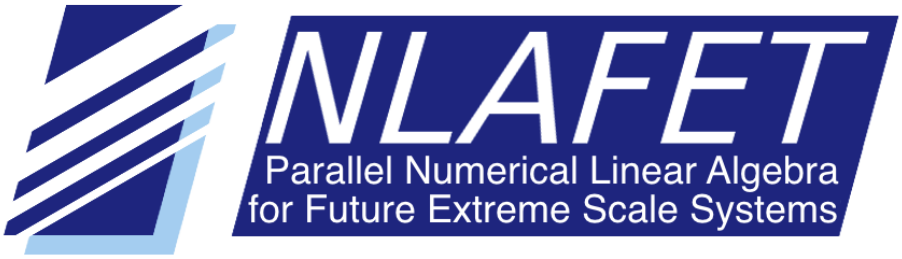


H2020–FETHPC–2014: GA 671633

D3.7

Software for Hybrid Methods

February 2019

Document information

Scheduled delivery	2019-02-28
Actual delivery	2019-02-28
Version	2.0
Responsible partner	RAL

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2018-11-28	Iain Duff	Draft	0.1	Skeleton
2018-12-21	Iain Duff	Draft	0.2	Minor
2019-02-18	Iain Duff and Sébastien Cayrols	Internal review	1.0	Major
2019-02-27	Iain Duff and Sébastien Cayrols	Submitted version	2.0	Minor

Author(s)

Sébastien Cayrols, RAL
Iain Duff, RAL
Florent Lopez, RAL

Internal reviewers

Jan Papez, Inria
Srikara Pranesh, UMAN

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633. We also acknowledge the contribution of our collaborators from CERFACS and ENSEEIHT-IRIT in Toulouse.

Table of Contents

List of Figures

List of Tables

1 Introduction

The *Description of Action* document states for Deliverable 3.7:

"*Software for partitioning saddle point problems and overdetermined systems; improved block Cimmino methods incorporating new solvers from other deliverables. Includes extensive testing, documentation and benchmarking.*"

This deliverable is the final deliverable related to the work performed in WP3, Task 3.4 (Hybrid Direct-Iterative Methods). Our work has focused on the design and implementation of algorithms using the block Cimmino method, as developed in [4], to solve large sparse linear systems of equations $Ax = b$, where A is generally unsymmetric and can even be rectangular. We discussed the design for this software in Deliverable D3.6 submitted at M24 (October 2017). In this deliverable (D3.7), we describe the BC software package, where BC stands for Block Cimmino, that we have developed based on the work discussed in D3.6. Since then we have developed a new interface to the NLAFET SpLDLT code, incorporated a numerically aware partitioning algorithm, and developed a column partitioning for the least-squares solution of overdetermined systems. This work has been conducted in collaboration with our colleagues in CERFACS and ENSEEIHT-IRIT in Toulouse, France. Some of the work of our collaborators in France has been supported by the EU Centre of Excellence for energy applications, called EoCoE. Our BC code is also included in the ABCD (Augmented Block Cimmino Distributed) code distributed by them. The block Cimmino approach couples distributed memory parallelism (using MPI) with the efficient direct solution of sparse subsystems on single multicore nodes. In the preliminary version of the BC code in Deliverable D3.6 and the ABCD code in France, MUMPS [1] is used as the direct solver. Although MUMPS was originally developed for distributed memory computing, it is now able to exploit multicore parallelism through multithreading and the use of some OpenMP directives. Part of our work in NLAFET is to replace MUMPS with the SpLDLT code from the NLAFET S*y*LVER package that is specifically designed to exploit multicore architectures.

