



A Parallel \mathcal{H} -Matrix Implementation
v0.9.3

Developer Manual

by
Ronald Kriemann

Contents

1	Introduction	1
2	Installation	3
2.1	Prerequisites	3
2.2	Configuration	4
2.3	Compilation	6
2.4	Final Installation Step	7
3	Basic Datatypes, RTTI and Error Handling	9
3.1	Basic Datatypes	9
3.2	Global Variables and Functions	10
3.3	Runtime Type Information	10
3.4	Error Handling	11
4	Classes	15
4.1	Tools	15
4.1.1	Lists and Arrays	15
4.1.2	Smart Pointers	19
4.1.3	Classes for Parallel Environments	20
4.1.4	Bytestreams	32
4.1.5	Measuring Time and Memory	35
4.1.6	Miscellaneous	36
4.2	Clusters	40
4.2.1	Clustertrees	40
4.2.2	Construction of Cluster Trees	42
4.2.3	Block Clustertrees	45
4.2.4	Construction of Block Clustertrees	48
4.3	Matrices	51
4.3.1	TMatrix	51
4.3.2	Dense Matrices	55
4.3.3	Low Rank Matrices	56
4.3.4	Sparse Matrices	58
4.3.5	Blockmatrices	61

4.3.6	\mathcal{H} -Matrices	62
4.3.7	Domain Decomposition Matrices	64
4.3.8	Representing Inverse LU Decompositions	66
4.3.9	Permutation Matrices	66
4.4	Vectors	67
4.4.1	TVector	67
4.4.2	TScalarVector	69
4.4.3	Blockvectors	71
4.4.4	Vectors for Domain Decomposition	72
4.5	Algebra	73
4.5.1	Matrix Building	73
4.5.2	Matrix-Vector Multiplication	75
4.5.3	Matrix Addition	75
4.5.4	Matrix Multiplication	76
4.5.5	Matrix Inversion	77
4.5.6	LU and Cholesky Factorisation	77
4.5.7	Computing the Norm of a Matrix	78
4.6	Input and Output	79
4.6.1	Printer Classes	79
4.6.2	Cluster and Blockcluster I/O	84
4.6.3	Matrix Input and Output	85
	Bibliography	87

1 Introduction

PHI stands for *Parallel H-matrix Implementation* and hence implements parallel algorithm for \mathcal{H} -matrices or *hierarchical* matrices. This type of matrices, first introduced in [Hac99], provide a technique to represent various full matrices in a data-sparse format and furthermore, allow usual matrix arithmetics, e.g. matrix-vector multiplication, matrix multiplication and inversion, with almost linear complexity. Examples of matrices, which can be represented by \mathcal{H} -matrices stem from the area of partial differential or integral equations. For a detailed description of \mathcal{H} -matrices, please refer to [?].

PHI implements algorithms in the context of \mathcal{H} -matrices, e.g. building \mathcal{H} -matrices and \mathcal{H} -matrix-arithmetics. It does not provide algorithms for discretising differential or integral equations.

2 Installation

Before starting to install PHI you have to decide, which operational mode shall be used when working with the library. The first mode is for computer systems with a distributed memory, e.g. a network of workstations. The programming style for these kind of systems was chosen to be the *bulk synchronous parallel* or *BSP* machine.

Since a *BSP* machine is based on the idea of a computer system with a distributed memory, you have to choose the kind of communication between individual nodes. PHI provides four different types of communication:

MPI

Uses a *MPI* (message passing interface) library for send/receive operations.

PUB

Directly maps all *BSP* functions onto the **PUB** library.

SHM

Uses a *pipe*-based communication between different processes on a shared memory system.

SEQ

Support only sequential computations with one processor and no communication.

The fourth option exists for computer systems with only one processor, where no parallel computations are wanted.

If you have a shared memory system, e.g. a multiprocessor computer, you can alternatively use the *POSIX* thread interface to utilise more than one processor.

On a computer with just one processor, both programming modes can be used to work with PHI. With the *PThread mode* or *BSP mode* and *SHM* communication one can also simulate a parallel system.

2.1 Prerequisites

PHI was tested on a variety of operating systems and is known to work on the following environments

- Linux

- Solaris
- AIX
- HP-UX
- Darwin
- Tru64
- FreeBSD

In particular, each *POSIX* conforming system should be fine. The more crucial part plays der compiler. PHI demands a C++ compiler which closely follows the C++ standard. The following compiler versions are known to work

GCC	Version 3.4 and above, including 4.0.
Intel Compiler	Version 5 and above.
Portland Compiler	Version 2.x
Sun Forte Developer	Version 7 and above.
IBM VisualAge	Version 6
HP C++ compiler	Version 3.31 and above
Compaq C++	Version 6.5 and above

When using the *BSP mode* of PHI along with *MPI* or *PUB*, you also have to have the corresponding libraries installed on your system. In the same way, for the *PThread mode*, a version of the PThread library has to be installed. But the latter should already be included in all *POSIX* conforming systems.

To use the configuration system, a *perl* interpreter is needed and for the makefiles, *GNU make* is mandatory.

2.2 Configuration

If the right operation mode is chosen, the configuration system of PHI can be used to create all the makefiles needed to compile the package. For this, simply type

```
./configure
```

which tries to identify your operating system and compiler and chooses appropriate options for the compilation, e.g. include directives, libraries etc.. By default, the *PThread mode* will be used for PHI.

All available options for the configuration system can be printed by

```
./configure --help
```

which will result in the following list:

-prefix=dir

Set the prefix for installing PHI to the directory **dir**. By default, the local directory is chosen.

-objdir=dir

Define a prefix for all object files during compilation, e.g. **/tmp**. By default, object files will be created in the same directory, where the source files reside.

-phi-bsp=type

Configure PHI to compile in *BSP mode* with **type** defining one of the above mentioned communication modes: *MPI*, *PUB*, *SHM* or *SEQ*.

-phi-thr

Configure PHI to compile in *PThread mode*.

-cc=CC**-cxx=CXX**

Use **CC** and **CXX** as the C- and C++-compiler. By default appropriate settings for the considered operating system are used, e.g. **gcc** and **g++** for *Linux*.

-ld=LD**-ar=AR**

Set the dynamic linker to **LD** and the static linker to **AR**. By default, operating system dependent settings will be used.

-ranlib=RANLIB

Define the ranlib-command to bless static archives.

-enable-debug, -disable-debug**-enable-opt, -disable-opt****-enable-prof, -disable-prof**

Enable or disable compilation of PHI with debugging, optimisation or profiling flags. Any combination of the above is possible. By default, PHI will be compiled with debugging options.

-cflags=FLAGS**-cxxflags=FLAGS**

Define the compiler flags for C and C++ files.

-with-cpuflags[=PATH]

Enable the usage of **cpuflags** which will determine appropriate compiler flags for the combination of compiler, operating system and processor. The optional argument **PATH** defines the correct path to **cpuflags**.

-with[out]-x[=DIR]**-x11-cflags=FLAGS**

Turns on/off *X11* support in PHI. The optional argument **DIR** specifies the location of the *X11* environment, e.g. includes and libraries. Alternatively, you can define the compilation flags for *X11* directly.

A typical example for the usage of the configuration system might be

```
./configure --phi-thr --enable-opt --disable-debug --enable-cpuflags
```

which enables the *PThread mode*, an optimised compilation without debugging information and the usage of `cpuflags` to get the correct compiler flags.

If *MPI* or *PUB* were chosen as the communication layer, the definition of the C and the C++ compiler have to be changed to the appropriate programs of the corresponding implementations, e.g.

```
./configure --phi-bsp=MPI --cc=mpicc --cxx=mpic++
```

or

```
./configure --phi-bsp=PUB --cc=pubcc --cxx=pubc++
```

Otherwise, the compilation might result in an error indicating, that the include-files of the corresponding libraries could not be found.

Beside these PHI related options, the following parameters can be used to change the behaviour of the configuration system itself:

-c, -check

Turns on checking of used tools and libraries, e.g. if the C++ compiler is capable of producing object files.

-h, -help

Prints a detailed description of all options for the configuration system.

-q, -quiet

Do not print any information during configuration.

-s, -show

Just print the current options of the configuration system without creating makefiles.

-v, -verbose

Print additional information during configuration.

All settings which were changed by the user will be written to the file `config.cache`, where one can optionally edit the parameters with a text-editor.

2.3 Compilation

After PHI was configured, you can compile it by

```
make
```

Parallel compilation, e.g. via `-j`, is also supported.

After compiling, a library should reside in the `lib/` directory and an example on the usage of PHI named `phiex` should have been generated in the `example/` subdirectory.

2.4 Final Installation Step

If you have not chosen a specific installation directory, e.g. with the `--prefix` option, the installation is complete. Otherwise, you can install PHI by using

```
make install
```

which copies the PHI library and all include files to the corresponding directory.

It also copies the script `phi-config`, which was generated by `configure` to the `bin/` subdirectory. This script can be used to gather the correct compilation flags when including PHI in your projects. Options to `phi-config` are

- `--prefix`**
Returns to installation directory of PHI.
- `--version`**
Prints the version of PHI.
- `--lflags`**
Prints correct flags to links programs with PHI.
- `--cflags`**
Prints correct flags to compile programs with PHI.

So, to compile and link a program, one would use the following statement

```
CC -o program program.cc `phi-config --cflags` `phi-config --lflags`
```


3 Basic Datatypes, RTTI and Error Handling

This chapter describes the basic types and function upon which the classes of PHI are based. Of special importance are thereby the runtime type information system and the error handling.

3.1 Basic Datatypes

PHI defines some basic datatypes which are used in the library. They cover the usual integer and floating point types and are meant as an abbreviation.

uchar

represents an *unsigned character* with a range of $0 \dots 255$.

uint

represents an *unsigned integer* with a guaranteed range of $0 \dots 2^{32} - 1$.

ulong

represents an *unsigned long* with a guaranteed range of $0 \dots 2^{32} - 1$ on 32-bit systems and $0 \dots 2^{64} - 1$ on 64-bit systems.

real

Floating point type with *double* precision by default.

Furthermore comparison functions for general datatypes are defined:

min (x, y)

Returns the minimum of **x** and **y**.

max (x, y)

Returns the maximum of **x** and **y**.

Both procedures are implemented as generic functions with the type of **x** and **y** defined by a template parameter.

3.2 Global Variables and Functions

Although avoided if possible, some global variables and associated functions have been introduced in PHI. They cover settings for verbosity, e.g. how many messages are printed during computations, information about the parallel machine, a threadpool and a prefix for all written files.

Furthermore initialisation and finalisation of PHI is handled by global functions.

INIT (int * argc, char * argv);**

Initialise PHI, e.g. setup the parallel machine and global variables.

DONE ();

Finalise PHI, e.g. shut down the parallel machine.

set_verbosity (uint v);

Set the verbosity level in PHI.

bool verbose (uint level);

Return **true**, if the given level is less or equal to the verbosity level of PHI.

set_file_prefix (const char * prefix, ulong len);

Define the variable `file_prefix`, which is prepended to all filenames during IO operations in PHI.

3.3 Runtime Type Information

PHI implements it's own runtime type information (RTTI) system. Although, C++ already provides this feature, it has some drawbacks. First, it is, or at least was slow, e.g. during matrix multiplication millions of type comparisons are performed to decide the most efficient type of multiplication. The usage of the standard C++ RTTI led to a visible increase of the runtime.

The second reason, why the internal RTTI of PHI was preferred, is it's capability of printing typenamees. This makes error messages much more understandable.

Each class which uses the RTTI has to register it's type through the macro

```
DECLARE_TYPE( id, type )
```

where `id` is the name of the variable which will be used in PHI to access the typename, e.g. the *type-identifier*, and `type` is the name of the type as a string. An example for a matrix class would be

```
DECLARE_TYPE( TYPE_TDenseMatrix, 'TDenseMatrix' )
```

After this is declared, one can compare different types with the registered type-ids, e.g. with `TYPE_TDenseMatrix` in the example above. With the function

```
const char * id_to_str ( uint id );
```

it is possible to access the corresponding name of the type, e.g.

```
cout << id_to_str ( TYPE_TDenseMatrix ) << endl
```

yields

```
TDenseMatrix
```

By convention, all type identifier should be preceded with a `TYPE_` after which the original classname follows.

3.4 Error Handling

If possible, most of the functions in PHI return a code, indicating the status of the performed operation. If an error occurred inside a function, the internal error handling system is called, which, by default, prints a warning and logs the error. In most cases, the function then immediately returns, with this error code. The code itself is of type *error_t*.

The following list contains all error codes in PHI along with a description of that error:

<code>NO_ERROR</code>	No error occurred.
<code>ERR_TYPE</code>	Found an unknown or unsupported type.
<code>ERR_STRUCT</code>	Unexpected or unsupported structure, e.g. block-structure or matrix-dimension not compatible.
<code>ERR_NULL</code>	Found an unexpected NULL pointer.
<code>ERR_DIV_ZERO</code>	A division by zero was detected.
<code>ERR_INF</code>	Encountered ∞ floating point value.
<code>ERR_NAN</code>	Encountered <code>NAN</code> (not-a-number) floating point value.
<code>ERR_SPACE</code>	Not enough workspace was supplied.
<code>ERR_NOT_IMPL</code>	Functionality is not implemented.
<code>ERR_ARG</code>	Wrong argument in function call.
<code>ERR_MEM</code>	Insufficient memory available (no free RAM).
<code>ERR_SYSCALL</code>	Error during a call to a external function.
<code>ERR_RES</code>	A resource was not available.
<code>ERR_INIT</code>	A specific function/class was not initialised.
<code>ERR_COMM</code>	An error was detected during a communication.
<code>ERR_CONSISTENCY</code>	A general inconsistency was found.
<code>ERR_BS_SIZE</code>	No space left in bytestream for reading/writing.
<code>ERR_BS_WRITE</code>	Error while reading from bytestream.
<code>ERR_BS_READ</code>	Error while writing to bytestream.
<code>ERR_BS_TYPE</code>	Found object with wrong type in a bytestream.
<code>ERR_LAPACK_fn</code>	An error occurred in the LAPACK function “fn”.
<code>ERR_LAPACK_ARG</code>	Invalid argument for a LAPACK function.

The interface to the error handling system of PHI consists of the following functions:

```
error_t error ( error_t ec, const string & fname, const string & msg );  
    Report an error with code ec in function fname. The optional parameter msg  
    can hold a further description of the error.  
string sterror ( error_t ec );  
    Return a string containing a description for the given error code.  
string sterror ();  
    Return a string containing a description for current error code.  
void print_error_history ();  
    Print the error history.
```

The error history can be useful in case that an error occurred on a deep execution level, e.g. while handling a leaf in a block clustertree, and triggered several other error reports, overwriting the original code.

Although all functions return immediately after detecting an error, this might lead to undefined states in case of a parallel execution. Especially in the **BSP mode** with global

synchronisation, if one processor no longer handles communication, the global state of the BSP machine is undefined.

Also the return of error codes in case of a multi-threaded application, e.g. in **PThread mode** is not yet handled.

4 Classes

As PHI is written in C++, it also makes use of the object oriented features, this programming language provides. This means, most of the functionality in PHI is encapsulated in classes. Especially, the different matrix and vector types in the context of \mathcal{H} -matrices form a hierarchy of classes, which is mapped onto a class hierarchy in PHI. But also more common task, e.g. handling arrays and lists, measuring time or showing the progress of a computation, is put into the form of classes.

In this section all these classes in PHI are described in detail, whereby first the definition of each class is presented, followed by a description of each method.

4.1 Tools

The classes described in this section are not really related to \mathcal{H} -matrices but serve as basic building blocks, e.g. lists and dynamic arrays, or are useful for other purposes, e.g. timers and for memory measurement.

Some of these are already implemented in the *Standard Template Library* which is part of most C++ compilers. But since the implementation is not too difficult it was decided to avoid the complex overhead of this library. Furthermore this gives control over the complexity of some basic algorithms.

4.1.1 Lists and Arrays

Two of the most often used datatypes are lists and arrays. Both are implemented in PHI in the form of **TSLL** and **TArray** which will be discussed in the following sections.

