroneous programs. A `Resize` operation is necessary if sizes differ.

Implementation 2 The same can be implemented as overloaded packing/unpacking and separate template communication functions. The packing/unpacking looks like in implementation 1 (without the send and receive, respectively). Sending, for example, only needs the following template function which is valid for all C-XSC types:

```
template<class T>
int MPI_Send(T& Tobj, int i1, int i2, MPI_Comm MC)
{
  int pos=0;
  char sendbuffer[MPI_CXSC_BUFFERLEN]; //Example length
  int err;

  if (err=MPI_Pack(Tobj, sendbuffer, MPI_CXSC_BUFFERLEN,
                   &pos, MC)!=MPI_SUCCESS)
    return err;
  err=MPI_Send(sendbuffer, pos, MPI_PACKED, i1, i2, MC);
  return err;
}
```

The MPI functions covered by the template versions are:

- `MPI_Pack` and `MPI_Unpack`
- `MPI_Send`, `MPI_Bsend`, `MPI_Ssend`, `MPI_Rsend`
- `MPI_Isend`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend`
- `MPI_Recv`
- `MPI_Bcast` (in preparation)

Though not required there are additional specializations for the C-XSC base types to avoid unnecessary calls to `MPI_Pack`/`MPI_Unpack`.

For the multiple precision types, the proceeding is more complicated. Apart from the additional information, data in an `l_real` variable looks much the same as in an `rvector`. A vector or matrix of multiple precision values, in contrast, has entries whose size can differ from entry to entry. Receiving an `l_rvector`, for example, does not guarantee that all entries have the same staggered precision. Moreover, all entries contain pointers to dynamically allocated memory. The only way to deal with this is to pack and unpack these elements separately.

Unfortunately, since there is no `Resize` operation for multiple precision types in C-XSC, the only way to take into account varying sizes is to construct new objects for the received data.

Usage The overloaded communication routines are used rather the same way as the original MPI routines. The following features are different from the MPI routines:

- The object to be packed or sent (respectively), has to be specified as a reference parameter and not as a pointer. This corresponds to the fact that C-XSC is a *C++* class library. Objects as instances of classes should not be manipultaed via pointers in order not to violate the integrity of the object. Even if MPI as a C function interface requires pointers for its communication functions, it is safer to hide the use of pointers from the user.
- In the MPI functions for C-XSC data types, the parameter for the number of objects to be processed is left out. In the original MPI functions, a parameter value of n can be used to specify a memory area n times as large as the memory occupied by the original object or variable, so that a vector of elements can be sent efficiently in one operation. However, this only works for contiguously stored objects. Since C-XSC objects (except for the base types which are also defined as MPI data types, see above) contain dynamically allocated memory, the described proceeding wouldn't succeed for a vector of C-XSC objects. Trying to process multiple objects at once would thus be rather misleading.
- The MPI data type parameter can be left out since the type of the object can be determined from the object reference itself.

The above applies to all C-XSC data types except for the base types `real`, `interval`, `complex` and `cinterval` which are implemented as MPI data types. For these, the user can choose between the original MPI interface and the new C-XSC/MPI interface.

An `rvector rv` can thus be sent to processor `dest` with message tag `tag` and communicator `comm` by

```
MPI_Send (rv, dest, tag, comm);
```

while a `real r` with identical communication envelope can be sent as

```
MPI_Send (r, dest, tag, comm);
```

or

```
MPI_Send (&r, 1, MPI_CXSC_REAL, dest, tag, comm);
```

(original interface).

3 Tests

The implementations were tested on the parallel computer ALiCEnext in Wuppertal [1]. Before giving some test results, this parallel environment will be introduced, including a short presentation of techniques and remarks towards practically running programs in such environments.

3.1 ALiCEnext - Parallel environment

ALiCEnext *(Advanced Linux Cluster Engine, next generation)* was installed at the University of Wuppertal in June 2004. It consists of 1024 1.8 GHz AMD Opteron processors (64 Bit architecture) on 512 nodes connected by Gigabit Ethernet connections [1]. At the time of installation, it held rank 74 in the world wide Top500 supercomputer list. In the LINPACK benchmark, it reached a maximum performance of 2083 GFlops [17].

ALiCEnext processors are run with Linux as operating system and MPI as communication interface for parallel communication (see above).

There's a C-XSC version available that works with the architecture of ALiCEnext processors [4].

Batch System Many parallel computers cannot be run interactively. There's a client machine that hosts the user's home directory and that acts as the entry point for the machine. Parallel jobs are started using a *batch system* that manages the distribution of system resources to the user's jobs. On ALiCEnext, as at July 2005, the batch system Torque *(Tera-scale Open-source Resource and Queue manager)* v1.2.0p4 (a version of the *Portable Batch System (PBS)*) is used [1] [18].