1.1 Underdetermined systems

We first discuss the block Cimmino algorithm for solving square or underdetermined systems based on a row partitioning of the matrix. We assume that we are solving the linear system $Ax = b$, where A is a sparse matrix of dimensions $m \times n$, with $m \leq n$ and $\text{rank}(A) = m$.

$$Ax \;=\; b \text{ is partitioned as } \begin{pmatrix} A_1 \\ A_2 \\ . \\ . \\ . \\ A_p \end{pmatrix} x \;=\; \begin{pmatrix} b_1 \\ b_2 \\ . \\ . \\ . \\ b_p \end{pmatrix}$$

and then the algorithm computes a solution iteratively from an initial estimate $x^{(0)}$ according to:

$$u_i \;=\; A_i^{+}\left(b_i - A_i x^{(k)}\right) \quad i = 1, ...p \tag{1.1}$$

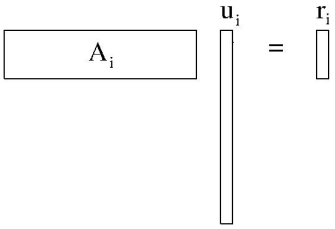
$$x^{(k+1)} \;=\; x^{(k)} + \omega \sum_{i=1}^{p} u_i, \tag{1.2}$$

where we note independence of the set of p equations. Since A is full row rank, so are the A_i and $A_i^+ = A_i^T(A_iA_i^T)^{-1}$. Elfving [7] proves convergence of the method so long as

$$\omega < 2/\rho(H)$$

where $\rho(H)$ is the spectral radius of the iteration matrix that we define in equation (1.3). Equations (1.1) and (1.2) define the block Cimmino method.

Even if the original matrix is square the partitions are underdetermined systems whose shape is of the form:



We obtain the minimum norm solution of the underdetermined systems in equation (1.1) viz.

$$A_iu_i \;=\; r_i, \qquad (r_i = b_i - A_ix^{(k)})$$

by using the augmented system

$$\begin{pmatrix} I & A_i^T \\ A_i & 0 \end{pmatrix}\begin{pmatrix} u_i \\ v_i \end{pmatrix} \;=\; \begin{pmatrix} 0 \\ r_i \end{pmatrix}$$

and will solve these augmented systems using a direct method.

Thus the block Cimmino method is a hybrid method combining direct and iterative methods. Notice that, in common with many hybrid methods, by setting a block size larger than the system dimension, we are just using a direct solver, albeit not very efficiently.

The iteration equations can be written as:

$$\begin{aligned} x^{(k+1)} &= x^{(k)} + \omega\sum_{i=1}^{p} A_i^+\left(b_i - A_ix^{(k)}\right) \\ &= \left(I - \omega\sum_{i=1}^{p} A_i^+A_i\right)x^{(k)} + \omega\sum_{i=1}^{p} A_i^+b_i \\ &= Qx^{(k)} + \omega\sum_{i=1}^{p} A_i^+b_i. \end{aligned}$$

We can write the fixed point iteration as

$$Hx \;=\; \xi, \tag{1.3}$$

where $H = I - Q$. Then, since we assume that the matrices A_i have full row rank, the matrix $H = \omega\sum_{i=1}^{p} A_i^+A_i$ is a sum of projection matrices and is thus positive definite. Therefore the system (1.3) can be solved by the conjugate gradient method. We note that, since $\xi \;=\; \omega\sum_{i=1}^{p} A_i^+b_i$, the parameter ω appears on both sides of equation (1.3) so we can arbitrarily set it to one. However, it has been shown by [11] that we need to accelerate the convergence of CG by using a block conjugate gradient algorithm, usually with only a handful of vectors.

1.2 Overdetermined systems

When the system is overdetermined, that is when there are more rows than columns, we cannot use the row partitioning of the previous section since many of the subsystems would not be of full row rank even if the input matrix is full column rank. In this case, we assume that the matrix A is of full column rank, of dimensions $m \times n$ with $m > n$, and that we want to obtain the vector x to minimize

$$\|b - Ax\|_2.$$

In this case we partition the matrix by columns. That is:

$$Ax \;=\; b \text{ is partitioned as } \begin{pmatrix} A_1 & A_2 & \cdot & \cdot & \cdot & A_p \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_p \end{pmatrix} \;=\; b$$

Elfving [7, equation (2.3)] shows that the subsystems that we then want to solve are of the form:

$$A_i(x_i^{(k+1)} - x_i^{(k)}) = \omega r^{(k)}, \quad i \;=\; 1, ...p, \tag{1.4}$$

where $r^{(k)} = b - Ax^{(k)}$. We see that the set of p equations are again independent so the parallelism is the same as for the row partitioning case. The convergence of the method is again guaranteed [7] if ω is less than 2 over the spectral radius of the iteration matrix shown in equation (1.6).