4.1.1.1 Lists

The class **TSLL** implements a single linked list of objects. The type of object is defined by a template parameter to achieve the highest degree of flexibility. To access elements in the list, *iterators* are also part of the class.

The definition of the class is:

```
template <class T> class TSLL {
public:
    class TItem {
        T          _data;
```

```
    TItem * _next;
}

protected:
    TItem * _first, * _last;
    uint   _size;

public:
    TSSL ();
    TSSL ( const TSSL<T> & l );
    ~TSSL ();

    uint size () const;

    TIterator first ();
    TIterator last ();

    TSSL<T> & prepend ( const T & elem );
    TSSL<T> & append  ( const T & elem );

    void remove ( const T & elem );

    T behead ();

    void remove_all ();

    void copy ( const TSSL<T> & list );
}
```

An object of type `TSSL` holds a pointer to the first and last element in the list, along with the number of elements stored. Due to the storage of both ends of the list, it is possible to put new entries at the beginning and the end with constant costs. Each entry is enclosed in an object of type `TItem`, which also carries a pointer to the next element in the list to ensure connectivity.

The complete description of the mentioned methods of `TSSL` is as follows:

TSSL()

default constructor for `TSSL`, creates an empty list

TSSL(const TSSL<T> & l)

copy constructor for `TSSL`, copies all elements from `l` to the local list

TSSL()

destructor, removes all elements. Does not *delete* stored objects !

uint size ()

return number of items stored in list,

TIterator first ()**TIterator last ()**

return iterator to the first/last item in list,

```

TSSL<T> & prepend ( const T & elem )
TSSL<T> & append ( const T & elem )
    prepend/append item elem to the list,
void remove ( const T & elem )
    remove item elem from list,
T behead ( )
    remove and return first item from list,
void remove_all ( )
    remove all items from list
void copy ( const TSSL<T> & list );
    Copy all elements of list into local list.

```

The enclosed iterator for `TSSL` is defined as

```

class TSSL<T>::TIterator {
protected:
    TItem * _item;
public:
    TIterator ( const TIterator & iter );

    bool eol ( ) const;

    TIterator & operator ++ ( );
    TIterator operator ++ (int);

    T & operator ( ) ( );
}

```

with the methods

```

TIterator ( const TIterator & iter );
    constructs iterator pointing to iter,
bool eol ( ) const;
    return true if end of list was reached and false otherwise
TIterator & operator ++ ( );
TIterator operator ++ (int);
    prefix/postfix operator to go to next item in list and return current/next item,
T & operator ( ) ( );
    return current item in list the iterator points at.

```

A typical example on how to use `TSSL` is presented in the next example.

```
TSSL<int>          list;
TSSL<int>::ITerator iter;

list.append( 1 ).append( 2 ).append( 3 ).prepend( 5 );

for ( iter = list.first(); ! iter.eol(); ++iter )
  cout << iter() << ' ';
```

The output of this piece of code would then be:

```
5 1 2 3
```

4.1.1.2 Dynamic Arrays

Dynamic array are represented in PHI by objects of the class `TArray`. Similar to lists, the stored data is defined by a template parameter.

```
template <class T> class TArray {
protected:
  T    * _data;
  uint  _size;

public:
  TArray ( unsigned size );
  TArray ( unsigned size, const T & val );
  TArray ( const TArray<T> & vec );

  ~TArray ();

  uint  size      () const
  void  set_size ( uint n, bool copy );
  void  set_size ( uint n, const T & val, bool copy );

  TArray<T> & operator = ( const TArray<T> & vec );

  T & operator [] ( uint i );

  T * c_array ();
}
```

The following table contains the definition of each of the methods in `TArray`.

TArray (uint size);

TArray (uint size, const T & val);

TArray (const TArray<T> & vec);

Construct a dynamic array with a specific size (and default content `val`) or by copying the content of another array.

TArray ();

Destruct the array by removing the space needed to store the data. No `delete` is called if `T` equals a pointer to some data.

```
uint size () const;
```

```
error_t set_size ( uint n, bool copy );
```

```
error_t set_size ( uint n, const T & val, bool copy );
```

Get or set the size of the array. If `copy` is `true`, the content of the old array is copied into the new array. By default, `copy` is `false`. One can also define a default value for the new elements in the array by the parameter `val`.

```
T & operator [] ( uint i );
```

Access a single element of the array. If compiled with `-DDEBUG`, bound-checks are performed.

```
T * c_array ();
```

Return a pointer to a standard C-style array containing the data. This is actually the internal pointer and not a copy so do not delete !

4.1.2 Smart Pointers

Often data is shared between several objects. To control the number of references and guarantee a consistent lifetime of the data, smart pointers are a convenient programming mechanism. In PHI this is implemented in the class `TMultiRef`.

```
class TMultiRef {
protected:
    int _references;

public:
    TMultiRef ();
    TMultiRef ( const TMultiRef & );

    ~TMultiRef ();

    void add_ref ();
    bool del_ref ();

    bool free_ref ();
}
```

`TMultiRef` consists of a single integer holding the number of references and the corresponding methods to manage it:

```
TMultiRef ();
```

```
TMultiRef ( const TMultiRef & );
```

Constructs a smart pointer with 0 references. The copy constructor ensures that no reference numbers are copied.

```
TMultiRef ();
```

Warns if pending references to this object exists.

void add_ref ();

bool del_ref ();

Adds or removes a reference to this object. If no references exist, `del_ref` returns `true` and `false` otherwise.

bool free_ref ();

Removes a reference to the object. If the reference number is 0, it deletes the object (suicide mode) and returns `true`. Otherwise, `false` is returned.

4.1.3 Classes for Parallel Environments

In the following the functions and classes for handling parallel computations in PHI are described. As PHI works in two different modes, the function and classes also fall in two different areas: thread-based classes and BSP-based classes. These will be described in the following sections

4.1.3.1 TMachInfo

The basic information about the considered parallel machine is provided by objects of type `TMachInfo`. This covers the number of processors and the id of the local processor, whereby the latter only reflects the true processor id in the case of a BSP computation. In a multithreaded environment the id always equals 0, e.g. the id of the main process. The special id `NO_PROC` is used to describe the case that no specific processor is meant.

Since only a single object is needed to provide the mentioned information, PHI uses a global object of type `TMachInfo` named `mach_info`.

```
class TMachInfo {
protected:
    uint _nprocs;
    uint _pid;

public:
    TMachInfo ();
    ~TMachInfo ();

    uint pid          () const;
    uint nprocs       () const;
    bool is_parallel  () const;

    void set_pid      ( uint n );
    void set_nprocs   ( uint n );

    bool on_proc      ( uint p ) const;
}
```

Beside the mentioned data, `TMachInfo` also provides functions for related information, e.g. if whether the program works in parallel or if a given processor is equal to the local processor.

TMachInfo ();

TMachInfo ();

Construct and destroy objects of type **TMachInfo**.

uint pid () const;

uint nprocs () const;

Return the local processor id and the total number of processors in the parallel machine.

bool is_parallel () const;

Return **true** if more than one processor is used.

void set_pid (uint n);

void set_nprocs (uint n);

Set the corresponding data about the local id and the number of processors.

bool on_proc (uint p) const;

Return true if the given processor id **p** is equal to the local id. Identifiers equal to **NO_PROC** are also considered local.

4.1.3.2 Processor Sets

Sets of (continuously numbered) processors are represented in PHI by objects of type **TProcSet**. They provide functions for testing, whether a specific processor is in the set or for splitting the set into subsets. Furthermore, each set has a unique *master* processor, which can be used for certain parallel tasks (see **TStreamable**).

```
class TProcSet {
protected:
    uint _first, _last;

public:
    TProcSet ( uint size );
    TProcSet ( uint first, uint last );
    TProcSet ( const TProcSet & ps );
    ~TProcSet ();

    uint first () const;
    uint last () const;
    uint size () const;
    uint master () const;

    TProcSet set ( uint i, uint n ) const;

    void split ( uint n, TArray< TProcSet > & psets ) const;
    bool is_in ( uint p ) const;
}
```

The set is defined by the id of the first and the last processor. Hence, only continuously numbered sets are supported.

TProcSet (uint size);

Create a processor set with **size** processors. The first processor has id 0, whereas the last processor has id **size-1**.

TProcSet (uint first, uint last);

Create a processor set defined by the interval **first** and **last**.

TProcSet (const TProcSet & ps);

Create a processor set by copying the given set **ps**.

TProcSet ();

Destroy a processor set.

uint first () const;

uint last () const;

Return the id of the first and last processor in the set.

uint size () const;

Return the size of the processor set.

uint master () const;

Return the id of the master processor in the set. Currently, this equals the first processor.

TProcSet set (uint i, uint n) const;

Return the **i**'th subset in a partition of the set into **n** subsets.

void split (uint n, TArray<TProcSet> & psets) const;

Split the processor set in **n** (more or less equally sized) subsets and put the partition into **psets**.

bool is_in (uint p) const;

Return **true** if the given processor id is contained in the local processor set.

4.1.3.3 Thread Classes

When working on a shared memory system the problem of using multiple processors with this shared memory must be solved. In PHI this is done by using the *POSIX-Thread* (or *PThread* for short) interface. It provides functions for starting and synchronising threads, defining scheduling strategies and controlling access to certain data or functions.

Unfortunately, this interface sometimes is too common and therefore too complicated for many typical situations. To simplify the thread access, a *thread pool* is used in PHI to work in parallel.

The wrapping of the PThread functions and the implementation of the thread pool is described in the following sections.

Because of the dependence of most functions a PThread library on the computer system, the actual implementation is only compiled, if the **PThread mode** was chosen

during the installation. Otherwise, most functions immediately return without doing something. Therefore, the usage of the following classes should be restricted to this case.

Threads

The class `TThread` provides a wrapper for the most common PThread functions for handling threads, e.g. for starting, joining and killing threads.

```
class TThread {
protected:
    bool _running;
    int  _thread_no;

public:
    TThread ( int thread_no );

    ~TThread ();

    int thread_no () const;
    int proc_no   () const;

    void set_thread_no ( int n );
    bool on_proc ( int p ) const;

    virtual void run () = 0;

    virtual error_t start ( bool detached, bool sscope );
    virtual error_t stop ();

    error_t create ( bool detached, bool sscope );
    error_t detach ();
    error_t join   ();
    error_t exit   ();
    error_t cancel ();
    error_t kill   ( int signo );
    error_t sleep  ( long m_sec, long nano_sec );
}

```

The interface of `TThread` is similar to that of threads in Java. It contains a `run` method, which contains the code to execute and two methods to start and stop the thread.

In addition to that, each object of type `TThread` also has a field `_thread_no` which can hold an identifier to distinguish between the parallel threads, e.g. if the same code is executed with different data on multiple processors.

TThread (int thread_no);

Construct a thread object with id `thread_no`. The thread is not started. By default, the identifier is -1.

TThread ();

Deletes a thread object. A running thread is canceled, i.e. terminated if a cancelation point is reached.

int thread_no () const;

void set_thread_no (int n);

bool on_proc (int p) const;

Get/set thread identifier and determine, if the thread with id `p` is equal to the current thread. If `p` equals -1, `true` is returned.

virtual void run () = 0;

Method containing the actual code to execute in parallel. Since it is a pure virtual method, it has to be overwritten in derived classes.

virtual error_t start (bool detached, bool sscope);

virtual error_t stop ();

Start or stop execution of a thread. If `detached` is `true`, the thread is detached from the calling thread, i.e. it can't be joined anymore. The scheduling strategy is determined by `sscope`. If this parameter is true, the thread is started in *system contention scope* in which the scheduling is done by kernel functions. In the case that `sscope` equals `false`, all scheduling is done by user mode functions. Keep in mind, that this is dependent on the actual PThread implementation.

error_t create (bool detached, bool sscope);

Synonym for `start` and corresponds to the PThread function `pthread_create`.

error_t detach ();

Detaches a thread from the calling thread which prohibits future synchronisation with the thread.

error_t join ();

Synchronises with the termination of the thread, i.e. finishes only when the thread finishes.

error_t exit ();

Terminates the calling (!) thread (called from inside `run`).

error_t cancel ();

Terminate the thread if some cancelation point is reached.

error_t kill (int signo);

Immediately terminate the thread by sending signal `signo`.

error_t sleep (long m_sec, long nano_sec);

Pause execution of the calling (!) thread by specified amount of time.

Mutices

To enable the synchronisation of several threads when it comes to common data or critical sections inside a program, a *mutex* is the typical answer. In PHI this is done by using an object of type `TMutex`.

```
class TMutex {
public:
    TMutex ();
    ~TMutex ();

    int lock ();
    int unlock ();

    bool is_locked ();
}
```

The interface is restricted to switching between the two possible states of a mutex: *locked* and *unlocked*.

TMutex ();

TMutex ();

Construct and destruct a mutex. The mutex is unlocked after construction.

int lock ();

int unlock ();

Lock or unlock a mutex. If the mutex is locked, the call to `lock` blocks, until it is unlocked by another thread. The semantics of unlocking an already unlocked mutex is not defined by the PThread standard. Both function return 0 on success and the corresponding error code at failure.

bool is_locked ();

Returns `true` if the mutex is locked and `false` otherwise.

Condition Variables

A *condition variable* is a signaling mechanism between several threads which enables the waking of more than one thread as it is the case with a mutex. A typical example is a set of threads reading from a queue and blocking whenever the queue is empty. If items are inserted to the queue, several threads can be signaled to start reading.

```
class TCondition {
public:
    TCondition ();
    ~TCondition ();

    void wait      ( TMutex & m );
    void signal    ();
    void signal    ( TMutex & m );
    void broadcast ( TMutex & m );
    void broadcast ();
}
```

A condition variable is associated with a mutex, hence the argument to some of the methods.

TCondition ();

TCondition ();

Construct and destroy a condition variable.

void wait (TMutex & m);

Wait for the condition to arrive. The mutex `m` is locked before entering the wait state and `unlocked` after leaving it.

void signal ();

void signal (TMutex & m);

Signal one thread waiting on a condition variable that the corresponding condition has occurred. If a mutex is supplied it is locked before and unlocked after signaling.

void broadcast ();

void broadcast (TMutex & m);

In contrast to `signal` all threads waiting on a condition variable are signaled.

A Thread Pool

In many situations one does not want to deal with calling PThread functions directly, especially if more complex features are wished. For this case a *thread pool* was implemented in PHI which is initialised at the start of the program with a *fixed* number of threads. Each task which should be executed in parallel is given to the thread pool in the form of a `TJob` object which is a subclass of the actual thread pool class `TThreadPool`.

```
class TThreadPool {
protected:
    uint _max_parallel;

public:
    TThreadPool ( uint max_p );
    ~TThreadPool ();

    void run ( TJob * job, void * arg, bool del );
    void sync ( TJob * job );

    void sync_all ();
}
```

Compared to the PThread standard, the interface of the thread pool consists just of three functions: `run`, `sync` and `sync_all`. The variable `_max_parallel` defines the degree of parallelity the thread pool has, i.e. the maximal number of concurrent threads.

TThreadPool (uint max_p);

TThreadPool ();

Create or destroy a thread pool with `max_p` threads. At destruction, a synchronisation with all running threads is done before deleting them.

```
void run ( TJob * job, void * arg, bool del );
```

Run the task `job` in the thread pool. If no idle thread is available, the method blocks until another job has finished. Additional arguments to the job object are supplied by the parameter `arg`. If `del` is `true`, the corresponding job is deleted after it has finished. The default values of `arg` and `del` are `NULL` and `false` respectively.

```
void sync ( TJob * job );
```

Block until the task `job` has finished.

```
void sync_all ();
```

Block until no job is executed by the thread pool, i.e. all jobs have finished.