A script has to be written to correctly submit jobs through the batch system. Here's an example of how this has to be done:

```
#!/bin/bash

###################
# PBS options:

# The name of the queue to submit this job to

#PBS -q short

# Set name of file to which stderr will be redirected

#PBS -e cxsc_stagtst_r.err

# Set name of file to which stdout will be redirected

#PBS -o cxsc_stagtst_r.out

# Set the number of nodes that will be used.

#PBS -l nodes=10

# PBS will automatically create a file that lists all nodes
# allocated for running your job. The name and location of
# this file is stored in the $PBS_NODEFILE environmental variable.

# Count the number of processors PBS has allocated for our job
NPROCS=‘wc -l < $PBS_NODEFILE‘

################
# program

TARGETDIR=/home/user/progs
TARGETPROGRAM=mpiprogram

cd $TARGETDIR
./$TARGETPROGRAM -np $NPROCS
```

The PBS options given manage which *job queue* the job is sent to, where to direct the output and, most important, how many processors will be requested. The target program is then run using the actual number of processors provided by the batch system.

An I/O issue ALiCEnext, as many parallel computers, also has separate hard disks provided for each node. The user’s home directory, though, is located on the client or entry machine. It is considered important (and will effect the efficiency of the program as well) that output generated by a parallel program is written to a local file system and only copied to a central location (i.e. the home directory) at the end of program execution, so that the network load is not unnecessarily increased. One way to do this is to handle

file output inside the program you are running, since instances of the program are run on every processor involved.

Process Topologies There are several parameters that influence the effectiveness of a program in a parallel environment. One influencing factor that shall be mentioned here is the process topology. Depending on the location of processors in the physical topology of the machine, communication times can differ. In general, allocation of processors is done by the batch system and cannot be influenced directly - the user is not to interfere with the batch system's process distribution. Therefore, one has to take into account that the distribution of processes in different jobs can result in differing execution times.

MPI also implements a concept for defining topologies. These topologies, however, are virtual topologies that can be used to arrange the processes in a way that fits the needs of the application to be implemented. MPI topologies are machine independent and are not connected to the hardware topology.

3.2 Test programs and results

We now want to give the results of some tests that have been done.

Test 1 The test setting for this test is a communication of C-XSC objects around a ring of n processes, carried out k times. The following tests were done:

- Comparison for varying data types
- Comparison for varying numbers of processors
- Comparison for varying staggered precision values for multiple precision objects
- Comparison of template vs. non-template versions of communication routines

The tests have been done in order to take communication times where no time is consumed by the application. (A test application is given in the following section.)

Varying data types and numbers of processors

Processors	`real`	`interval`	`rvector`	`ivector`	`rmatrix`	`imatrix`
10	5030	5070	5240	5290	5650	6130
20	9330	9440	9680	9840	10450	11340
30	12060	14620	13520	15200	11440	12230
40	16060	17030	14890	16430	16140	13980

(Time in msec., 10000 passes, vector length 4, matrix dimension 4×4)

Most parts of the data show the expected growth of times with growing number of processors and growing size of data. There are some values, however, that were seemingly influenced by other factors. More extensive tests are done to analyze these influences.

Varying staggered precision values for multiple precision objects

Staggered precision	`real`	`interval`	`rvector`	`ivector`	`rmatrix`	`imatrix`
4	5320	6070	5900	7350	8040	10960
6	5300	6090	5990	7630	8280	11830
8	5310	6170	6110	7940	8550	12850
10	5310	6230	6140	8190	8830	14630

(Time in msec., 10000 passes, vector length 4, matrix dimension 4×4)

Template vs. non-template versions of communication routines

Version	`real`	`interval`	`rvector`	`ivector`	`rmatrix`	`imatrix`
Overloaded	5030	5070	5240	5290	5650	6130
Template	5139	5170	5320	5370	5720	6250

(10 processsors, time in msec., 10000 passes, vector length 4, matrix dimension 4×4)

The template versions cause a rather negligible constant overhead here.

4 An Application

We now introduce an application that is being developed making use of the new interface.

4.1 A verified Integral Equation Solver in C-XSC

Let $I\mathbb{R}$ be the set of real intervals where a real interval is a non-empty, closed and bounded subset of $\mathbb{R}$. The equation

$$y(s) - \lambda \int_a^b k(s,t)y(t)\,dt \;=\; g(s) \tag{1}$$

where $D := [a,b] \in I\mathbb{R}$, $k : D \times D \to I\mathbb{R} \in C(D \times D)$, $g : D \to I\mathbb{R} \in C(D)$, is called *Fredholm integral equation of the second kind* with kernel k and right hand side g.