We now look at the iteration matrix for the solution on the full vector, $x^{(k)}$, to see how we might accelerate the solution in a manner similar to that in Section 1.1. The iteration for the full vector $x^{(k)}$ can be written as

$$\begin{aligned} x^{(k+1)} \;&=\; x^{(k)} \;+\; \omega \begin{pmatrix} A_1^+ \\ A_2^+ \\ \cdot \\ \cdot \\ \cdot \\ A_p^+ \end{pmatrix} (b - Ax^{(k)}) \\ &=\; (I - \omega \begin{pmatrix} A_1^+ \\ A_2^+ \\ \cdot \\ \cdot \\ \cdot \\ A_p^+ \end{pmatrix} A)x^{(k)} + \omega \begin{pmatrix} A_1^+ \\ A_2^+ \\ \cdot \\ \cdot \\ \cdot \\ A_p^+ \end{pmatrix} b. \end{aligned}$$

As before, when we look at the fixed-point iteration, we need to solve the system

$$Hx = \xi, \tag{1.5}$$

with

$$H = \begin{pmatrix} A_1^+ \\ A_2^+ \\ . \\ . \\ . \\ A_p^+ \end{pmatrix} A, \tag{1.6}$$

where, as earlier, the parameter ω appears on both sides of equation (1.5) and can be set to one. Since the A_i are all full column rank

$$A_i^+ = (A_i^T A_i)^{-1} A_i^T,$$

which we note is a different generalized inverse to that in the underdetermined case when $A_i^+ = A_i^T (A_i A_i^T)^{-1}$. This means that

$$H = D^{-1} A^T A, \tag{1.7}$$

where

$$D = diag({A_1}^T A_1, {A_2}^T A_2, ..., {A_p}^T A_p).$$

Thus H is positive definite and is similar to a symmetric matrix through the scaling $D^{1/2} H D^{-1/2}$. We thus, as in the case of the row partitioning of Section 1.1, can solve the system using block conjugate gradients.

To be more precise, we are solving the system $Hx = \xi$ using block CG, where H is defined by equation (1.7) and ξ is obtained by solving the p independent least-square problems $A_i \xi_i = b$ and concatenating the ξ_i to get the vector ξ. Then, for each iteration of block conjugate gradients, we solve the p independent systems by using a sparse direct method on the augmented equations

$$\begin{pmatrix} I & A_i \\ A_i^T & 0 \end{pmatrix} \begin{pmatrix} y \\ z_i \end{pmatrix} = \begin{pmatrix} r^{(k)} \\ 0 \end{pmatrix},$$

where $r^{(k)}$ is the residual at iteration k. From the augmented equations, we note that the matrix-vector product that occurs in CG is now implicitly obtained from the solution of the augmented systems. This allows us to compute at the same time this product as well as the solution z_i, and should lead to better performance.

Our code, whose availability we discuss in Section 2, is included in the ABCD package [14] that was developed originally by a postdoc, Mohamed Zenadi, working at ENSEEIHT-IRIT in Toulouse. ABCD [6] has many other options that have been developed by our colleagues in France but their emphasis has been on further development that augments the system to improve the convergence of the iterative method. In this document we focus on the software we have developed as part of the NLAFET project where we have concentrated more on getting good multi-level parallel performance and offering a new feature of the code, that is the solution of least-squares problems.

The version developed by our collaborators in France that includes the augmentation of the matrix to force greater orthogonality between blocks can be accessed through the repository `https://bitbucket.org/APO_IRIT/ABCD`.

2 Installing the NLAFET BC library

2.1 Overview

The software developed in the NLAFET project by STFC-RAL is available from `https://github.com/NLAFET/` and consists of four repositories: SpLLT, S*y*LVER, ParSHUM, and BC. It is the last one that is described in this deliverable.