The interface to the job class is similar to a `TThread` object:

```
class TThreadPool::TJob {
protected:
    uint _job_no;

public:
    TJob ( uint n );

    void run ( void * arg ) = 0;

    uint job_no () const;
    bool on_proc ( uint p ) const;
}
```

`TJob` is an abstract class from which real “jobs” have to be derived. Beside the interface to the `run` method, `TJob` implemented methods for identifying the current job (or processor).

```
TJob ( uint n );
```

Construct job object with job identifier set to `n`, which has a default value of -1.

```
void run ( void * arg ) = 0;
```

Pure virtual method for executing the actual code. The value of the parameter `arg` equals the corresponding parameter in the `run` method of the thread pool.

```
uint job_no () const;
```

Returns the job identifier of the object.

```
bool on_proc ( uint p ) const;
```

Returns `true` if the argument `p` equals the job identifier or one of both is -1.

Due to the different synchronisation methods in `TThreadPool` two kinds of using the pool can be used: *online scheduling* and *offline scheduling*. Online scheduling, here in

the form of *list scheduling*, corresponds to starting an arbitrary number of jobs and synchronising with *all* of them at a specific point in the future. The load balancing is done automatically by the thread pool by executing the next job in the list on the first idle processor. The following listing shows an example for this programming style.

```
TThreadPool * thread_pool;

void f ( int l ) {
    if ( l == 0 ) thread_pool->run( new TMyJob() );
    else          { f( l-1 ); f( l-1 ); }
}

void main () {
    thread_pool = new TThreadPool( p );
    f( max_depth );
    thread_pool->sync_all();
}
```

Here a computation follows a tree-like structure, where at the leaves the real work is done. After the tree is completely traversed and all jobs (of type `TMyJob`) have been started, the method `sync_all` is used to synchronise with the termination of all threads.

Remark

In the example it is assumed, that all job-objects are deleted by some explicit mechanism. For automatic deletion one has to change the corresponding code line to

```
if ( l == 0 ) thread_pool->run( new TMyJob(), NULL, true );
```

An explicit scheduling, i.e. one with a precomputed load balancing, corresponds to offline scheduling. Here the degree of parallelism is set by the programmer and usually equals the number of threads in the pool. An example for using a `TThreadPool` object with this technique would be:

```
TThreadPool  thread_pool( p );
TMyJob      ** jobs = new TMyJob* [p];

for ( int i = 0; i < p; i++ ) jobs[i] = new TMyJob( i );
for ( int i = 0; i < p; i++ ) thread_pool.run( jobs[i] );
for ( int i = 0; i < p; i++ ) thread_pool.sync( jobs[i] );
for ( int i = 0; i < p; i++ ) delete jobs[i];

delete[] jobs;
```

Each job is created, run and synchronised at the same time. Note that, since `p` equals `max_parallel` in the thread pool, a call to `sync_all` would also be sufficient for a synchronisation.

In both examples a local thread pool is used to handle the jobs. PHI also provides a global thread pool, which is also used by most internal algorithms. To access this pool, the above functions are decoupled from the actual object:

```

error_t tp_init ( uint p );
void tp_run ( TThreadPool::TJob * job, void * ptr = NULL, bool del = false );
void tp_sync ( TThreadPool::TJob * job );
void tp_sync_all ();
void tp_done ();

```

The functions `tp_run`, `tp_sync` and `tp_sync_all` behave exactly like the corresponding methods of `TThreadPool`. The respective functions for initialisation and destruction of the pool are `tp_init` and `tp_done`, which are also called from the initialisation and finalisation functions of PHI.

Using this interface, the examples above would be:

```

void f ( int l ) {
    if ( l == 0 ) tp_run( new TMyJob() );
    else        { f( l-1 ); f( l-1 ); }
}

void main () {
    tp_init( p );
    f( max_depth );
    tp_sync_all();
}

```

and

```

TMyJob ** jobs = new TMyJob* [p];

tp_init( p );
for ( int i = 0; i < p; i++ ) jobs[i] = new TMyJob( i );
for ( int i = 0; i < p; i++ ) tp_run( jobs[i] );
for ( int i = 0; i < p; i++ ) tp_sync( jobs[i] );
for ( int i = 0; i < p; i++ ) delete jobs[i];

delete[] jobs;

```

4.1.3.4 BSP functions

Functions for using PHI if compiled with the **BSP mode** are described in this section. In contrast to most other algorithms, they are not encapsulated into a class but in a namespace since you would end up with exactly one BSP object. The corresponding namespace is *BSP*.

The functions provide an interface for the communication between different processors by sending and receiving messages or doing a global synchronisation.

```

namespace BSP {
    error_t init ( int * argc, char *** argv );
    error_t done ();

    uint nprocs ();
    uint pid   ();
}

```

```
error_t sync ();

error_t send ( uint dest, const void * buf, size_t bsize );
error_t hpsend ( uint dest, const void * buf, size_t bsize );

TMessage get_msg ();

unsigned int nmsgs ();

error_t dsend ( uint dest, const void * buf, size_t bsize );
error_t drecv ( uint source, void * buf, size_t bsize );
size_t dprobe ( uint source );

void statistics ();
}
```

Beside the BSP style communication with a non-blocking send, a global synchronisation, during which the data is transmitted, and finally receiving the message, encapsulated in a **TMessage** object, PHI also offers a direct communication between two processes. Note, that this communication is blocking and might lead to dead-locks. It also violates the BSP standard, but was included since it is more appropriate in some circumstances.

```
error_t init ( int * argc, char *** argv );  
error_t done ();
```

Initialise and finish a BSP computation. The parameters for the initialisation must be equal to the command line arguments of the **main** function of the program.

```
uint nprocs ();  
uint pid ();
```

Return the number of processors in the BSP computer or the id of the local processor.

```
error_t sync ();
```

Perform a global synchronisation between all processors. After returning from the function, all messages have been sent and received.

```
error_t send ( uint dest, const void * buf, size_t bsize );  
error_t hpsend ( uint dest, const void * buf, size_t bsize );
```

Send message contained in the buffer **buf** of size **bsize** to the processor **dest**. If the content of the buffer is not changed before the next synchronisation, the function **hpsend** can be used. Otherwise **send** is recommended, which copies the content before returning to the program.

```
TMessage get_msg ();
```

Returns the next message in the queue of received messages. The returned message is removed from the queue.

unsigned int nmsgs ();

Returns the number of messages still present in the receive-queue.

error_t dsend (uint dest, const void * buf, size_t bsize);

Send the content of **buf** of size **bsize** to processor **dest**. The communication is blocking, e.g. is only finished, after the message is received.

error_t drecv (uint source, void * buf, size_t bsize);

Receive a message from processor **source** and copy the content of size **bsize** into **buf**.

size_t dprobe (uint source);

Return the size of the message, which was directly sent by processor **source**.

void statistics ();

Prints statistics of the BSP computer, e.g. number of bytes transmitted or how many steps (synchronisations) have been done.

The class **TMessage** contains all information about received messages, e.g. the processor number of the sender or the size of the message. Since this is dependent on the actual BSP implementation, the public interface is restricted to the following methods.

```
class TMessage {
public:
    TMessage ( const TMessage & m );
    ~TMessage ()

    void * data () const;
    size_t size () const;
    uint source () const;

    TMessage & operator = ( const TMessage & m )
}

```

The constructor, destructor and the copy operator are implemented for a correct handling of messages in the program. The remaining methods give access to the interesting data.

void * data () const;

Return a pointer to a buffer containing the actual data of the message. At the next synchronisation, this buffer is freed. Hence the content has to be copied if it is need after that.

size_t size () const;

Returns the size of the received data.

uint source () const;

Returns the id of the processor which has sent the corresponding message.

4.1.4 Bytestreams

Matrices, vectors and other data has to be transmitted over a network in a special way to preserve the internal structures, e.g. pointers can not be send directly. The type of object to handle this comes in the form of a *bytestream*. As the name indicates, a bytestream represents a single stream of bytes and provides functions to read and write data to this stream. In PHI, this is handled by objects of the type `TByteStream`.

Beside this basic functionality, each object should be able to read and write itself, e.g. the internal data, from and to streams. To provide a common interface for such methods, the class `TStreamable` is introduced.

Please not, the sending and receiving data over networks is only needed when working in the BSP mode of PHI. In the case of a multithreaded environment, an additionaly property of bytestreams can be used, namely writing the streams to a hard disk. Of course, this is also possible in the BSP mode.

4.1.4.1 TByteStream

The actual bytestream is implemented as an array of bytes, where the array is represented by an *internal* or an *external* memory block. In the latter case, no memory is allocated or freed when initialising or finishing the bytestream-object.

```
class TByteStream {
public:
    TByteStream ( ulong s = 0 );
    TByteStream ( const TByteStream & str );
    ~TByteStream ();

    ulong position () const;
    ulong size      () const;

    TByteStream & set_size ( ulong n );
    TByteStream & set_pos  ( ulong p );

    TByteStream & set_stream ( void * data, ulong size );
    TByteStream & copy_stream ( void * data, ulong size );

    TByteStream & to_start ();
    TByteStream & to_end   ();

    uchar *      data ();
    const uchar * data () const;

    error_t put ( const void * buf, ulong n );
    error_t get (          void * buf, ulong n );

    error_t save ( const char * filename ) const;
    error_t load ( const char * filename );

    void print ( bool show_content = false ) const;
}
```

The interface provides basic methods to initialise a bytestream of a particular size and to read/write data from/to the stream.

TByteStream (ulong s = 0);

TByteStream (const TByteStream & str);

Initialise the bytestream to be of size **s** or by the provided stream.

TByteStream ();

Free internal data if the bytestream is not external.

ulong position () const;

ulong size () const;

TByteStream & set_size (ulong n);

TByteStream & set_pos (ulong p);

Get and set current position in the stream or the total size of the bytestream.

TByteStream & set_stream (void * data, ulong size);

Set the bytestream to the data provided by **data** and **size**. The stream is considered to be external, i.e. the memory is not freed.

TByteStream & copy_stream (void * data, ulong size);

Create an internal stream of size **size** and copy the content from **data**.

TByteStream & to_start ();

TByteStream & to_end ();

Set the current position of the stream to the beginning or the end of the stream.

uchar * data ();

const uchar * data () const;

Directly access the internal data-pointer to the stream.

error_t put (const void * buf, ulong n);

Write **n** bytes of the content pointed to by **buf** to the bytestream at the current position. The position pointer is also increased by **n**. If not enough space is left in the stream, **false** is returned. Otherwise **true** is the result of the function.

error_t get (void * buf, ulong n);

Copy **n** bytes from the current position in the bytestream to **buf**.

error_t save (const char * filename) const;

Write the whole content of the bytestream to the file indicated by **filename**.

error_t load (const char * filename);

Read the content of **filename** into the bytestream, which is initialised to hold enough data.

void print (bool show_content = false) const;

Print some debug info about the content of the bytestream. If `show_content` equals `false`, a checksum is printed. Otherwise also the content of the stream is written.

4.1.4.2 TStreamable

A bytestream gives us the basic functionality to put general data into streams, which can then be send over networks or written to the harddisk. To use this capabilities, `TStreamable` defines the interface for all objects, which can be read from or written to streams. Additionally, it provides communication functions for parallel data, e.g. to scatter or gather data to or from different processors.

```
class TStreamable {
public:
    TStreamable ();
    ~TStreamable ();

    error_t read ( TByteStream & s );
    error_t write ( TByteStream & s ) const;

    ulong bs_size () const;

    error_t scatter ( const TProcSet & p,
                    uint          pid,
                    TByteStream * bs );
    error_t sum     ( const TProcSet & p,
                    uint          pid,
                    uint          nparts,
                    TByteStream * bs );
}
```

The definition of the above listed methods is:

TStreamable ();

TStreamable ();

Default constructor and destructor for streamable objects.

error_t read (TByteStream & s);

error_t write (TByteStream & s) const;

Read the object in the bytestream `s` into the local object or write the local object into `s`.

ulong bs_size () const;

Return the size of the object, needed in the bytestream.

error_t scatter (const TProcSet & p, uint pid, TByteStream * bs);

Send the object located at the master of the processorset `p` to all processors in the group. If `bs` is not `NULL`, it will be used during the communication. In that case, it is presumed, that `bs` already contains the correct data at the master processor.

```
error_t sum ( const TProcSet & p, uint pid, uint nparts, TByteStream
* bs );
```

Sum up all `nparts` parallel versions of the object. The final result will be available at the master of the local group `p`. Note that `nparts` does not necessarily equal the size of the processor group. Again, if `bs` is not `NULL`, it will be used for encapsulating the data.

In contrast to `scatter`, which is already implemented in `TStreamable`, `sum` has to be overloaded and implemented in derived classes.

4.1.5 Measuring Time and Memory

One of the basic routines in the context of numerical computations is the measuring of time, e.g. to examine the complexity of algorithms. In PHI various types of time can be measured.

REAL_TIME

The *realtime* refers to the wallclock time, i.e. the total time between to measurements. Since it is dependent on the workload, it does not necessarily reflect the complexity of algorithms, e.g. on a fully loaded machine. Unfortunately, in the threaded mode of PHI one usually can only measure this, since no calls for a specific thread are possible.

CPU_TIME

This is the actual time the process or thread needed for a computation independent of the load of the computer system.

USER_TIME

The *user time* is the time of the process spent outside the kernel, i.e. when the program does something the user has programmed. It is part of the cpu time.

SYSTEM_TIME

In contrast to the user time, the *system time* is the part of the cpu time, the process has spent inside the kernel, e.g. doing a system call for reading/writing files.

The class for measure these types of time is `Timer`:

```
class Timer {
protected:
    double    _start, _sum;
    timetype_t _type;

public:
    Timer ( timetype_t type )

    Timer & start ();
```

```
    TTimer & pause ();
    TTimer & cont  ();

    float elapsed () const;

    TTimer & real ();
    TTimer & cpu  ();
    TTimer & user ();
    TTimer & sys  ();
}
```

The timer class can handle intervals of measurement by accumulating the time between specific points, e.g. pause and continue.

TTimer (timetype_t type)

Constructs a timer with a given time type. The default is `CPU_TIME`.

TTimer & start ();

Start a time measurement by storing the current time. Resets the internal accumulator to 0.

TTimer & pause ();

Pauses a time measurement. Adds the difference between the current and the stored start time into the accumulator.

TTimer & cont ();

Continues a time measurement. Stores the current time as a new start time.

float elapsed () const;

Returns the time since the last start time plus the accumulated time.

TTimer & real ();

TTimer & cpu ();

TTimer & user ();

TTimer & sys ();

Sets the corresponding type of time for the next measurement.

4.1.6 Miscellaneous

Some, rather unusual classes are covered in this section. This contains a random number generator and a class to count object creation and destruction.

4.1.6.1 Object Counter

One way to avoid memory leaks is a control of the number of objects which were created and destroyed afterwards. This is accomplished by the class `TObjCounter`.

```

class TObjCounter {
public:
    TObjCounter ();
    ~TObjCounter ();

    static void statistics ();
}

```

The number of allocated and deallocated objects is stored in local variables and, due to performance reasons, not protected from parallel access. This limits the usability to sequential environments. The counting is done by the constructor and destructor which are automatically called by deriving other classes from `TObjCounter`.

Since this is only an instrument to profile the implementation, the counting is only compiled in, if explicit debugging was chosen during the installation.

void statistics ();

Print a statistics about the number of created, destroyed and active objects.

4.1.6.2 Sorting Algorithms

The class `TSort` defines the basic interface for sorting algorithms in PHI but leaves out the actual algorithm part. This is accomplished in the class `TMergeSort` which implements the corresponding technique.

```

template <class T> class TSort {
    void sort ( TArray<T> & array, int lb, int ub ) const;

    void sort ( TArray<T> & array, TArray< int > & perm,
               int lb, int ub ) const;

    int compare ( const T & t1, const T & t2 ) const;
}

```

The class is defined as a template which corresponds to the type of objects to be sorted. The specific sorting algorithm is called by the method `sort`, whereas `compare` serves to compare to individual objects. Both methods are *pure virtual functions* in the baseclass. In `TMergeSort` the `sort` methods are implemented.

void sort (TArray<T> & array, int lb, int ub) const;

void sort (TArray<T> & array, TArray<int> & perm, int lb, int ub) const;

Sort given array. With `lb` and `ub` a part of the array can be specified to sort.

To sort the whole array, bot parameters must be `-1`.

int compare (const T & t1, const T & t2) const;

Compare `t1` and `t2` and return a value less than 0 iff $t1 < t2$, equal to 0 iff $t1 = t2$ and greater than 0 iff $t1 > t2$.

