Contents

The ABCD Solver

The package `ABCD Solver` is a distributed hybrid (iterative/direct) solver for sparse linear systems $Ax = b$ where A is a double precision [1] square [2] matrix with any structure. `ABCD Solver` uses two methods to solve the linear system:

- *Regular Block Cimmino*: A block-projection technique that iterates to solve the linear system. During the iterations it solves a set of small problems (augmented systems built using the partitions of the original system).
- *Augmented Block Cimmino*: A pseudo-direct technique that augments the original system and through a succession of direct solves finds the solution.

The solver is in the form of a class called `abcd` and an instance of it represents an instance of the solver with its own linear system. In the following we will use the `Boost::MPI` syntax for `MPI` specefic commands. Moreover, in the following description we refer to the members of the class by `obj.member` where `obj` is an instance of the solver and `member` is the corresponding member, similarly, we refer to the public methods by `obj.method()`. Finally, arrays will have `[]` appended to them, if we specify a size then the array is pre-initialized at construction, otherwise it is either allocated by the user (such as the linear system entries) or by the solver (such as the solution).

An introductory example

To see how the solver can be used, we expose a basic example that uses the regular block Cimmino scheme. We comment the interesting parts, and explain how they fit together, the details regarding the members of the `abcd` class are explained in The linear system and the controls are detailed in The Controls.

```
#include "abcd.h"
// use boost::mpi for simplicity, the user can use which ever he wants
#include "mpi.h"
#include <boost/mpi.hpp>

int main(int argc, char* argv[])
{
    mpi::environment env(argc, argv);
    // obtain the WORLD communicator, by default the solver uses it
    mpi::communicator world;
```

[1] The current implementation is double precision only.

[2] In reality the system can rectangular, but this case is not fully tested yet.

```
// create one instance of the abcd solver per mpi-process
abcd obj;

if(world.rank() == 0) { // the master
    // we create a 5x5 matrix for a 1D mesh + three-point stencil
    obj.sym = true; // the matrix is symmetric
    obj.m = 10; // number of rows
    obj.n = obj.m; // number of columns
    obj.nz = 2*obj.m - 1; // number of nnz in the lower-triangular part

    // allocate the arrays
    obj.irn = new int[obj.nz];
    obj.jcn = new int[obj.nz];
    obj.val = new double[obj.nz];

    // initialize the matrix
    // Notice that the matrix is stored in 1-based format
    size_t pos = 0;
    for (size_t i = 1; i < obj.m; i++) {
        // the diagonal
        obj.irn[pos] = i;
        obj.jcn[pos] = i;
        obj.val[pos] = 2.0;
        pos++;

        // the lower-triangular part
        obj.irn[pos] = i + 1;
        obj.jcn[pos] = i;
        obj.val[pos] = -1.0;
        pos++;
    }

    // the last diagonal element
    obj.irn[pos] = obj.m;
    obj.jcn[pos] = obj.m;
    obj.val[pos] = 2.0;

    pos++;

    // set the rhs
    obj.rhs = new double[obj.m];
    for (size_t i = 0; i < obj.m; i++) {
        obj.rhs[i] = ((double) i + 1)/obj.m;
    }

    // ask the solver to guess the number of partitions
    obj.icntl[Controls::part_guess] = 1;
}

try {
    // We call the solver directly using the object itself
    // (the abcd class is a functor)
    obj(-1); // initialize the object with defaults
```

```
            obj(5); // equivalent to running 1, 2 and 3 successively
            // the solution is stored in obj.sol
        } catch (runtime_error err) {
            // In case there is a critical error, we throw a runtime_error exception
            cout << "An error occured: " << err.what() << endl;
        }

        delete[] obj.irn;
        delete[] obj.jcn;
        delete[] obj.val;

      return 0;
    }
```

The linear system

The definition of the linear system uses 7 members:

- `obj.m` (type: `int`), the number of rows.
- `obj.n` (type: `int`), the number of columns.
- `obj.nz` (type: `int`), the number of entries.
- `obj.sym` (type: `bool`), the symmetry of the matrix. If the matrix is symmetric, the matrix must be given in a lower-triangular form.
- `obj.irn` (type: `int *`), the row indices.
- `obj.jcn` (type: `int *`), the column indices.
- `obj.val` (type: `double *`), the matrix entries.
- `obj.rhs` (type: `double *`), the right-hand side.

If either of the row and column indices start with `0` the arrays are supposed to be zero based (`C` arrays indexation), otherwise, if they start with `1` the arrays are supposed to be one based (`Fortran` arrays indexation). If however, none starts with `0` or `1` then there is either an empty row or an empty column and the solver stops.

```
    // Create an object for each mpi-process
    abcd obj;
    obj.n = 7;
    obj.m = 7;
    obj.nz = 15;
    obj.sym = false;
    // put the data in the arrays
    obj.irn[0] = 1;
    //..
```

The Controls

Define the general behavior of the solver. They are split into two arrays, `icntl` and `dcntl`. `icntl` is an *integer* array and defines the options that control the specific parts of the solver, such as the scaling, the type of algorithm to run and so on. `dcntl` is a *double precision* array and defines some of the options required by the algorithms we use such as the imbalance between the partition sizes and the stopping criteria of the solver.