For this block Cimmino code (BC) we have decided to maintain a single version of the combined code and so the code is cross-referenced and maintained also at the bitbucket repository in France. Since the NLAFET Library code is concerned with high levels of parallelism and not with the augmentation of the matrix for greater orthogonality between blocks, we differentiate between the full ABCD code and our subset which we call BC. Within our BC code we have two main functionalities defined by the governing parameters that determine whether we are using

1. row partitions for underdetermined or square matrices, or
2. column partitions for overdetermined matrices when a least-squares solution is obtained.

The code chooses the first option if $m \leq n$ and the second if $m > n$.

2.2 Dependencies with external libraries

BC depends on a few external libraries that need to be installed and linked with BC in order to use it:

- BLAS and LAPACK [3]: BLAS is a standard library for performing basic vector and matrix operations. LAPACK is a standard software for numerical linear algebra. Although any library providing BLAS and LAPACK can be used, we recommend MKL [13] (`https://software.intel.com/en-us/mkl`).
- MeTiS [10] is a graph partitioning tool that is used to provide a nested dissection ordering for the sparse direct solver used on the subsystems from the partitioning. Our codes were tested with MeTiS 5.1.0. This partitioning tool can be downloaded from `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`.
- SPRAL is an open access suite of codes from RAL (`https://github.com/SPRAL/`) that is only used directly for matrix scaling although other routines are used by SpLDLT.
- S*y*LVER is the direct solver package that includes the SpLDLT routine with dependencies
  - SPRAL
  - MeTiS
  - StarPU is a runtime system developed in Bordeaux that is used to exploit the parallel architecture. It is available in Open Source form from `http://starpu.gforge.inria.fr/`.

2.3 Installation

We now describe the complete installation of BC.

2.3.1 Install the dependencies

The installation of the first two packages in Section 2.2 is well described in their respective documentation therefore we discuss here the installation of SPRAL, StarPU, and S*y*LVER. The compilation process for these installations is handled by the GNU Autotools package[1].

Installing SPRAL The latest development version of the SPRAL library can be found on the GitHub repository `https://github.com/ralna/spral`. Instructions for compiling SPRAL are shown in the listing in Figure 1.

```
# Get latest development version from github and run dev scripts
git clone https://github.com/ralna/spral.git spral
cd spral
./autogen.sh

# Build and install library
mkdir build
cd build
../configure --with-metis='-L/path/to/metis -lmetis'
make
sudo make install # Optional
```

Figure 1: Compiling the SPRAL library.

Installing StarPU The latest development version of the StarPU runtime system can be downloaded via git. Instructions for downloading and compiling StarPU are shown in the listing in Figure 2.

```
# Get latest development version via git
git clone https://scm.gforge.inria.fr/anonscm/git/starpu/starpu.git starpu
cd starpu
./autogen.sh

# Build and install library
mkdir build
cd build
../configure
make
sudo make install # Optional
```

Figure 2: Compiling the StarPU library.

Installing S*y*LVER The S*y*LVER package is held in the GitHub repository `https://github.com/NLAFET/Sylver`. The steps to download and install the package are shown in Figure 3.

[1] `https://www.lrde.epita.fr/~adl/autotools.html`

```
git clone git@github.com:NLAFET/sylver.git sylver
cd sylver
mkdir build
cd build
cmake .. -DRUNTIME=StarPU # Use the StarPU runtime system
make # This will create the library
```

Figure 3: Compiling the S*ylver* package.

2.3.2 Get and install BC

1. Get the latest version of BC.
   `$ git clone https://github.com/NLAFET/block_cimmino.git BC`

2. In the BC directory that has just been created, make a copy named `abcdCmake.in` of one of the abcdCmake.in.* files located in the `cmake.in` directory, and edit it in order to set the compiler directives, the library paths, and flags.