The following example implements a class for sorting integers by using the mergesort algorithm.

```
class TIntSort : public TMergeSort< int > {
    int compare ( const int & t1, const int & t2 ) const {
        if ( t1 < t2 ) return -1;
        if ( t1 > t2 ) return 1;
        return 0;
    }
}

int main (void) {
    TIntSort    sort;
    TArray< int > array( 4 );

    array[0] = 5; array[1] = 3; array[2] = 9; array[3] = 6;
    sort( array );
}
```

4.1.6.3 Show Progress of Computations

Especially for long computations the user usually needs some kind of feedback, how much was done, when the algorithm would finish or whether the program has actually stopped. This can be accomplished by `TProgressBar`, which implements a text-based status display.

```
class TProgressBar {
    TProgressBar ( real min, real max, real curr );

    void set_format ( const string & format );

    bool is_initialised () const;

    void init      ( real min, real max, real curr );
    void advance  ( real f );
    void finish   ();
}
```

The progress is defined by a position in an interval specified by `min` and `max`. The output is defined by a format string similar in style to the argument of the `printf`-function. It can be modified by `set_format`. The following table lists the corresponding arguments:

<code>%b</code>	Print progressmeter with optional length argument, e.g. <code>"%30b"</code> .
<code>%p</code>	Print current percentage.
<code>%e</code>	Print time to finish.
<code>%m</code>	Print current memory usage.
<code>%R</code>	Turn on red colour.
<code>%G</code>	Turn on green colour.
<code>%B</code>	Turn on blue colour.
<code>%F</code>	Turn on bold font.
<code>%N</code>	Reset to normal font and colour.

Colours are only supported on some output devices, e.g. X-terminals. The default format string is `"%b %p %e"`.

TProgressBar (real min, real max, real curr);

Construct a progressbar for the interval $[min, max]$ and the corresponding current position.

void set_format (const string & format);

Set the format of the progress meter.

bool is_initialised () const;

Return `true` if the progress meter was initialised.

void init (real min, real max, real curr);

Reset values of progressbar as in constructor.

void advance (real f);

Increase position in interval by f .

void finish ();

Erase progressbar from screen.

The output of a progressbar initialised by:

```
TProgressBar progress( 0, 100, 0 );
progress.set_format( "example %p %e (%m)" );
progress.advance( 40 );
```

would be

```
example 40% ETA 30s (104 MB)
```

4.1.6.4 Random Number Generator

Instead of the function `rand` or its variants on a typical UNIX system, PHI has its own random number generator. The main reason for this was the bad performance of the first ones in a parallel environment. They seem to be protected by a mutex and hence to parallel scaling was visible.

In most real-life situations this might not be a problem, but for algorithm tests it was convenient to fill matrices by random numbers and for this a good speedup was necessary.

```
class TRNG {
    TRNG ( ulong seed );

    double rand ( double max );

    void init ( ulong seed );
}
```

`TRNG` implements the [Mersenne Twister](#) (see [MN98]) random number generator which proved faster than `rand` even in sequential computations. The produced random numbers are also better.

```
TRNG ( ulong seed );
void init ( ulong seed );
    Construct or initialise the RNG by given seed.
double rand ( double max );
    Return next random number in sequence scaled by max.
```

4.2 Clusters

4.2.1 Clustertrees

A hierarchical decomposition of indexsets, usually called a *clustertree* is represented by an object of type `TCluster`. As the name indicates, this objects is actually not the whole tree itself, but a single node, namely the *root* of the tree.

```
class TCluster : public TObjCounter {
protected:
    uint          _first, _last;
    TArray< TCluster * > _sons;
    uint          _id;

public:
    TCluster ();
    TCluster ( uint first, uint last )
    ~TCluster ();
}
```

```

void set_is ( uint first, uint last );
uint first  () const;
uint last   () const;
uint offset () const;
uint size   () const;

uint n_sons      () const;
void set_n_sons ( uint n );

uint id          () const;
void set_id     ( uint n );

TCluster * son ( uint i );
void set_son   ( uint i, TCluster * son );
void add_son   ( TCluster * son );
void clear_sons ();

bool is_leaf   () const;

uint tree_size () const;
}

```

Beside the indexset, defined by the first and the last index, and the son nodes, a cluster also contains an *id*, which is a globally unique identifier, e.g. to directly address a specific cluster without pointers. The number of sons in a cluster is not fixed, which allows arbitrary clustertree constructions.

TCluster ();

TCluster ();

Construct and destroy a cluster. Upon deletion, all sons will be destroyed.

void set_is (uint first, uint last);

uint first () const;

uint last () const;

uint offset () const;

uint size () const;

Access the indexset of the cluster.

uint n_sons () const;

void set_n_sons (uint n);

Get or set the number of sons in the cluster. If the number of sons is changed, old links will be preserved if possible.

uint id () const;

void set_id (uint n);

Access the id of the cluster.

TCluster * son (uint i);

void set_son (uint i, TCluster * son);

Get or set a specific son of the cluster.

void add_son (TCluster * son);

Assign **son** to the first free slot in the local son-array. If no slot is free, an error occurs.

void clear_sons ();

Remove all references to son-clusters, e.g. to delete the cluster without deleting all son-clusters.

bool is_leaf () const;

Return **true**, if the cluster has no sons. Note, that this is not equal to **n_sons() == 0**, but it is also checked, whether there exists a non-NULL pointer in the son-array.

uint tree_size () const;

Return the number of nodes in the tree defined by the cluster as the root.

4.2.2 Construction of Cluster Trees

PHI provides two different methods to construct a cluster tree. The first uses *binary space partitioning* to divide the indexset into two (or more) subsets based on geometrical data provided for each index. The second method follows an algebraic approach and uses the matrix graph of a sparse matrix to bisect the graph and consequently partition the indexset.

4.2.2.1 Geometrical Cluster Tree Construction

As long as geometrical data for each index is available, this clustering technique is the preferred method for building a cluster tree. Two variants of geometrical clustering are available. The first balances the number of indices in each son, such that each subcluster has (almost) the same number of unknowns. In contrast to this, the second variant balances the geometry, e.g. each sub-cluster covers the same amount of space. The distinction between these two types is made by a parameter of type

```
typedef enum { BSP_CARDINALITY, BSP_GEOMETRY } bsp_type_t;
```

where the first value defines the cardinality driven clustering, whereas the second value defines the geometrically balanced method. Both algorithms can be combined with *nested dissection*, which separates both subclusters by a common interface, leading to 3 sons per node. Albeit, the latter approach only makes sense in a context with a local coupling of indices, e.g. finite elements or finite differences (see [GBK05]).

Implemented is the cluster tree construction in the class **TBSPCTBuilder**:

```
class TBSPCTBuilder {  
protected:  
    bsp_type_t                _bsp_type;
```

```

uint          _n_min;
uint          _min_leaf_lvl;
bool          _use_nd;
uint          _max_if_depth;
const TSparseMatrix::TCRS * _mat_S;

public:
  TBSPCTBuilder ( bsp_type_t bsp_type, uint n_min );

  TBSPCTBuilder ( bsp_type_t          bsp_type,
                  uint                n_min,
                  const TSparseMatrix * S,
                  uint                min_leaf_lvl,
                  uint                max_if_depth );

  void use_nd ( bool b );

  void set_bsp_type ( bsp_type_t t );

  TCluster * build ( const TArray< double * > & vertices,
                    uint                dim,
                    TArray< uint >        * perm,
                    uint                idx_ofs ) const;

  void join_if ( TCluster          * cluster,
                const TArray< double * > & vertices,
                uint                dim,
                TArray< uint >        * perm,
                uint                n ) const;
}

```

Beside the type of clustering stored in `_bsp_type`, each object also holds a lower bound for the size of each cluster in `n_min` and a minimal level for leaves in `_min_leaf_lvl`. Furthermore, parameters for nested dissection, which is activated by `_use_nd`, are part of the class, including the maximal depth of the interface part in `_max_if_depth` and the sparse matrix, which defines connectivity between clusters to determine the interface in `_mat_S`.

TBSPCTBuilder (bsp_type_t bsp_type, uint n_min);

Constructs a cluster tree construction object for standard bisection with the partitioning strategy `bsp_type` and the minimal leaf size `n_min`.

TBSPCTBuilder (bsp_type, n_min, S, min_leaf_lvl, max_if_depth);

Constructs a cluster tree construction object for nested dissection with the partitioning strategy `bsp_type`, the minimal leaf size `n_min`, sparse matrix `S`, the minimal leaf level `min_leaf_lvl` and the maximal interface depth `max_if_depth`.

void use_nd (bool b);

If `b` equals `true`, nested dissection is used for constructing the cluster tree. Otherwise, standard bisection is used.

void set_bsp_type (bsp_type_t t);

Depending on `t`, the partitioning of the clusters is done via cardinality or geometrically balanced clustering.

TCluster * build (vertices, dim, perm, idx_ofs) const;

Build a clustertree with the coordinates of each index stored in `vertices`. Afterwards, the computed permutation will be available in `perm`, while the numbering of the indices starts at `idx_ofs`.

void join_if (cluster, vertices, dim, perm, n) const;

In the case of nested dissection, e.g. each node has at least 3 sons, where the last son corresponds to the interface, this function reconstructs the given cluster tree `cluster` such that the first `n` levels are compressed to one level with at least $2^n + 1$ sons. The last son holds the newly created interface node between all of the other clusters. The resulting cluster tree is equal to a classical domain decomposition approach.

4.2.2.2 Algebraic Cluster Tree Construction

If no geometrical data for the indices is available but the connectivity between the unknowns can be described by a sparse matrix, one can choose the algebraic clustering algorithm (see [GBK05]). It is implemented in the class `TAlgCTBuilder`:

```
class TAlgCTBuilder {
protected:
    uint    _n_min;
    bool    _use_nd;
    uint    _min_if_depth;
    uint    _max_if_depth;

public:
    TAlgCTBuilder ( uint n_min,
                    bool use_ne_di,
                    uint min_if_depth,
                    uint max_if_depth );

    void use_nd ( bool b );

    TCluster * build ( const TSparseMatrix * S,
                      TArray< uint >      & perm,
                      uint                  idx_ofs ) const;
};
```

The local data of a `TAlgCTBuilder` object consists of the minimal size of a cluster in `_n_min` and data for nested dissection. Here, `_min_if_depth` describes the minimal level, on which interface leaves might occur. In the same way, `_max_if_depth` describes the maximal depth for the interface tree.

TAlgCTBuilder (n_min, min_if_depth, max_if_depth, use_ne_di);

Construct a cluster tree building object with a minimal leaf size of `n_min`, a minimal interface depth of `min_if_depth` (default = 0), a maximal interface depth of `max_if_depth` (default = ∞) and the flag `use_de_di` (default = `false`), signaling the usage of nested dissection.

void use_nd (bool b);

If `b` is `true`, the cluster tree will be built via nested dissection. Otherwise, bisection is used.

TCluster * build (S, perm, idx_ofs) const;

Build a cluster tree based on the connectivity described by the sparse matrix `S`. The resulting permutation of the indices is stored in `perm`. The numbering of the indices starts at `idx_ofs`.

4.2.3 Block Clustertrees

Putting together two clustertrees defines a *block clustertree*. In PHI this results again not in a tree as such, but in a single node, a *blockcluster*. The corresponding class `TBlockCluster` contains methods for handling the underlying tree.

```
class TBlockCluster : public TObjCounter {
protected:
    TBlockCluster          * _parent;
    TCluster               * _tau, * _sigma;
    TArray< TBlockCluster * > _sons;
    bool                   _adm;
    uint                   _proc;
    uint                   _id;

public:
    TBlockCluster ( TBlockCluster * parent );
    TBlockCluster ( TBlockCluster * parent, TCluster * tau, TCluster * sigma );
    ~TBlockCluster ();

    TBlockCluster * parent ();
    void set_parent ( TBlockCluster * parent );

    uint proc      () const;
    void set_proc ( uint p, bool recursive );

    TCluster * tau   ();
    TCluster * sigma ();

    void set_tau      ( TCluster * c );
    void set_sigma    ( TCluster * c );
    void set_clusters ( TCluster * tau, TCluster * sigma );

    bool is_adm      () const;
    void set_adm     ( bool b );

    uint id          () const;
    void set_id      ( uint i );
};
```

```
uint max_id () const;

uint n_sons      () const;
void set_n_sons ( uint n );

TBlockCluster * son ( uint i );
void set_son ( uint i, TBlockCluster * son );
void add_son ( TBlockCluster * son );

TBlockCluster * son ( uint i, uint j );
void set_son ( uint i, uint j, TBlockCluster * son );

TBlockCluster * son ( const TCluster * tau, const TCluster * sigma );

bool is_leaf () const;

uint tree_size () const;

uint compute_c_sp () const;
uint compute_c_sh ( uint nprocs ) const;

void collect_leaves ( TSSL< TBlockCluster * > & list, int depth ) const;
}
```

Each blockcluster is defined by a pair of clustertrees *tau* and *sigma*. Since it also represents a tree, the set of sons is stored in an array. The access to this set can be in a linear fashion by a single index or by two indices which represent the position defined by the product of *tau* and *sigma*.

Further data stored in a blockcluster is related to the admissibility of the block, a processor it was assigned to and as in the clustertree, an identifier.

The defined methods provide tree functionality, e.g. access of sons, or access to special properties like the sparsity and sharing constant.

```
TBlockCluster ( TBlockCluster * parent );
TBlockCluster ( TBlockCluster * parent, TCluster * tau, TCluster *
sigma );
```

Create an empty blockcluster or a blockcluster defined by the product of **tau** and **sigma**. No hierarchy is build.

```
TBlockCluster ();
```

Destruct blockcluster and all existing sons.

```
TBlockCluster * parent ();
void set_parent ( TBlockCluster * parent );
```

Get and set parent of blockclustertree.

```
uint proc () const;
```

```
void set_proc ( uint p, bool recursive );
```

Get and set processor of node. If recursive is **true**, the processor in all sons of the corrent node is set to the same value.

```

TCluster * tau ();
TCluster * sigma ();
void set_tau ( TCluster * c );
void set_sigma ( TCluster * c );
void set_clusters ( TCluster * tau, TCluster * sigma );
    Return or set clustertrees this block is defined over.

bool is_adm () const;
void set_adm ( bool b );
    Access admissibility flag in blockcluster.

uint id () const;
void set_id ( uint i );
uint max_id () const;
    Get or set information about identifier. The function max_id returns the maximal id in all nodes of the current tree.

uint n_sons () const;
void set_n_sons ( uint n );
    Return or define the number of sons of this node.

TBlockCluster * son ( uint i );
void set_son ( uint i, TBlockCluster * son );
void add_son ( TBlockCluster * son );
    Access sons numbered in a linear fashion from 0...no_of_sons() - 1.

TBlockCluster * son ( uint i, uint j );
void set_son ( uint i, uint j, TBlockCluster * son );
    Access sons as defined by tensor product construction.

TBlockCluster * son ( const TCluster * tau, const TCluster * sigma );
    Return son corresponding to given clustertrees,

bool is_leaf () const;
    Return true, if the block cluster has no sons. Note, that this is not equal to n_sons() == 0, but it is also checked, whether there exists a non-NULL pointer in the son-array.

uint tree_size () const;
    Return the number of nodes in the tree defined by the block cluster as the root.

uint compute_c_sp () const;
uint compute_c_sh ( uint nprocs ) const;
    Compute the sparsity and sharing constant of the block clustertree and return result.

void collect_leaves ( TSSL<TBlockCluster * > & list, int depth ) const;

```

Collect all leaves reachable from current node into given list. If depth does not equal -1, the tree is only traversed up to the corresponding depth and the reached nodes are collected.

4.2.4 Construction of Block Clustertrees

The construction of blockclustertrees is supported by several classes. The main algorithm is implemented in `TBCBuilder` which expects two clustertrees and two criteria. The first criterion is of type `TAdmCondition` defines the *admissibility*. The standard algorithm is terminated if a leaf in the clustertree is reached, or if the newly created blockcluster is smaller than some predefined size. The latter is checked by using objects of type `TSizeCriterion`, which is the second criterion supplied to `TBCBuilder` objects.