If k has a representation

$$k(s,t) = \sum_{i=1}^{N} a_i(s)b_i(t)$$

with linearly independent functions a_i, b_i in the Banach space $C(D)$, then k is called *degenerate kernel of order N*. The integral equation (1) can then be written as

$$y(s) - \lambda \sum_{i=1}^{N} a_i(s) \int_a^b b_i(t)y(t)\,dt \;=\; g(s). \tag{2}$$

The kernel k can be written as

$$k = k_D + k_S$$

where k_D is degenerate. k, k_D and k_S have corresponding integral operators K, K_D and K_S and K_S satisfies $|\lambda| \, \|K_S\| < 1$ (with λ as in (1)) (which guarantees the existence of a unique solution of the equation; see [11] [12]).

In [11] [12], Klein proposes the following solver for Fredholm integral equations of the second kind.

Let $k = k_D + k_S$ as described above with corresponding integral operators K, K_D and K_S and $k_D(s,t) = \sum_{i=1}^{N} a_i(s) b_i(t)$ with enclosures A_i, B_i of the functions a_i, b_i of the degenerate kernel $(i = 1..N)$ and an enclosure G of the right hand side of the Fredholm integral equation as given in (1).

Execute the following iteration (until $F := F^{i+1} \subseteq F^i$ (or abort)):
$F^0 := G; \;\; F^{i+1} := G + K_S F^i, \;\; i = 0, 1, ...$
For $m := 1..N$
Execute the following iteration (until $\alpha_m := \alpha_m^{i+1} \subseteq \alpha_m^i$ (or abort)):
$\alpha_m^0 := A_m; \;\; \alpha_m^{i+1} := A_m + K_S \alpha_m^i, \;\; i = 0, 1, ...$
Solve the following interval linear system:
$\mathbf{X_m} - \lambda \sum_{n=1}^{N} \int_a^b B_m(t) \alpha_n(t) \, dt \, \mathbf{X_n} = \int_a^b B_m(t) F(t) \, dt, \;\; m = 1..N$
Compute solution interval $Y := F + \lambda \sum_{m=1}^{N} \alpha_m \mathbf{X_m}$

A C-XSC implementation of this solver is now available. The following components that are integrated in the implementation shall be mentioned.

Taylor Arithmetic with C-XSC To be able to calculate with elements of a function space like above, an appropriate representation and arithmetic are necessary. One way to do so for functions of one or two real variables (as the kernel and right hand side of the integral equation) is to use the Taylor expansions

$$g(x) = g(x_0) + g'(x_0)(x - x_0) + \frac{g''(x_0)}{2!}(x - x_0)^2 + ... + \frac{g^{(p)}(x_0)}{p!}(x - x_0)^p + R_p(x)$$

and

$$f(x,y) = \sum_{k_1=0}^{p} \sum_{k_2=0}^{p-k_1} \frac{1}{k_1! k_2!} \frac{\partial^{k_1+k_2} f(x_0, y_0)}{\partial x^{k_1} \partial y^{k_2}} (x - x_0)^{k_1} (y - y_0)^{k_2} + \tilde{R}(x,y)$$

of functions $g : x \to g(x) \in C(D)$ and $f : (x,y) \to f(x,y) \in C(D \times D)$, respectively, where R_p, $\tilde{R}_p$ are the remainders in terms of Taylor arithmetic.

The implementation of a verified Taylor expansion needs efficient computation of exact higher order derivative values of the considered function. *Automatic differentiation* methods serve this purpose (e.g. [5]). In [2] it is shown how to implement a Taylor

arithmetic including the exact and efficient computation of the necessary higher order derivatives.

M.C. Bräuer has provided a Taylor arithmetic in C-XSC in [3], comprising versions for one-dimensional, two-dimensional and multi-dimensional Taylor arithmetic. This implementation has been further developed as described in [2] and is available for current C++ compilers and the current C-XSC version. The versions for one-dimensional and two-dimensional real Taylor arithmetic have been provided with some additional functions so that they can be used in the integral equation solver as given above.

For parallelization, the MPI extension for C-XSC has been individually adapted to also include the Taylor expansion classes that are provided by the one-dimensional and two-dimensional C-XSC Taylor arithmetic.

Verified Linear System Solving with C-XSC Many interval linear system solvers available are based on a result by S.M. Rump. In [16] he shows for the Newton-like iteration

$$x^{(k+1)} = Rb + (I - RA)x^{(k)}, k = 0, 1, ...$$

to find a zero of $f(x) = Ax - b$ with arbitrary starting value $x^{(0)}$ and an approximate inverse $R \approx A^{-1}$ of A, that if there exists an index k with $[x]^{(k+1)} \overset{\circ}{\subset} [x]^{(k)}$ ($[x]^{(k+1)}$ included in the interior of $[x]^{(k)}$) , then the matrices R and A are regular, and there is a unique solution x of the system $Ax = b$ with $x \in [x]^{(k+1)}$ (also see [6]).