   In order to use SpLDLT as the inner solver with StarPU, do
   `$ cp cmake.in/abcdCmake.in.spldlt_starpu abcdCmake.in`

3. Create a build directory in the BC directory, and move into it
   `$ mkdir build && cd build`

4. To compile BC and create the library `lib/libBC.a` and the examples, type
   `$ cmake .. && make`

5. Now, the subroutines from the library BC can be called by a program by including their corresponding `header` file. The directory `example` provides a few stand-alone programs to test the library.

3 Using the BC code

There are a large number of parameters in the ABCD code. Mostly the default values are suitable for our BC code and many are not applicable to our code. We describe the input data format in Section 3.1 and the most relevant parameters in Section 3.2.

3.1 Input data format

On entry, the sparse matrix should be in coordinate form. That is, it is represented by three arrays of length the number of entries in the matrix. There are two integer arrays `irn` and `jcn`, and a real array `val`. The entry `val(i)` is in row `irn(i)` and column `jcn(i)` for $i = 1, \ldots nz$ where nz is the number of entries in the sparse matrix. Of course the dimensions of the matrix m, the number of rows and n, the number of columns and nz must also be set on entry. We present in Listing 1 the relevant components of the internal structure of the BC solver.

```
   int m;          /*! The number of rows in the matrix */
   int n;          /*!  The number of columns in the matrix */
   int nz;         /*! The number of entries in the matrix */
   bool sym;       /*! The symmetry of the matrix*/
```

```
5    int *irn;       /*! The row indices of size #nz */
6    int *jcn;       /*! The column indices of size #nz */
7    double *val;    /*! The entries of the matrix of size #nz */
8    int nrhs;       /*! The number of right-hand sides to solve, default is 1 */
9    double *rhs;    /*! The right-hand side of size #m * #nrhs */
10   double *sol;    /*!  The solution vector of size #n * #nrhs */
11   int start_index; /*! Defines whether it's Fortran-Style (1) or C-Style */
12   std::vector<double> rhoVector;        /* the residual vector */
13   std::vector<double> scaledResidualVector; /* the scaled residual vector */
14   std::vector<int> icntl;               /*! The integer control array */
15   std::vector<double> dcntl;            /*! The real control array */
16   std::vector<int> info;                /*! The integer info output array */
17   std::vector<double> dinfo;            /*! The real info output array */
18   std::vector<int> man_scaling;         /*! The real scaling Number of iterations */
19   mpi::communicator comm; /*! The global communicator */
```

Listing 1: Presentation of the main components of the internal structure of BC.

The internal routines in BC require matrices to be stored in a compressed sparse row format (CSR). The conversion is done automatically by the code.

3.2 Parameters

In this section, we present some parameters that can be set by the user. The integer parameters are in the `icntl` array in Listing 1.

- `nbparts` (default 2) defines the number of partitions of the matrix
- `part_type` is the identifier of the partitioning technique used. This defaults to the numerically aware partitioning of Torun [12].
- `scaling` (default 1) determines whether scaling is used (1) or not (0).
- `itmax` (default 1000) sets up the maximum number of iterations in block-CG
- `block_size` (default 1) corresponds to the number of rhs used in block-CG
- `verbose_level` defines the verbosity of the solver
- `innerSolver_ncpu` sets the number of CPUs for the inner solves on the augmented systems.
- `innerSolver_ngpu` sets the number of GPUs for the inner solves on the augmented systems.
- `innerSolver_nemin` sets the tree amalgamation level for the inner solves on the augmented systems.
  - The matrix is partitioned by rows for solving square or underdetermined systems (value 1).
  - The matrix is partitioned by columns for overdetermined systems in the context of solving least-squares problems (value 0).

The double precision parameter is in the `dcntl` array which consists of:

- `threshold` is the stopping criterion in block-CG

3.3 Example program

The following listings illustrate how BC can be used in an example program for solving a sparse linear system.