4.2.4.1 TBCBuilder

The class `TBCBuilder` mainly consists of a single method implementing the actual algorithm for blockcluster tree construction.

```
class TBCBuilder {
protected:
    uint _min_lvl;

public:
    TBCBuilder ();
    ~TBCBuilder ();

    void set_min_level ( uint l );

    TBlockCluster * build ( TCluster * tau, TCluster * sigma,
                           const TSizeCriterion * sc,
                           const TAdmCondition * ac ) const;

protected:
    TBlockCluster * rec_build ( TBlockCluster * parent,
                               TCluster * tau, TCluster * sigma,
                               const TSizeCriterion * sc,
                               const TAdmCondition * ac,
                               uint level, uint & id ) const;
}
```

The method `build` implements the standard construction of a block clustertree by multiplying two clustertrees and stopping the procedure, if an admissible node is detected. With `_min_lvl` one can control the first level, on which leaves are allowed (see HDD).

```
void set_min_level ( uint l );
```

Set the first level of the block cluster tree on which leaves are allowed.

TBlockCluster * build (...) const;

Build a block cluster tree over **tau** and **sigma** with the size criterion **sc** and the admissibility condition **ac**.

TBlockCluster * rec_build (...) const;

Actual implementation of the recursive tree generation algorithm.

A variant of this scheme is implemented in **TDDBCBuilder** which creates block clustertrees in the context of *domain-decomposition* methods. Here, off-diagonal blocks, except the last row/column, are not refined. Furthermore, the minimal leaves level is necessary to ensure, that block clusters are not shared by different processors.

```
class TDDBCBuilder : public TBCBuilder {
public:
    TDDBCBuilder ( uint min_lvl );
    ~TDDBCBuilder ();

    TBlockCluster * build ( const TProcSet      & procs,
                           uint                pid,
                           TCluster           * tau,
                           TCluster           * sigma,
                           const TSizeCriterion * sc,
                           const TAdmCondition * ac ) const;
}

```

To take into account the distribution of data in case of parallel computations, a processor set has to be supplied to the construction of the block cluster tree.

TBlockCluster * build (const TProcSet & procs, ...) const;

Parallel construction of the block cluster tree on the processor set **procs**. Only local nodes of the tree are built.

4.2.4.2 TAdmCondition

The class **TAdmCondition** as it is implemented in PHI is an abstract class, only defining the interface.

```
class TAdmCondition {
public:
    TAdmCondition ();
    ~TAdmCondition ();

    bool is_adm ( const TBlockCluster * c ) const = 0;
}

```

Available in PHI is a implementation for the standard admissibility in FEM/BEM applications, e.g.

$$\min(\text{diam}(t_1), \text{diam}(t_2)) \leq \eta \text{dist}(t_1, t_2),$$

where $\text{diam}(t)$ describes the diameter of the cluster t and $\text{dist}(t_1, t_2)$ stands for the distance between the clusters t_1 and t_2 . The corresponding class for this condition is [TStdAdmCond](#):

```
class TStdAdmCond : public TAdmCondition {
protected:
    real _eta;
    bool _use_max;

public:
    TStdAdmCond ( real eta, bool use_max );

    real eta      () const;
    void set_eta ( real eta );
}
```

Beside the parameter `eta` from the admissibility condition, a flag is stored, which enables the test of the maximum of the diameters instead of the minimum.

A variant if this condition, based on a combination of standard and weak admissibility (see [[HKK04](#)]) is implemented in [TWeakStdAdmCond](#):

```
class TWeakStdAdmCond : public TStdAdmCond {
public:
    TWeakStdAdmCond ( real eta );
}
```

Furthermore, the weak admissibility for the algebraic clustering method (see [[GBK05](#)]) is available in the form of [TAlgAdmCond](#):

```
class TAlgAdmCond : public TAdmCondition {
protected:
    TSparseMatrix * _mat;
    TPermutation  * _old_to_new, * _new_to_old;
    bool          _del_new_to_old;

public:
    TAlgAdmCond ( TSparseMatrix * S,
                  TPermutation  * old_to_new,
                  TPermutation  * new_to_old );
}
```

Here, the connectivity between clusters, and hence their admissibility is tested by looking at the matrix graph described by the sparse matrix `mat`. Depending on the ordering of this matrix, permutations from the ordering of the indices in the matrix to the ordering of the indices in the cluster tree and the corresponding inverse permutation have to be supplied. The latter might be `NULL`, in which case it is constructed inside the object.

4.2.4.3 TSizeCriterion

In contrast to `TAdmCondition` the class `TSizeCriterion` comes with a standard implementation. The criterion is based upon two variables: `n_min` and `rank`. If `rank` is 0, a blockcluster is considered *small*, if its size is less than n_{min}^2 . If `rank` is greater than zero, the storage size of a low-rank matrix is additionally compared with the storage size of a dense matrix. If the latter is smaller, it is also considered as a *small* blockcluster.

```
class TSizeCriterion {
protected:
    uint _n_min;
    uint _rank;

public:
    TSizeCriterion ( uint n_min, uint rank );

    void set_n_min ( uint n_min );
    void set_rank ( uint rank );

    bool too_small ( const TBlockCluster * cluster ) const;
}
```

The full definition of the methods is above is as follows:

```
TSizeCriterion ( uint n_min, uint rank );
    Construct object with given parameters.

void set_n_min ( uint n_min );
void set_rank ( uint rank );
    Set n_min and rank.

bool too_small ( const TBlockCluster * cluster ) const;
    Return true if cluster is small and false otherwise.
```

4.3 Matrices

Matrices in PHI are defined over a blockcluster, which itself holds the information about the size or the relative position of the matrix. Only under special circumstances the implementation differs from this paradigm, i.e. if no blockcluster in a blockcluster tree exists which corresponds to a matrix.

4.3.1 TMatrix

`TMatrix` is the baseclass for all matrix classes in PHI. It defines the basic interfaces and implements some of the functionality connected to it.

```
class TMatrix : public TObjCounter, public TStreamable {
protected:
    TBlockCluster * _cluster;
    uint           _row_ofs, _col_ofs;
    bool           _symmetric;
    uint           _proc;
    TMutex        _mutex;

public:
    TMatrix ();
    TMatrix ( TBlockCluster * c );

    uint rows () const;
    uint cols () const;

    uint row_ofs () const;
    uint col_ofs () const;
    void set_ofs ( uint r, uint c );

    bool is_symmetric () const;
    void set_symmetric ( bool b );

    uint proc () const;
    void set_proc ( uint p );

    void lock ();
    void unlock ();

    TBlockCluster * cluster ();
    void set_cluster ( TBlockCluster * c );

    error_t scale ( real f );
    error_t add_mat ( real a, const TMatrix * matrix );
    error_t mul_vec ( real alpha, const TVector * x,
                    real beta, TVector * y, bool trans ) const;
    TMatrix * mul_mat_r ( real alpha, const TMatrix * B,
                        bool trans_A, bool trans_B ) const;
    TMatrix * mul_mat_l ( real alpha, const TMatrix * A,
                        bool trans_A, bool trans_B ) const;

    TMatrix * clone () const;
    TMatrix * copy () const;
    error_t copy_to ( TMatrix * A ) const;

    TVector * row_vector () const;
    TVector * col_vector () const;

    ulong byte_size () const;

    uint type () const;
    bool is_type ( uint t ) const;

    error_t read ( TByteStream & s );
    error_t build ( TByteStream & s );
    error_t write ( TByteStream & s ) const;
    ulong bs_size () const;
}

```

Beside the methods for accessing informations about the matrix, the interface also contains functions for matrix operations and for serialisation.

The local data in an `TMatrix` object stores information about the cluster and the relative position. Allthoug this could also be retrieved from the blockcluster, it is copied for performance reasons. Also contained in a matrix is the information about symmetry, which defaults to `false` and has to be set manually. For parallel computations the corresponding processor associated to the matrix is also stored. The local mutex can be used in the `PThread mode` to prevent simultaneous access to the matrix in concurrent threads.

```
TMatrix ();
```

```
TMatrix ( TBlockCluster * c )
```

Constructs an empty matrix or a matrix defined by given blockcluster `c`. In the latter case, the

```
uint rows () const;
```

```
uint cols () const;
```

Return the number of rows and columns in the matrix. Both methods are pure virtual and have to be overwritten in derived classes.

```
uint row_ofs () const;
```

```
uint col_ofs () const;
```

```
void set_ofs ( uint r, uint c );
```

Get or set the offset (relative position) of the matrix.

```
bool is_symmetric () const;
```

```
void set_symmetric ( bool b );
```

Access the symmetry information in the matrix.

```
uint proc () const;
```

```
void set_proc ( uint p );
```

Access processor field of the matrix.

```
void lock ();
```

```
void unlock ();
```

Lock or unlock private mutex of the matrix.

```
TBlockCluster * cluster ();
```

```
void set_cluster ( TBlockCluster * c );
```

Access local blockcluster of matrix.

```
error_t scale ( real f );
```

Scale matrix by the factor `f`.

```
error_t add_mat ( real a, const TMatrix * M );
```

Add matrix `M` scaled by `a` to local matrix

error_t mul_vec (alpha, x, beta, y, trans) const;

Perform matrix-vector product $y := \beta y + \alpha Ax$ where A is the local matrix.

TMatrix * mul_mat_r (alpha, B, trans_A, trans_B) const;

Do matrix multiplication αAB where A is the local matrix and return result. If **trans_A** (**trans_B**) is **true**, matrix A (B) is transposed.

TMatrix * mul_mat_l (alpha, A, trans_A, trans_B) const;

Do matrix multiplication αAB where B is the local matrix and return result. If **trans_A** (**trans_B**) is **true**, matrix A (B) is transposed.

TMatrix * clone () const;

Return an object of the same type, e.g. **TMatrix**. (*pure virtual*)

TMatrix * copy () const;

Return a copy of the local object. (*pure virtual*)

error_t copy_to (TMatrix * A) const;

Copy content of local matrix to matrix **A**. **A** is assumed to be of a compatible (or same) type as the local matrix, otherwise an error is returned. (*pure virtual*)

TVector * row_vector () const;

TVector * col_vector () const;

Return a compatible vector for left or right multiplication. By default, this is a scalar vector of type **TScalarVector** with the correct size and offset.

ulong byte_size () const;

Return memory size of object with all associated subobjects, e.g. submatrices.

type () const;

bool is_type (uint t) const;

string typestr () const;

Implements runtime type information aside from C++ for performance reasons (is called too often). Each new matrix class has to define a new type and set up the hierarchy of types in the corresponding methods (see **RTTI**). By calling **typestr**, the registered name of the type is returned.

error_t read (TByteStream & s);

error_t build (TByteStream & s);

error_t write (TByteStream & s) const;

Read or write the local matrix from or to the bytestream **s** and return **true** if the operation was successful and **false** otherwise. If **build** is called, the matrix with all submatrices is constructed, whereas **read** just fills an existing matrix.

ulong bs_size () const;

Returns the size needed to store the object in a bytestream.

Most methods in `TMatrix` are not pure virtual although no real implementation can be done. Instead, a warning will be printed that the corresponding method has to be overwritten.

In the following sections only those methods are presented which are not already in the definition of the `TMatrix` class and which have to be overwritten in any case. Only of the implementation contains specific details important to that class it is listed and described.

4.3.2 Dense Matrices

As the name indicates, objects of type `TDenseMatrix` represent dense matrices, i.e. matrices storing a coefficient for each index in the product indexset.

The following de

```

class TDenseMatrix : public TMatrix {
protected:
    uint    _n, _m;
    real    * _data;

public:
    TDenseMatrix ( uint n, uint m );
    TDenseMatrix ( const TDenseMatrix & mat );

    ~TDenseMatrix ();

    error_t set_size ( uint n, uint m, bool zero_fill );

    real * f_matrix ();

    real & operator () ( uint i, uint j );

    error_t add_block ( real alpha, real beta, const TDenseMatrix * M );

    error_t add_rank ( const TScalarVector & a, const TScalarVector & b );
    error_t add_rank ( const TRkMatrix * A );
    error_t copy_rank ( const TRkMatrix * A );
}

```

The member variables of `TDenseMatrix` hold information about the size of the matrix and an array with the coefficients. The matrix in this array is stored column-wise which is different than the usual C-style arrays and follows the Fortran-style. The reason for this lies in the usage of *LAPACK* which expects the latter format.

The main extension to the basic matrix interface is the access to a specific coefficient and methods for handling low rank matrices.

TDenseMatrix (uint n, uint m);

Construct a dense matrix with a size definedr by **n** and **m**.

TDenseMatrix (const TDenseMatrix & mat);

Copy constructor for dense matrices.

TDenseMatrix ();

Destructs a **TDenseMatrix** object and deletes the corresponding data.

error_t set_size (uint n, uint m, bool zero_fill);

Set the size of the matrix to $n \times m$. If **zero_fill** is **true**, the matrix is initialised with 0 coefficients.

real * f_matrix ();

Returns the internal pointer to the real matrix. Do not delete this !

real & operator () (uint i, uint j);

Returns the coefficient with index (i, j) .

error_t add_block (real alpha, real beta, const TDenseMatrix * M);

Adds the intersection of the local matrix and **M** to the local matrix. While **M** is scaled by **beta**, the local part is scaled by **alpha**.

error_t add_rank (const TScalarVector & a, const TScalarVector & b);

void add_rank (const TRkMatrix * A);

Add low rank matrices to the local dense matrix.

error_t copy_rank (const TRkMatrix * A);

Copies the given low rank matrix **A** to the local dense matrix.

4.3.3 Low Rank Matrices

Low rank matrices are represented by the product of two dense matrices A and B : $A \cdot B^T$. For a $n \times m$ matrix R , the matrices A corresponds to a $n \times k$ and B to a $m \times k$ matrix, while k defines the rank of R . This structure can also be found in the class for these matrices in PHI: **TRkMatrix**.

```
class TRkMatrix : public TMatrix {
protected:
    real * _mat_A, * _mat_B;
    uint   _n, _m;
    uint   _mem_rank, _cur_rank, _wanted_rank;
    real   _trunc_eps;

public:
    TRkMatrix ( uint n, uint m );
    ~TRkMatrix ();

    real * matrix_A ();
    real * matrix_B ();

    uint rank          () const;
    uint mem_rank      () const;
    uint current_rank  () const;
    uint wanted_rank   () const;

    error_t set_rank      ( uint k, bool zero_fill );
```

```

void    set_current_rank ( uint k );
void    set_wanted_rank  ( uint k );

error_t set_size ( uint n, uint m, uint k, bool zero_fill );

real epsilon      () const;
void set_epsilon  ( real f );

void truncate      ();
void truncate_rank ( uint k );
void truncate_eps  ( real eps );
void truncate_to_wanted ();

error_t add_rank   ( uint k, real * A, real * B, real eps );
error_t add_dense  ( real * D );

error_t copy_dense ( const TDenseMatrix * A, uint k );
error_t copy_dense ( const TDenseMatrix * A, real eps );
error_t copy_dense ( const TDenseMatrix * A )
}

```

The arrays `_mat_A` and `_mat_B` contain the two matrices, while the size of the low-rank matrix is stored in `_n` and `_m`. Several data fields are needed to hold the information about the rank of the matrix. The current rank of the storage used by `_mat_A` and `_mat_B` can be found in `_mem_rank`. But usually one is more interested in the mathematical rank of the matrix which equals the value of `_cur_rank`. Since many algorithms in the context of \mathcal{H} -matrices involve truncating matrices, some form of representation is necessary to define the *wished* rank of low rank blocks. This is accomplished by `_wanted_rank` and `_trunc_eps`. While the former directly sets the rank to truncate to, the latter specifies a relative border during the *singular value decomposition* (or *SVD*).

TRkMatrix (uint n, uint m);

Construct a low rank matrix of size `n` by `m` and rank 0.