In [7], C. Hölbig and W. Krämer give an implementation in C-XSC based on an extension of this idea as described in [13]. This implementation was integrated into the integral equation solver mentioned above.

Parallelization The solver is currently being prepared for parallelization. There are three main parts to be parallelized:

1. The fixed point iterations in Klein's algorithm can be solved in parallel.
2. A system version of the integral equation solver that solves systems of integral equations (based on the same ideas) can make use of parallelization.
3. A parallellized linear system solver can be used to solve the interval linear system.

The now available MPI extension for C-XSC is used in and (as for Taylor arithmetic) was adapted to being used in the implementation of the above tasks.

References

[1] ALiCEnext information: http://www.alicenext.uni-wuppertal.de.

[2] Blomquist, F.; Hofschuster, W.; Krämer, W.: Real and Complex Taylor Arithmetic in C-XSC. Preprint BUW-WRSWT 2005/4, Wissenschaftliches Rechnen / Softwaretechnologie, University of Wuppertal, 2005

[3] Bräuer, M.C.: Berechnungsmethoden für Ableitungen und Steigungen und deren Realisierung in C-XSC. Diploma Thesis, University of Karlsruhe, 1999.

[4] C-XSC Versions and Download:
http://www.math.uni-wuppertal.de/wrswt/xsc/cxsc_new.html

[5] Griewank, A.; Corliss, G.: Automatic Differentiation of Algorithms, Theory, Implementation and Applications. Proceedings of Workshop on Automatic Differentiation at Breckenridge, SIAM, Philadelphia 1991.

[6] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D., C++ Toolbox for Verified Computing: Basic Numerical Problems. Springer-Verlag, Berlin / Heidelberg / New York, 1995.

[7] Hölbig, C.; Krämer, W.: Selfverifying Solvers for Dense Systems of Linear Equations Realized in C-XSC. Preprint BUW-WRSWT 2003/1, Wissenschaftliches Rechnen / Softwaretechnologie, University of Wuppertal, 2003. http://www.math.uni-wuppertal.de/wrswt/literatur/lit_prep.html.

[8] Hofschuster, W.; Krämer, W.; Wedner, S.; Wiethoff, A., C-XSC 2.0 – A C++ Class Library for Extended Scientific Computing. Preprint BUW-WRSWT 2001/1, Wissenschaftliches Rechnen / Softwaretechnologie, University of Wuppertal, 2001.
http://www.math.uni-wuppertal.de/wrswt/preprints/prep_01_1.pdf

[9] Hofschuster, W.; Krämer, W., C-XSC 2.0 – A C++ Class Library for Extended Scientific Computing. In: *Numerical Software with Result Verification*, R. Alt, A. Frommer, B. Kearfott, W. Luther (eds), Springer Lecture Notes in Computer Science 2991, 2004, pp. 15–35.

[10] Klatte, R.; Kulisch, U.; Lawo, C.; Rauch, M.; Wiethoff, A.: C-XSC, A C++ Class Library for Extended Scientific Computing. Springer - Verlag, Berlin / Heidelberg / New York, 1993.

[11] Klein, W.: Zur Einschließung von linearen und nichtlinearen Fredholmschen Integralgleichungssystemen zweiter Art. Dissertation, University of Karlsruhe, 1990.

[12] Klein, W.: Enclosure Methods for Linear and Nonlinear Systems of Fredholm Integral Equations of the Second Kind. In: Adams, E., Kulisch, U.: Scientific computing with automatic result verification, Academic Press, Boston 1993.

[13] Krämer, W., Kulisch, U., Lohner, R.: Numerical Toolbox for Verified Computing II. University of Karlsruhe, 1994 (Draft version).
http://www.math.uni-wuppertal.de/wrswt/literatur/pxsc_docu.html

[14] Message Passing Interface Forum: MPI: A Message Passing Interface Standard. University of Tennessee, Knoxville, Tennessee, 1993-1995.

[15] Message Passing Interface Forum: MPI-2: Extensions to the Message Passing Interface. University of Tennessee, Knoxville, Tennessee, 1995-1997.

[16] S.M. Rump: Kleine Fehlerschranken bei Matrixproblemen. Dissertation, University of Karlsruhe, 1980.

[17] Top500 list June2004: http://www.top500.org/lists/2004/06/

[18] Torque (Tera-scale Open-source Resource and Queue manager): http://www.clusterresources.com/products/torque/