The steps needed to solve a sparse system are:

- `BC_INITIALISE` converts the matrix given by the user in coordinate format into a CSR format, and if the rhs is not given by the user, the routine creates a rhs b such that $b = A \times x_f$, where x_f is a vector of dimension n whose entries are $x_f(i) = 1.0$.
- `BC_PREPROCESS` scales the matrix, calls the partitioner, and creates augmented systems.
- `BC_FACTOR` distributes the data (that computes the matrix inf norm ,*i.e.*, max over local inf norm), initializes the inner solver, and factorises the augmented systems.
- `BC_SOLVE` synchronizes masters, distributes the rhs, and calls bcg.

To compile these codes:

- For the C++ example, do
  `make bc_example`
- and for the C example, do
  `make bc_example_c`

```
#include "abcd.h"

int main(int argc, char* argv[])
{
  int err = 0;
  double t = 0.0;

  mpi::environment env(argc, argv);
  mpi::communicator world;
  string matrix_file, rhs_file, config_file;

  abcd solver;

  if(world.rank() == 0) {

    config_file = "config_file.info";
    if(argc >= 2) config_file = argv[1];

    std::cout << "Load the config_file " << config_file << std::endl;
    err = solver.parse_configFile(config_file, matrix_file, rhs_file);

    std::cout << "Load the matrix " << matrix_file << std::endl;
    err = solver.load_MM(matrix_file);

    if(!rhs_file.empty()){
      std::cout << "Load the RHS from " << rhs_file << std::endl;
      err = solver.load_RHS(rhs_file);
    }
  }

```

```
  try {
    t = MPI_Wtime();
    solver(BC_INITIALISE);
    solver(BC_PREPROCESS);
    solver(BC_FACTOR);
    solver(BC_SOLVE);
    if(world.rank() == 0) clog << "Total time: " << MPI_Wtime() - t << endl;
  } catch(std::runtime_error e) {
    std::cout << world.rank() << " Error code : " << e.what() << std::endl;
    err = 1;
  }
  world.barrier();

  return err;
}
```

Listing 2: Example of a C++ code calling BC.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <string.h>
#include "abcd_c.h"

int main(int argc, char **argv)
{
  int err = 0;
  int rank = 0;
  int size = 1;
  double t = 0.0;
  char matrix_file[50] = "../example/e05r0500.mtx";
  char rhs_file[50]  = "../example/e05r0500_rhs.mtx";
  abcd_c *solver;

  /* Initialize MPI */
  MPI_Init(&argc, &argv);

  /* Find out my identity in the default communicator */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  solver = new_solver();

  /* Setup the solver */
  solver.icntl[abcd_nbparts] = size;
  solver.icntl[abcd_part_type] = 5;
  solver.icntl[abcd_scaling] = 2;

  /* Setup bcg parameters */
  solver.icntl[abcd_itmax]   = 1000;
  solver.dcntl[abcd_threshold] = 1e-8;
  solver.dcntl[abcd_block_size] = 1;

  /* Setup inner solver */
  solver.icntl[abcd_innersolver_ncpu] = 1;
  solver.icntl[abcd_innersolver_ngpu] = 1;
  solver.icntl[abcd_innersolver_nemin] = 1;

```