TRkMatrix ();

Destruct low rank matrix and delete all data.

real * matrix_A ();

real * matrix_B ();

Return pointers to the internal arrays holding matrix coefficients for *A* and *B*.

uint rank () const;

uint mem_rank () const;

uint current_rank () const;

uint wanted_rank () const;

Return corresponding rank information. The value returned by the method `rank` equals the current rank of the matrix.

error_t set_rank (uint k, bool zero_fill);

void set_current_rank (uint k);

void set_wanted_rank (uint k);

Set corresponding rank of the matrix. In this case, `set_rank` sets the memory rank of the matrix (mathematical rank is unknown till next truncation). If `zero_fill` equals `true`, the newly created arrays are initialised with 0.

error_t set_size (uint n, uint m, uint k, bool zero_fill);

Set size and memory rank of matrix. Again, if `zero_fill` is `true`, everything is initialised by 0.

real epsilon () const;

void set_epsilon (real f);

Get or set truncation epsilon for adaptively choosing the rank.

void truncate ();

void truncate_rank (uint k);

Truncate matrix to given rank `k`. After the truncation, the mathematical rank is at most `k`.

void truncate_eps (real eps);

Truncate matrix up to precision `eps`. The mathematical rank is set accordingly.

void truncate_to_wanted ();

Truncate the matrix according to wished truncation mode. If `_wanted_rank` is greater than 0, this fixed rank is chosen. Otherwise the value in `_trunc_eps` is used during truncation.

error_t add_rank (uint k, real * A, real * B, real eps);

void add_dense (real * D);

Add a given low rank matrix or dense matrix, represented in direct form, to the local matrix.

error_t copy_dense (const TDenseMatrix * A, uint k);

error_t copy_dense (const TDenseMatrix * A, real eps);

error_t copy_dense (const TDenseMatrix * A)

Copy given dense matrix with defined rank, precision or depending on local setting to matrix.

4.3.4 Sparse Matrices

A sparse matrix, e.g. found in the context of finite element computations, is represented in two different types of data structure. The first one is a set of indices associated with a coefficient. The row index is defined as an array position, whereas the array contains lists to objects of type `entry_t`:

```
typedef struct {
    uint col;
    real coeff;
} entry_t;
```

This structure holds the corresponding column position and the coefficient of the entry. This allows a very flexible way of changing the coefficient pattern, e.g. add new entries.

In contrast to this, the second type of data structure is meant for a fixed sparsity pattern and is based on the *compressed row storage* format. That is, three arrays are stored in a **TCRS** object:

```
class TCRS {
public:
    TArray< real > coeff;
    TArray< uint > col_ind;
    TArray< uint > row_ind;
}
```

The arrays `coeff` and `col_ind` hold all coefficient and the corresponding column indices, ordered row-wise, e.g. beginning with the first row. The third array `row_ind` stores to start positions to the first two arrays for each row, e.g. `row_ind[i]` points to the position in `coeff` and `col_ind` where the entries of the *i*'th row are stored.

The advantage of the *CRS* data structure is its compactness, which needs much less storage space than the list-based format. Furthermore, matrix-vector multiplications can be performed much faster.

All of this put together into a matrix class results in **TSparseMatrix**:

```
class TSparseMatrix : public TMatrix {
protected:
    TArray< TSSL< entry_t > > _data;
    uint _cols;
    TCRS * _crs;

public:
    real entry ( uint i, uint j ) const;
    error_t set_entry ( uint i, uint j, real c );
    error_t add_entry ( uint i, uint j, real c );

    TArray< TSSL< entry_t > > & entries ();

    error_t init_entry_data ();
    void sort_entries ();

    error_t init_crs ( uint n );
    error_t build_crs ();
    bool has_crs_format () const;
    TCRS * crs_data ();

    error_t permute ( const TPermutation & perm );

    bool test_symmetric () const;
}
```

The interface of `TSparseMatrix` is mainly restricted to the access of the coefficients of the matrix and performing conversions between both storage formats. It should also be noted that no matrix multiplication methods have been implemented so far.

real entry (uint i, uint j) const;

Returns the coefficient associated with index (i, j) or 0 if no such index can be found in the matrix.

error_t set_entry (uint i, uint j, real c);

error_t add_entry (uint i, uint j, real c);

Set or add coefficient at index (i, j) to the matrix. If `add_entry` is used and no such index can be found, it is inserted to the matrix

TArray<TSSL< entry_t > > & entries ();

Gives access to the complete set of entries.

error_t init_entry_data ();

Initialise the list-based data structure, e.g. resize the corresponding array to the number of rows.

void sort_entries ();

Sorts all lists or CRS arrays of entries according to their column number. This seems to be important for parallel computations to guarantee equal results on all processors. If not sorted, the results might differ due to rounding errors. The sorting is done by *bubble sort* ($O(n^2)$ complexity) since the number of elements per row is usually very small.

error_t init_crs (uint n);

Initialise the CRS-based data structure for `n` entries.

error_t build_crs ();

Convert the data represented by lists into a CRS-based structure. The former will be freed afterwards.

bool has_crs_format () const;

Return `true`, if the CRS-based format is used.

TCRS * crs_data ();

Give direct access to the CRS format.

error_t permute (const TPermutation & perm);

Permute the stored entries w.r.t. the given permutation `perm`.

bool test_symmetric () const;

Return `true`, if the stored matrix is symmetric. The method also changes the internal symmetry status of the matrix.

4.3.5 Blockmatrices

Blockmatrices build a basic part of representing the hierarchy of \mathcal{H} -matrices. In PHI blockmatrices are the counterpart to inner nodes of the blockcluster tree, e.g. no leaves, and have therefore the same number of subblocks or submatrices. These submatrices have to be defined by the same product indexset as the sons of the corresponding node in the blockcluster tree.

The datatype in PHI used to represent blockmatrices is `TBlockMatrix`:

```
class TBlockMatrix : public TMatrix {
protected:
    uint      _rows, _cols;
    uint      _n, _m;
    TMatrix  ** _blocks;

public:
    ~TBlockMatrix ();

    uint block_rows () const;
    uint block_cols () const;
    uint no_of_blocks () const;

    void set_block_struct ( uint n, uint m );

    TMatrix * block ( uint i, uint j );
    void set_block ( uint i, uint j, TMatrix * A );

    void clear_blocks ();

    TMatrix * bc_block ( const TBlockCluster * t ) const;
    TMatrix * bc_block ( const TCluster * tau, const TCluster * sigma ) const;

    void collect_leaves ( TSSL< TMatrix * > & list ) const;

    void truncate_rank ( uint k );
    void truncate_eps ( real eps );
    void truncate_to_wanted ();
}

```

As can be seen, most methods are related to the management of the block-structure and submatrices. Furthermore, truncation methods regarding only low rank matrices are included in the interface.

TBlockMatrix ();

Destroy the blockmatrix and all subblocks.

uint block_rows () const;

uint block_cols () const;

uint no_of_blocks () const;

Return the number of blockrows, blockcolumns and the total number of subblocks in the matrix.

void set_block_struct (uint n, uint m);

Set the structure of the matrix to **n** blockrows and **m** blockcolumns.

TMatrix * block (uint i, uint j);

void set_block (uint i, uint j, TMatrix * A);

Get or set the block (i, j) .

void clear_blocks ();

Remove all references to subblocks, e.g. to delete the blockmatrix without freeing submatrices.

TMatrix * bc_block (const TBlockCluster * t) const;

TMatrix * bc_block (const TCluster * tau, const TCluster * sigma) const;

Return the subblocks corresponding to the given block cluster **t** or $(\mathbf{tau}, \mathbf{sigma})$.

void collect_leaves (TSSL<TMatrix * > & list) const;

Put all leaves of the blockmatrix into the given list.

void truncate_rank (uint k);

void truncate_eps (real eps);

void truncate_to_wanted ();

Truncate all reachable low rank matrices to the given rank **k**, the accuracy *eps* or to the wanted rank of each matrix.

4.3.6 \mathcal{H} -Matrices

An H -matrix is a special kind of blockmatrix but with further capabilities, e.g. for parallel computations.

```
class THMatrix : public TBlockMatrix {
protected:
    TSSL< TMatrix * >          _leaves;
#ifdef PHI_THR
    TArray< TSSL< TMatrix * > > _proc_leaves;
#endif
    TArray< uint >            _tau_idx, _sig_idx;
    uint                      _nprocs;

public:
    uint nprocs      () const;
    void set_nprocs ( uint nprocs );

#ifdef PHI_BSP
    uint min_tau_idx () const;
    uint max_tau_idx () const;
    uint min_sig_idx () const;
    uint max_sig_idx () const;
#endif

    uint min_tau_idx ( uint p ) const;
```

```

uint max_tau_idx ( uint p ) const;
uint min_sig_idx ( uint p ) const;
uint max_sig_idx ( uint p ) const;

void comp_min_max_idx ();

const TSSL< TMatrix * > & leaves () const;

#ifdef PHI_THR
    const TSSL< TMatrix * > & leaves ( uint i ) const;
#endif

void build_leaf_list ();

#ifdef PHI_THR
    void truncate_rank      ( uint k );
    void truncate_eps      ( real eps );
    void truncate_to_wanted ();
#endif
}

```

In addition to the set of direct submatrices as it is stored in a blockmatrix, objects of type `THMatrix` also hold lists for the set of all leaves. The reason for this is an optimised matrix-vector multiplication for H -matrices with no recursion. Furthermore, information about the distribution of the matrix is stored. In particular, the minimal and maximal indices of locally stored blocks is hold in the arrays `_tau_idx` and `_sig_idx`.

Due to differences between `PThread mode` and `BSP mode`, the access to these data is dependent on the specific mode in use, e.g. since in `PThread mode`, the complete address space is shared, all leaf-lists are stored in the same object, whereas in `BSP mode`, each processor has it's own address space and hence, it's own version of the H -matrix object.

Also, in `PThread mode`, special versions of the truncation functions are implemented to make use of the parallel environment. In `BSP mode`, this is exploited already in the routines in `TBlockMatrix` due to the distributed storage.

```
uint nprocs () const;
```

```
void set_nprocs ( uint nprocs );
```

Get or set the number of processors, the H -matrix is defined over.

```
uint min_tau_idx () const;
```

```
uint max_tau_idx () const;
```

```
uint min_sig_idx () const;
```

```
uint max_sig_idx () const;
```

Return the minimal/maximal row-/column-indices of the locally stored matrix blocks.

```
uint min_tau_idx ( uint p ) const;
```

```
uint max_tau_idx ( uint p ) const;
```

```
uint min_sig_idx ( uint p ) const;
```

```
uint max_sig_idx ( uint p ) const;
```

Return the minimal/maximal row-/column-indices of the matrix-blocks stored on the `p`'th processor.

```
void comp_min_max_idx ();
```

Compute the minimal/maximale row-/column-indices for each processor.

```
const TSSL<TMatrix *> & leaves () const;
```

```
const TSSL<TMatrix *> & leaves ( uint i ) const;
```

Give access to the set of leaves (of the `i`'th processor).

```
void build_leaf_list ();
```

Build the list of leaves per processor.

```
void truncate_rank ( uint k );
```

```
void truncate_eps ( real eps );
```

```
void truncate_to_wanted ();
```

Truncate in parallel all reachable low rank matrices to the given rank `k`, the accuracy `eps` or to the wanted rank of each matrix.

4.3.7 Domain Decomposition Matrices

This matrix class was implemented to handle the case of domain decomposition with non-overlapping domains and an interface. Beside the classical case, e.g. `p` sub-domains and one interface, it is also capable of handling nested dissection. For `p` subdomains the matrix has the form

$$\begin{pmatrix} A_{11} & & & & A_{1,p+1} \\ & A_{22} & & & A_{2,p+1} \\ & & \ddots & & \vdots \\ & & & A_{pp} & A_{p,p+1} \\ A_{p+1,1} & A_{p+1,2} & \cdots & A_{p+1,p} & A_{p+1,p+1} \end{pmatrix},$$

e.g. all off-diagonal blocks are zero, except the last row and column. Matrices with an index containing `p + 1` correspond to the interface-domain-coupling or the interface part of the problem.

```
class TDDMatrix : public TBlockMatrix {
protected:
    TProcSet    _procs;

public:
    void set_pid ( uint i );
    uint pid    () const;

    const TProcSet & procs    () const;
    uint          nprocs    () const;
    void          set_procs ( const TProcSet & ps );
}
```

Beside the id of the local processor, which is already defined in `TMatrix`, objects of type `TDDMatrix` also hold information about the complete processor set, which belongs to this matrix.

```
void set_pid ( uint i );
uint pid () const;
```

Wrapper for accessing the processor id, here called *pid*.

```
const TProcSet & procs () const;
uint nprocs () const;
void set_procs ( const TProcSet & ps );
```

Access the processor set of the matrix.

The coupling between the interface and the domain, e.g. matrices in the last column and row of a domain decomposition matrix, is put into a special matrix type, where the type of coupling is determined by a local flag:

```
typedef enum { IF_TO_INNER, INNER_TO_IF } coupling_t;
```

The first value defines the coupling from interface indices to the inner or domain indices and results in the following structure for p subdomains:

$$(A_{11} \ A_{12} \ \cdots \ A_{1,p} \ A_{1,p+1})$$

whereas the second value defines the opposite case with a transposed matrix structure.

```
class TDDCouplingMatrix : public TBlockMatrix {
protected:
    TProcSet    _procs;
    coupling_t  _coupling;

public:
    void set_pid ( uint i );
    uint pid     () const;

    const TProcSet & procs     () const;
    uint           nprocs     () const;
    void           set_procs   ( const TProcSet & ps );

    coupling_t     coupling_type () const;
    void           set_coupling_type ( coupling_t c );
}
```

Beside the handling of the coupling type, `TDDCouplingMatrix` is identical to `TDDMatrix`.

4.3.8 Representing Inverse LU Decompositions

Since the result of a LU factorisation is stored in the matrix itself, it needs some special methods to evaluate the inverse operator. This is accomplished by objects of type `TLUInvMatrix`. Since only matrix-vector multiplications are possible with the factorised matrices, the set of overloaded methods is restricted to these functions.

```
class TLUInvMatrix : public TMatrix {
protected:
    TMatrix * _lu_factor;
    bool      _use_block;

public:
    TLUInvMatrix ( TMatrix * lu_factor, bool use_block );
}
```

Apart from the actual matrix, a flag is stored indicating whether the block version should be used (see `TLU`).

4.3.9 Permutation Matrices

Permutation matrices are used to store the reordering of indices during cluster tree construction. But they can hold any kind of permutation, e.g. for reordering sparse matrices.

```
class TPermutation : public TMatrix {
protected:
    TArray< uint > _perm;

public:
    TPermutation ( uint r, uint c );
    TPermutation ( TBlockCluster * cluster );
    TPermutation ( const TArray< uint > & perm );
    TPermutation ( const TPermutation & perm );

    void set_size ( uint r, uint c );

    TArray< uint > & permutation ();

    void invert ();

    error_t mul_vec ( real alpha,
                    const TVector * x,
                    real beta,
                    TVector * y,
                    bool trans = false ) const;
}
```

The actual permutation is stored in the array `_perm`. The main difference between a permutation and other matrix types is the set of algebraic operations. An object of type `TPermutation` can not be part of other matrix operations, e.g. addition, multiplication. Only matrix-vector multiplication and inversion is possible.

TPermutation (uint r, uint c);

TPermutation (TBlockCluster * cluster);

Construct an **TPermutation** object of size **r** x **c** or defined by **cluster**.

TPermutation (const TArray<uint> & perm);

TPermutation (const TPermutation & perm);

Construct an **TPermutation** object with the permutation given either by an array or another permutation.

void set_size (uint r, uint c);

Set dimension of the permutation matrix.

TArray<uint> & permutation ();

Give access to the local array holding the permutation.

void invert ();

Invert the permutation.

error_t mul_vec (alpha, x, beta, y, trans) const;

Do a matrix-vector multiplication, e.g. permute the coefficients of the vector according to the locally stored permutation.

4.4 Vectors

In the same way as different types of matrices are represented by different classes, vectors are encapsulated into types to handle varying requirements. In particular, special matrix classes, e.g. for domain decomposition, are based on special vector types for the usual arithmetics.