To access each of the control options we can either use the indices `0, 1, ..` or, preferably, use the *enums* defined in the header `defaults.h`. To access them, the user can use the namespace `Controls`, eg. `Controls::scaling` has a value of `5` and is used with `icntl` to handle the scaling of the linear system.

The integer control array

- `obj.icntl[Controls::nbparts]` or `obj.icntl[1]` defines the number of partitions in our linear system, can be from `1` to `m` (the number of rows in the matrix)

```
// we have 8 partitions
obj.icntl[Controls::nbparts] = 8;
```

- `obj.icntl[Controls::part_type]` or `obj.icntl[2]` defines the partitioning type. It can have the values:

  ~ `1`, manual partitioning, the *nbparts* partitions can be provided into the STL vector `obj.nbrows[]`. Example:

```
// use manual partitioning
obj.icntl[Controls::part_type] = 1;
// say that we want 20 rows per partition
obj.nrows.assign(obj.icntl[Controls::nbparts], 20);

// or
obj.nrows.resize(obj.icntl[Controls::nbparts]);
obj.nrows[0] = 20;
obj.nrows[1] = 20;
//...
```

  ~ `2` *(default)*, automatic uniform partitioning, creates *nbparts* partitions of similar size.

```
// use patoh partitioning
obj.icntl[Controls::part_type] = 2;
```

  ~ `3`, automatic hypergraph partitioning, creates *nbparts* partitions using the hypergraph partitioner `PaToH`. The imbalance between the partitions is handled using `obj.dcntl[Controls::part_imbalance]`. Example:

```
// use patoh partitioning
obj.icntl[Controls::part_type] = 3;
// say that we want an imbalance of 0.3 between the partitions
obj.dcntl[Controls::part_imbalance] = 0.3;
```

- `obj.icntl[3]` reserved for a future use.
- `obj.icntl[Controls::part_guess]` or `obj.icntl[4]` asks the solver to guess the appropriate number of partitions and overrides the defined *nbparts*.

  ~ `0` *(default)*, no guess

  ~ `1`, guess

- `obj.icntl[Controls::scaling]` or `obj.icntl[5]` defines the type of scaling to be used.

  ~ `0`, no scaling

  ~ `1`, infinity norm `MC77` based scaling

  ~ `2` *(default)*, combination of one norm and two norm `MC77` based scaling

- `obj.icntl[Controls::itmax]` or `obj.icntl[6]` defines the maximum number of iterations in block-CG acceleration, default is `1000`
- `obj.icntl[Controls::block_size]` or `obj.icntl[7]` defines the block-size to be used by the block-CG acceleration, default is `1` for classical CG acceleration
- `obj.icntl[Controls::verbose_level]` or `obj.icntl[8]` **Not Yet Implemented**, defines how verbose the solver has to be.
- `obj.icntl[9]` reserved for a future use.
- `obj.icntl[Controls::aug_type]` or `obj.icntl[10]` defines the augmentation type.

  ~ `0` *(default)*, no augmentation. This makes the solver run in **regular block Cimmino** mode.

  ~ `1`, makes the solver run in **Augmented Block Cimmino** mode with an augmentation of the matrix using the $C_{ij}/-I$ technique. For numerical stability, this augmentation technique has to be used with a scaling.

  ~ `2`, makes the solver run in **Augmented Block Cimmino** mode with an augmentation of the matrix using the $A_{ij}/-A_{ji}$ technique. This is the prefered augmentation technique.
- `obj.icntl[Controls::aug_blocking]` or `obj.icntl[11]` defines the blocking factor when building the auxiliary matrix S, default is `128`.
- `obj.icntl[Controls::aug_analysis]` or `obj.icntl[12]`, when set to a value different than `0`, analyses the number of columns in the augmentation.
- `obj.icntl[13]` to `obj.icntl[16]` are for development and testing purposes only.
- `obj.icntl[17]` to `obj.icntl[19]` are reserved for a future use.

The double precision control array

- `obj.dcntl[Controls::part_imbalance]` or `obj.dcntl[1]` defines the imbalance between the partitions when using `PaToH` (`obj.icntl[Controls::part_imbalance] = 3`).
- `obj.dcntl[Controls::threshold]` or `obj.dcntl[2]` defines the stopping threshold for the block-CG acceleration, default is `1e-12`.
- `obj.dcnlt[3]` to `obj.dcntl[20]` are reserved for future use.