4.4.1 TVector

The base class for all vector types is defined by **TVector**, which includes the basic interface for handling vectors. Furthermore, due to the inheritance from **TStreamable**, vectors are also capable of putting themselves into bytestreams.

The only data field, defined in **TVector** stores the offset of the indexset, e.g. the number of the first index.

```
class TVector : public TMultiRef, public TObjCounter, public TStreamable {
protected:
    uint _ofs;

public:
    TVector ( uint ofs );
    TVector ( const TVector & v );
    ~TVector ();

    uint ofs      () const;
```

```
void set_ofs ( uint n );
uint size    () const = 0;

TVector & set      ( real f );
TVector & set_rand ( uint seed );

TVector & scale ( real f );
TVector & assign ( real f, const TVector * x );
real      dot   ( const TVector * x ) const;
real      norm2 () const;
TVector & axpy  ( real alpha, const TVector * x );

uint type      () const;
bool is_type   ( uint t ) const;
string typestr () const;

ulong byte_size () const;

TVector & operator = ( const TVector & v );

TVector * copy  () const = 0;
TVector * clone () const = 0;

void print ( ostream & os, uint ofs ) const;
}
```

Beside the usual linear algebra methods, e.g. scaling, dot product and linear combination, `TVector` also defines methods for runtime type information and virtual construction.

TVector (uint ofs);

TVector (const TVector & v);

TVector ();

Create or destroy a `TVector` object with a given offset `ofs` or by copying the data from `v`.

uint ofs () const;

void set_ofs (uint n);

uint size () const = 0;

Access the offset and size information of the vector object.

TVector & set (real f);

TVector & set_rand (uint seed);

Fill the vector with the constant `f` or with random numbers. The parameter `seed` is used to initialise the random number generator. Please note, that the result of the random filling might be dependent on the number of processors.

TVector & scale (real f);

Scale the vector by `f`.

TVector & assign (real f, const TVector * x);

Assign $f \cdot x$ to the local vector object.

```
real dot ( const TVector * x ) const;
```

Compute the dot product of the local vector with **x**.

```
real norm2 () const;
```

Compute the Euclidean norm of the local vector.

```
TVector & axpy ( real alpha, const TVector * x );
```

Perform *a times x plus y*, i.e. compute the linear combination $y := \alpha x + y$, where **y** equals the local vector.

```
uint type () const;
```

```
bool is_type ( uint t ) const;
```

```
string typestr () const;
```

Give access to the **RTTI** system for vectors.

```
ulong byte_size () const;
```

Return the amount of memory the vector, together with all possible sub-vectors, occupies.

```
TVector & operator = ( const TVector & v );
```

Assign vector **v** to the local vector.

```
TVector * copy () const = 0;
```

```
TVector * clone () const = 0;
```

Create a copy, e.g. with all data, or a clone, e.g. an object of the same type but without data.

```
void print ( ostream & os, uint ofs ) const;
```

Print information of the vector to the stream **os**. With **ofs** an optional offset can be specified.

If the argument of a method is a vector, the correct datatype is checked before computing the result. If an invalid type is detected, a corresponding error message is printed.

4.4.2 TScalarVector

A scalar vector is the typical example of a vector defined over an indexset. It consists of a continuous array holding the coefficients.

```
class TScalarVector : public TVector {
protected:
    real * _data;
    uint  _size;

public:
    TScalarVector ();
    TScalarVector ( uint n, uint ofs, real f );
    TScalarVector ( const TCluster * c, real f );
    TScalarVector ( const TScalarVector & v );
```

```
void set_size ( uint n, bool copy );

real operator [] ( uint i ) const;
real & operator [] ( uint i );

real * data ();
const real * data () const;

void set_cluster ( const TCluster * c )
void set_vector ( const real * data, uint size, uint ofs );
}
```

Objects of type `TScalarVector` behave similar to dynamic arrays. Due to the storage format, efficient methods for linear algebra can be used as they are defined in *BLAS*.

TScalarVector ();

Construct an empty vector.

TScalarVector (uint n, uint ofs, real f);

Construct a vector of size `n` with offset `ofs` and all coefficients initialised to `f`, which is 0.0 by default.

TScalarVector (const TCluster * c, real f);

Construct a vector with informations about the indexset defined by the cluster `c` and all coefficients initialised to `f` (again, 0.0 by default).

TScalarVector (const TScalarVector & v);

Construct a scalar vector as a copy of `v`.

void set_size (uint n, bool copy);

Set the size of the scalar vector to `n`. If `copy` equals `true`, the data is preserved.

real operator [] (uint i) const;

real & operator [] (uint i);

Give direct access to a coefficient in the vector.

real * data ();

const real * data () const;

Return a pointer to the actual memory area holding the vector.

void set_cluster (const TCluster * c)

Initialise the scalar vector by the given cluster `c`, e.g. use the offset and size information of the cluster.

void set_vector (const real * data, uint n, uint ofs);

Initialise the vector to be of size `n` with offset `ofs` and copy the content from `data`.

Sometimes it is necessary to work only with parts of an array, but at the same time still have an vector object at hand. Since it would be inefficient to copy this specific part each time, *virtual* vectors are introduced in PHI as objects of type `TVirtualVector`. They are similar to scalar vectors, e.g. they store a pointer to a memory block representing the coefficients, but do not allocate or free this memory.

```
class TVirtualVector : public TScalarVector {
public:
    TVirtualVector ();
    TVirtualVector ( const TVirtualVector & v )
    TVirtualVector ( const real * data, uint size, uint ofs );
    TVirtualVector ( const real * data, const TCluster * cluster );
    TVirtualVector ( const TScalarVector * x, const TCluster * cluster );

    void set_vector ( const real * data, uint size, uint ofs = 0 );
    void set_vector ( const real * data, const TCluster * cluster );
    void set_vector ( const TScalarVector * x, const TCluster * cluster );
}
```

Due to the inheritance from `TScalarVector`, each virtual vector is also a scalar vector. The added functions mainly concern the assignment of the corresponding data.

TVirtualVector ();

Construct an empty vector.

TVirtualVector (const TVirtualVector & v);

Construct a vector as the copy of `v`.

TVirtualVector (const real * data, uint size, uint ofs);

Construct a vector of size `n`, offset `ofs` and coefficients stored in `data`, whereby the latter is not copied.

TVirtualVector (const real * data, const TCluster * cluster);

Construct a virtual vector with the indexset information defined by `cluster`.

TVirtualVector (const TScalarVector * x, const TCluster * cluster);

Construct a virtual vector with the indexset information defined by `cluster` which has to be contained in the indexset over which `x` is defined.

void set_vector (const real * data, uint size, uint ofs = 0);

void set_vector (const real * data, const TCluster * cluster);

void set_vector (const TScalarVector * x, const TCluster * cluster);

Corresponding initialisation methods for setting the content of a virtual vector.

4.4.3 Blockvectors

Just like blockmatrices, blockvectors hold a collection of subblocks and provide management functions for handling these.

```
class TBlockVector : public TVector {
protected:
    TArray< TVector * > _blocks;

public:
    uint n_blocks () const;

    TVector * block ( uint i );

    void set_block ( uint i, TVector * v );

    void set_block_struct ( uint i );
}
```

In contrast to blockmatrices, which are most of the time defined over a block cluster tree and therefore represent the hierarchy of this tree, there is no such correspondence between blockvectors and cluster trees. For now, they only serve as a management tool for partitioned vectors (see also `TDDVector`).

uint n_blocks () const;

Return the number of subvectors.

TVector * block (uint i);

Return the *i*'th subvector or `NULL`, if no such vector exists.

void set_block (uint i, TVector * v);

Set the *i*'th subvector to *v*.

void set_block_struct (uint i);

Set the number of subblocks. Old subvectors are kept, if possible.

4.4.4 Vectors for Domain Decomposition

To represent vectors in the context of domain decomposition or nested dissection (see `TDDVector`), the class `TDDVector` was implemented. It holds local portions of a distributed vector. For *p* subdomains, the resulting blockstructure of the vector is as follows:

$$(v_1 \ v_2 \ \cdots \ v_p \ v_{p+1})^T$$

The interface part is stored in the vector v_{p+1} .

```
class TDDVector : public TBlockVector {
protected:
    TCluster * _tau;
    uint _pid;
    TProcSet _procs;

public:
    TDDVector ( const TProcSet & ps, TCluster * tau );
}
```

```

void init ( const TProcSet & ps, TCluster * tau );

uint pid      () const;
void set_pid ( uint i );

TProcSet & procs ();
uint      nprocs ();
}

```

`TDDVector` extends `TBlockVector` with a local processor number and a processor set. Furthermore, the local cluster is stored in the vector.

```

TDDVector ( const TProcSet & ps, TCluster * tau );
void init ( const TProcSet & ps, TCluster * tau );

```

Construct a domain decomposition cluster over the cluster `tau` on the processors defined by `ps`. If `ps` contains more than one processor, then the number of sons of `tau` has to be identical with the number of processors in `ps` plus one.

```

uint pid () const;
void set_pid ( uint i );

```

Give access to the local processor number.

```

TProcSet & procs ();
uint nprocs ();

```

Give access to the processor set and the number of local processors.

4.5 Algebra

So far, only the datatypes for representing matrices and vectors were discussed. But the real power of \mathcal{H} -matrices lies in the efficient matrix algebra, e.g. matrix-vector multiplication, matrix addition, multiplication, inversion and various factorisation methods. Classes which are related to this topic shall be introduced in this section.

Although slightly contrary to the object-oriented programming paradigm, many algorithms regarding the algebra of matrices are encapsulated in special classes and not included in the matrix classes itself. The reason for this is the complexity of the methods, which would clutter up the implementation of the matrices.

4.5.1 Matrix Building

For standard \mathcal{H} -matrices the process of building is split into creating the blockmatrices corresponding to inner nodes and constructing the matrices for the leaves in the block-clustertree. Since the latter ones are the only objects which store data, building them is application dependent. The block-matrices on the other hand are completely defined by the structure of the block-clustertree.

The described process is implemented in the class `THMBuilder`. Only the application dependent parts are left out, defining just the interface.

```
class THMBuilder {
protected:
    bool _coarsening;
    real _epsilon;

public:
    THMBuilder ( bool coarsening, real eps );

    void set_coarsening ( bool b, real eps );

    TMatrix * build ( TBlockCluster * cluster,
                    uint nprocs,
                    uint root_type,
                    TProgressBar * progress ) const;

protected:
    TMatrix * rec_build ( TBlockCluster * cluster,
                        uint type,
                        TSSL< mat_link_t > * leaves,
                        TProgressBar * progress ) const;

    TBlockMatrix * build_blocked ( TBlockCluster * t,
                                  uint type ) const;

    TMatrix * build_leaf ( TBlockCluster * t ) const = 0;
}
```

Beside the actual construction of \mathcal{H} -matrices, the class `THMBuilder` also implements *coarsening* of the created matrices, e.g. further compression (see ???). The compression rate can be controlled by the parameter `eps`.

The algorithm is implemented in the method `rec_build`, which follows the block-clustertree recursively and calls `build_blocked` for each inner node.

THMBuilder (bool coarsening, real eps);

Construct a matrix building object with optional coarsening up to precision `eps`.

void set_coarsening (bool b, real eps);

Define coarsening behaviour and precision.

TMatrix * build (c, p, t, progress) const;

Construct a matrix defined by the blockcluster `c` on `p` processors (default: $p = 1$) with matrix type `t`, which is one of the types defined by the `RTTI` system. The optional argument `progress` specifies a progress bar indicating the current status of the matrix construction.

TMatrix * build_leaf (TBlockCluster * t) const = 0;

Interface which is called to build a matrix corresponding to a leaf in the block-clustertree. This method has to be overwritten with an application-specific implementation.

4.5.2 Matrix-Vector Multiplication

Although each matrix class has a method for matrix-vector multiplication, some of the involved algorithms are implemented in the special class `TMatrixVec`, e.g. the implementation of `mul_vec` in `THMatrix` is just a wrapper for the methods in `TMatrixVec`.

```
class TMatrixVec {
public:
    error_t multiply ( uint p,
                      real alpha, const TMatrix * A, const TVector * x,
                      real beta, TVector * y,
                      bool trans = false ) const;
}
```

The interface for the multiplication method in `TMatrixVec` is similar to that of the matrix classes and consists mainly of this single function.

```
error_t multiply ( p, alpha, A, x, beta, y, trans ) const;
```

Compute the product $y := \alpha Ax + \beta y$ on `p` processors. If `trans` equals `true`, the transpose of `A` is used.

4.5.3 Matrix Addition

The addition method included in the matrix classes computes a scaled update to the local matrix. The algorithm implemented in `TMatrixAdd` implements a similar method, whereby both matrices can be scaled. Furthermore, one can restrict the part of the matrix which is included in the computation to the upper or lower triangular part.

```
class TMatrixAdd {
public:
    error_t add ( real alpha, const TMatrix * A,
                 real beta, TMatrix * C,
                 char mode ) const;

    error_t add ( uint p,
                  real alpha, const TMatrix * A,
                  real beta, TMatrix * C,
                  char mode ) const;
}
```

Since in `BSP mode` the number of processors are defined by the parallel machine, only sequential operations are supported, e.g. $p = 1$ in the second function.

```
error_t add ( p, alpha, A, beta, C, mode ) const;
```

```
error_t add ( alpha, A, beta, C, mode ) const;
```

Compute $C := \alpha A + \beta C$ on (optional) `p` processors.

4.5.4 Matrix Multiplication

Each matrix class has functions for the multiplication with other matrices either as the left or the right factor (see `TMatrix`, `mul_left` and `mul_right`). In contrast to this, `TMatrixMult` implements a general matrix multiplication of the form

$$C := \alpha AB + \beta C$$

which also includes the special variants of the matrix classes. Furthermore, the arguments `A` and `B` can be multiplied in a transposed form.

```
class TMatrixMult {
public:
    error_t multiply ( real alpha,
                     bool trans_A, const TMatrix * A,
                     bool trans_B, const TMatrix * B,
                     real beta , TMatrix * C, char mode = 'N',
                     TProgressBar * progress = NULL ) const;

    error_t multiply ( real alpha, const TMatrix * A, const TMatrix * B,
                     real beta , TMatrix * C, char mode = 'N',
                     TProgressBar * progress = NULL ) const;

    error_t multiply ( uint p, real alpha,
                     bool trans_A, const TMatrix * A,
                     bool trans_B, const TMatrix * B,
                     real beta , TMatrix * C, char mode = 'N',
                     TProgressBar * progress = NULL ) const;

    error_t multiply ( uint p,
                     real alpha, const TMatrix * A, const TMatrix * B,
                     real beta , TMatrix * C, char mode = 'N',
                     TProgressBar * progress = NULL ) const;
}
```

Beside the general form with transposable arguments, abbreviations are available for the multiplication of the non-transposed form. The last two methods provide parallel matrix multiplications on `p` processors.

The optional argument `mode` restricts the computation on the whole matrix `C` (`mode = 'N'`), the lower triangular (`mode = 'L'`) and the upper triangular part (`mode = 'U'`).

If a non-NULL argument for `progress` was supplied, the current status of the multiplication is displayed.

```
error_t multiply ( alpha, trans_A, A, trans_B, B, beta , C, mode, progress
) const;
```

Compute $C := \alpha AB + \beta C$. If `trans_A` or `trans_B` equal `true`, the matrices `A` and `B` are transposed, respectively.

4.5.5 Matrix Inversion

There exists no method in the interface of the matrix classes which corresponds to the inversion of a matrix as it was the case with the previous operations. Instead, the inversion algorithm is purely implemented in the class `TMatrixInv`.

```
class TMatrixInv {
public:
    error_t invert (          TMatrix * A, TMatrix * C, TMatrix * T ) const;
    error_t invert ( uint p, TMatrix * A, TMatrix * C, TMatrix * T ) const;
}
```

Again, the parallel method is available only if PHI was compiled in `PThread mode`. Otherwise, all operations will be done sequentially or dependent on the type of matrix, e.g. a distributed domain decomposition matrix is handled in parallel.

```
error_t invert ( TMatrix * A, TMatrix * C, TMatrix * T ) const;
    Compute the inverse of A and store the result in C. A might be changed during
    the operation. T is used for temporary results and can be NULL.
```

```
error_t invert ( uint p, TMatrix * A, TMatrix * C, TMatrix * T ) const;
    Same as above, but do the computations on p processors (if in PThread mode).
```

4.5.6 LU and Cholesky Factorisation

LU factorisation and the symmetric Cholesky decomposition are implemented in PHI in the classes `TLU` and `TCholesky`.

The definition of these classes is

```
class TLU {
protected:
    bool _use_block;

public:
    TLU ( bool use_block );

    error_t factorise ( uint p,
                       TMatrix * A,
                       TProgressBar * progress ) const;

    error_t factorise ( TMatrix * A,
                       TProgressBar * progress ) const;

    error_t solve ( const TMatrix * A, TVector * x, bool trans ) const;
}
```

and

```
class TCholesky {
public:
    error_t factorise ( uint p, TMatrix * A ) const;
    error_t factorise (          TMatrix * A ) const;

    error_t solve ( const TMatrix * L, TVector * x ) const;
}
```

Beside the actual decomposition method, both classes also provide a function `solve` the resulting system, e.g. evaluate a vector with the inverse of the original matrix. It should be noted that the matrix `A` will be overwritten with the factorisation result.

If the flag `use_block` is set to `true`, a block-LU factorisation is computed, i.e. diagonal matrix blocks will be inverted. This slightly increases the stability of the algorithm without compromising the overall execution speed.

The parallel versions of the decomposition methods are only available in the `PThread mode` of PHI.

TLU (bool use_block);

Construct a LU object with optional inversion of diagonal matrix blocks.

error_t factorise (TMatrix * A, TProgressBar * progress) const;

Compute the LU decomposition of `A`. In case of a symmetric `A`, a Cholesky decomposition is computed. If `progress` is not `NULL`, the current status of the factorisation is shown with this progress meter. After the computation, `A` holds the result.

error_t solve (const TMatrix * A, TVector * x, bool trans) const;

Evaluate the decomposition stored in `A` with the vector `x`, which will be overwritten with the result. Again, if `A` is symmetric, the corresponding Cholesky method is used instead. If `trans` is `true`, `A` is transposed before evaluation.

The corresponding methods of the `TCholesky` class behave exactly like their pendants in `TLU`. Only due to the symmetric nature of the decomposition, the option for transposed operation is not available for solving a Cholesky decomposition.

4.5.7 Computing the Norm of a Matrix

A common task in the context of matrices is the computation of their different norms. In PHI this is accomplished with objects of type `TMatrixNorm` and derivative classes. Currently the Frobenius and the spectral norm are implemented as `TFrobeniusNorm` and `TSpectralNorm`.

The interface to `TMatrixNorm` is

```
class TMatrixNorm {
public:
```

```
TMatrixNorm ();  
~TMatrixNorm ();  
  
real norm ( const TMatrix * A ) const = 0;  
real operator () ( const TMatrix * A ) const;  
}
```

The actual norm computing method is defined as a pure virtual function and has therefore to be overwritten in derived classes.

```
real norm ( const TMatrix * A ) const;  
    Return the norm of the matrix A.  
real operator () ( const TMatrix * A ) const;  
    Abbreviation to the norm method.
```

4.6 Input and Output

4.6.1 Printer Classes

PHI supports several output mechanisms including Postscript, X11 and VRML. For this, different *printer classes* have been implemented. The main difference between these types is the output dimension, e.g. Postscript and X11 only support 2d output, whereas VRML is more appropriate for 3d data.

4.6.1.1 2D Printer Classes

All 2d bases printers are derived from the class *T2DPrinter*. It defines the basic interface for drawing various shapes, e.g. lines, triangles, circles, text etc. It also implements the routines for the transformation between the coordinate system of the user, e.g. of the data to print also denoted by the *world*, and the coordinate system of the output media, denoted by *window*.

```
class T2DPrinter {  
public:  
    T2DPrinter ();  
    ~T2DPrinter ();  
  
    TViewport & viewport ();  
    const TViewport & viewport () const;  
  
    void begin () = 0;  
    void end () = 0;  
  
    void draw_point ( real x, real y ) = 0;  
    void draw_line ( real x1, real y1, real x2, real y2 ) = 0;
```

```
void draw_triangle ( real x0, real y0,
                    real x1, real y1,
                    real x2, real y2 ) = 0;
void fill_triangle ( real x0, real y0,
                    real x1, real y1,
                    real x2, real y2 ) = 0;

void draw_rect ( real x1, real y1, real x2, real y2 ) = 0;
void fill_rect ( real x1, real y1, real x2, real y2 ) = 0;

void draw_circle ( real x, real y, real radius ) = 0;
void fill_circle ( real x, real y, real radius ) = 0;

void move_to ( real x, real y );
void line_to ( real x, real y );

void draw_text( real x, real y, const char * text,
               justification_t just = JUST_LEFT,
               real angle = 0.0 ) = 0;

void set_gray ( int g );
void set_rgb  ( int r, int g, int b ) = 0;
void set_hsv  ( int h, int s, int v );

void set_line_width ( real width ) = 0;
void set_font      ( const char * font, real size ) = 0;
}
```

Most of the methods are pure virtual since their implementation is strongly dependent on the actual output media.

TViewport & viewport ();

const TViewport & viewport () const;

Give access to the viewport object, e.g. the transformation between world and window coordinates.

void begin ();

void end ();

Start and end drawing.

void draw_point (real x, real y);

Draw a point at the point (x,y).

void draw_line (real x0, real y0, real x1, real y1);

Draw a line from (x0,y0) to (x1,y1).

void draw_triangle (real x0, y0, x1, y1, x2, y2);

void fill_triangle (real x0, y0, x1, y1, x2, y2);

Draw a (filled) triangle defined by the three points (x0,y0), (x1,y1) and (x2,y2).

void draw_rect (real x0, real y0, real x1, real y1);

void fill_rect (real x0, real y0, real x1, real y1);

Draw a (filled) axis-aligned rectangle the points (x0,y0) and (x1,y1).

void draw_circle (real x, real y, real r);

void fill_circle (real x, real y, real r);

Draw a (filled) circle with the middle point (x,y) and the radius r.

void move_to (real x, real y);

Move an internal pointer to the point (x,y).

void line_to (real x, real y);

Draw a line from the internal pointer to the (x,y) and move the pointer to the later coordinates.

void draw_text(x, y, text, justification_t just, real angle);

Print the string `text` and the coordinate (x,y) with the text-justification `just` and rotated by `angle`. The justification field can be one of `JUST_LEFT` (default), `JUST_RIGHT` and `JUST_CENTER`. The values for angle are interpreted as degrees.

void set_gray (int g);

void set_rgb (int r, int g, int b);

void set_hsv (int h, int s, int v);

Set the drawing and filling color to the gray value `g` (range: 0...255, the red/green/blue value defined by `r`, `g` and `b` (range: 0...255) or the HSV value defined by `h` (range: 0...360), `s` and `v` (range: 0...255).

void set_line_width (real width);

Set the width of the line to `width`.

void set_font (const char * font, real size);

Define the font for drawing text. The following fonts are supported: *Times*, *Courier*, *Helvetica* and *Symbol*.

The Postscript printer `TPSPrinter` has further options regarding the output format including the papersize and the file to write to. Beside the Postscript output, it also supports *Encapsulated* Postscript, which is better suited for including into text documents, e.g. LaTeX. Please note, that encapsulated Postscript does not have a *showpage* command, e.g. most printers do not print these files.

```
class TPSPrinter : public T2DPrinter {
public:
    TPSPrinter ( int format = PRINT_EPS );
    TPSPrinter ( const string & name, int format = PRINT_EPS );

    void set_paper ( const char * paper );

    const string & filename () const;

    virtual void begin ( const string & name );
}
```

The two options for the specification of the Postscript type are `PRINT_PS` and `PRINT_EPS`. Most of the known paper formats are supported, whereby the default is *A4*.

TPSPrinter (int format = PRINT_EPS);

Construct a Postscript printer object with the defined output format.

TPSPrinter (const string & name, int format = PRINT_EPS);

Construct a Postscript printer object with the output file `name` and the given output format.

void set_paper (const char * paper);

Set the paper format to use for printing. Has to be called *before* starting to print, e.g. before `begin` is executed.

const string & filename () const;

Return the name of the file to write to.

virtual void begin (const string & name);

Begin printing to the file `name`.

If PHI was compiled with X11 support, the X11 printer comes in the form of `TX11Printer`. To actually use it, of course a corresponding X11 implementation has to run on the computer in use.

```
class TX11Printer : public T2DPrinter {
public:
    TX11Printer ( int w, int h );
}
```

The only difference in the interface compared to a general 2d printer is the initialisation which expects the dimension of the window to use.

TX11Printer (int w, int h);

Construct a X11 printer with a window size of `w` times `h`.

Inside the X11 printer window, some keyboard commands are available:

P Print the current data to the file `screen.ps` in Postscript format.

Q Quit the X11 printer window and continue program execution.

4.6.1.2 3D Printer Classes

The only 3d output format is *VRML* v1.0 (see <http://www.web3d.org>). The functions which are available in the corresponding class `TVML` are similar to their 2d variants but expect 3d coordinates as arrays.

In contrast to the 2d printer, no conversion between the user coordinates and a VRML coordinate system is applied.

```
class TVRML {
public:
    TVRML ();
    ~TVRML ();

    void begin ( const string & name );
    void end   ();

    void draw_line ( real * orig, real * dest );

    void draw_triangle ( const real * x0, const real * x1, const real * x2 );
    void fill_triangle ( const real * x0, const real * x1, const real * x2 );

    void draw_rect ( const real * x0, const real * x1,
                    const real * x2, const real * x3 );
    void fill_rect ( const real * x0, const real * x1,
                    const real * x2, const real * x3 );

    void draw_sphere ( real * x, real r );

    void set_gray ( int g );
    void set_rgb  ( uint r, uint g, uint b );
    void set_hsv  ( int  h, uint s, uint v );
}
```

The output of the printer is written to a file which can then be viewed by any VRML viewer.

void begin (const string & name);

Start the VRML printer and write all output to file **name**.

void end ();

Finish the VRML printer and close the written file.

void draw_line (real * orig, real * dest);

Draw an line from **orig** to **dest**.

void draw_triangle (const real * x0, x1, x2);

void fill_triangle (const real * x0, x1, x2);

Draw a (filled) triangle with the three points **x0**, **x1** and **x2**.

void draw_rect (const real * x0, x1, x2, x3);

void fill_rect (const real * x0, x1, x2, x3);

Draw a (filled) rectangle with the points **x0**, **x1**, **x2** and **x3**.

void draw_sphere (real * x, real r);

Draw a sphere with the center **x** and the radius **r**.

void set_gray (int g);

void set_rgb (uint r, uint g, uint b);

void set_hsv (int h, uint s, uint v);

Set the drawing and filling color to the gray value $g \in [0, 255]$, the red/green/blue value defined by $r, g, b \in [0, 255]$ or the HSV (hue, saturation and value) value defined by $h \in [0, 360]$ and $s, v \in [0, 255]$.

4.6.2 Cluster and Blockcluster I/O

The output of clusters and blockclusters uses the different printer classes (**Printer**) to print the corresponding trees. To change the type of output, several options are available, which can be combined (ORed):

CTIO_COMP	Show complete tree.
CTIO_ID	Print IDs of nodes.
CTIO_TREE	Print tree at all.
CTIO_PROCS	Print processor information of nodes.
CTIO_SINGLE	Print tree for one processor.

The actual output is accomplished by derivatives of the class **TClusterIO**.

```
class TClusterIO {
public:
    TClusterIO ();
    ~TClusterIO ();

    void write_ct ( const TCluster * c,
                   const string & name,
                   uint          opt = CTIO_TREE ) const;

    void write_bc ( const TBlockCluster * c,
                   const string & name,
                   uint          opt = 0,
                   int          pid = -1 ) const;
}
```

The interface is restricted to the output of clustertrees and block clustertrees.

void write_ct (c, name, opt) const;

Print the given clustertree **c** to the file **name** with the options **opt**.

void write_bc (c, name, opt, pid) const;

Print the given block clustertree **c** to the file **name** with the options **opt**. The parameter **pid** indicates the specific processor for the options **CTIO_SINGLE**.

Usable implementations of cluster I/O can be found in **TPSClusterIO** and **TVRMLClusterIO**.

4.6.3 Matrix Input and Output

Similar to the output of clusters works the matrix related I/O. Again, the type of output can be defined by combinable options:

MATIO_NONEMPTY	Write only nonempty blocks.
MATIO_ENTRY	Write matrix coefficients.
MATIO_PATTERN	Write pattern of matrix coefficients.
MATIO_SVD	Write singular value decomposition of matrix blocks.
MATIO_HLEAVES	Write leaves of given H-matrix.

The base class for all matrix I/O is [TMatrixIO](#):

```
class TMatrixIO {
public:
    TMatrixIO ();
    ~TMatrixIO ();

    void write ( const TMatrix * A, const string & name, uint opt = 0 ) const = 0;
}

```

It defines a pure virtual function for printing matrices.

```
void write ( const TMatrix * A, const string & name, uint opt ) const;
    Print matrix A to the file name with the options opt.
```

Implementations for this are [TPSMatrixIO](#) for Postscript output, [TOctaveMatrixIO](#) for writing files in the Octave/Matlab format and [TASCIIMatrixIO](#) for writing matrices in a text representation. For handling sparse matrices, stored in the *AMG* format, the class [TAMGMatIO](#) can be used. Beside the matrix-IO, it also has functions for reading and writing vertices and vectors:

```
class TAMGMatIO : public TMatrixIO {
public:
    error_t write ( const TMatrix * A, const string & name, uint opt ) const;

    error_t write_vertices ( const TArray< double * > & vertices,
                           uint dim,
                           const string & name ) const;

    error_t write_rhs ( const TScalarVector * v, const string & name ) const;
    error_t write_sol ( const TScalarVector * v, const string & name ) const;

    TMatrix * read ( const string & name ) const;

    error_t read_vertices ( TArray< double * > & vertices,
                          uint & dim,
                          const string & name ) const;

    TVector * read_rhs ( const string & name ) const;
    TVector * read_sol ( const string & name ) const;
}

```

The argument `name` does not contain the filename itself, but just the basename, e.g. without suffix. The reason for this is, that the AMG format splits the data into several files.

```
error_t write ( TMatrix * A, string & name, uint opt ) const;
```

Write sparse matrix `A` into file "`name.amg`" with format file "`name.frm`".

```
error_t write_vertices ( vertices, dim, name ) const;
```

Write coordinates from `vertices` into file "`name.coo`". Each coordinates has to have the dimension `dim`.

```
error_t write_rhs ( v, name ) const;
```

```
error_t write_sol ( v, name ) const;
```

Write the scalar vector `v` into file "`name.rhs`" or "`name.sol`" respectively.

```
TMatrix * read ( string & name ) const;
```

Read sparse matrix from file "`name.amg`".

```
error_t read_vertices ( vertices, uint & dim, string & name ) const;
```

Read coordinates from "`name.coo`" into array `vertices`. The dimension of the coordinates is stored into `dim`.

```
TVector * read_rhs ( string & name ) const;
```

```
TVector * read_sol ( string & name ) const;
```

Read scalar vector from file "`name.rhs`" or "`name.sol`", respectively.

Bibliography

- [GBK05] L. Grasedyck, S. Le Borne, and R. Kriemann. Parallel black box domain decomposition based \langle -lu preconditioning. Technical Report 115, Max-Planck-Institute for Mathematics i.t.S., 2005. submitted to *Mathematics of Computation*:
- [Hac99] W. Hackbusch. A sparse matrix arithmetic based on \mathcal{H} -matrices. I. Introduction to \mathcal{H} -matrices. *Computing*, 62(2):89–108, 1999.
- [HKK04] W. Hackbusch, B.N. Khoromskij, and Ronald Kriemann. Hierarchical matrices based on a weak admissibility criterion. *Computing*, 73:207–243, 2004.
- [MN98] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, January 1998.