```
41  if(rank == 0) { // the master
42
43    printf("Load the matrix %s\n", matrix_file);
44    err = load_MM(solver, matrix_file);
45
46    printf("Load the RHS from %s\n", rhs_file);
47    err = load_RHS(solver, rhs_file);
48  }
49
50  t = MPI_Wtime();
51  call_solver(solver, BC_INITIALISE);
52  call_solver(solver, BC_PREPROCESS);
53  call_solver(solver, BC_FACTOR);
54  call_solver(solver, BC_SOLVE);
55  if(rank == 0) printf("Total time: %f\n", MPI_Wtime() - t);
56
57  MPI_Finalize();
58
59  return 0;
60 }
```

Listing 3: Example of a C code calling BC.

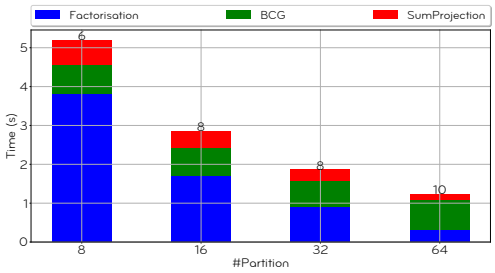
4 Numerical experiments

For the row partitioned algorithm, we have performed some runs on Kebnekaise at the High Performance Computing Center North (HPC2N). This machine has 432 nodes, where each node consists of Intel Xeon E5-2690v4 (2x14 cores) and 128 GB of memory. The characteristics of the test matrices are shown in Table 1. These matrices are from the SuiteSparse collection of sparse matrices [5].

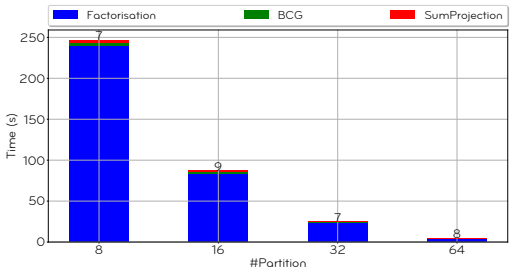
Name	n	entries	Description
`torso3`	2.59e+05	4.43e+06	Electro Physics
`cage13`	4.45e+05	7.48e+06	DNA electrophoresis
`hamrle3`	1.45e+06	5.51e+06	Circuit simulation

Table 1: Statistics for matrices used in row partitioned tests

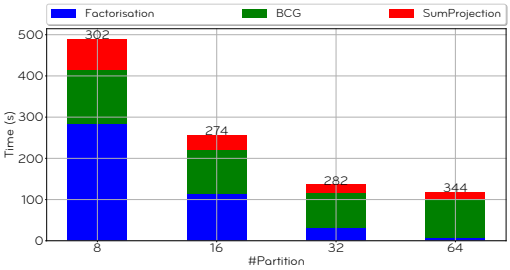
We perform a strong scaling on the number of partitions, and we show the time to solve each problem in Figure 4. To emphasize the behaviour of our method, we choose for block CG, a block size of one, and a threshold of 10^{-8}. We first notice that, as predicted, increasing the number of partitions reduces the time to solve the system. More precisely, the time to factorize the augmented systems decreases as the size of each partition decreases. We also note that for `torso3` and `cage13`, the number of iterations of CG does not increase drastically with the number of partitions. In the case of `cage13`, this number stays roughly the same. On the other hand, the number of iterations for the `Hamrle3` matrix decreases from 302 to 274 when the number of partitions is doubled from 8, and reaches 344 for 64 partitions. This means that increasing the number of partitions does not really impact the number of iterations while the global runtime benefits from it.



(a) Torso3



(b) Cage13



(c) Hamrle3

Figure 4: Convergence of the method for different matrices when the number of partitions increases from 8 to 64. The numbers on the top of each bar correspond to the number of CG iterations, using a threshold of 10^{-8}.

For our tests on the solution of least-squares problems we use some standard test problems and show data on these in Table 2. The first four matrices are in the SuiteSparse collection of sparse matrices [5]. The `PIGS` matrices are from pig breeding that were obtained from Markus Hegland [8, 9] for experiments on a QR code at CERFACS [2] and will shortly be available in SuiteSparse.

Name	m	n	entries	$\kappa(A)$	Description
`well1850`	1850	712	8755	1.1e+02	Paige and Saunders test
`Kemelmacher`	28452	9693	100875	2.5e+04	Computer graphics/vision problem
`mesh_deform`	234023	9393	853829	1.7e+03	Image mesh deformation
`deltaX`	68600	21961	247424	1.2e+16	High fill-in example
`PIGS_small1`	3140	1988	8510	3.9e+05	Pig breeding
`PIGS_small2`	6280	3976	25530	5.2e+05	Pig breeding
`PIGS_medium1`	9397	6119	25013	4.3e+05	Pig breeding
`PIGS_medium2`	18794	12238	75039	4.2e+05	Pig breeding
`PIGS_large1`	28254	17264	75018	4.6e+05	Pig breeding
`PIGS_large2`	56508	34258	225054	4.7e+05	Pig breeding

Table 2: Statistics for matrices used in least-squares tests.

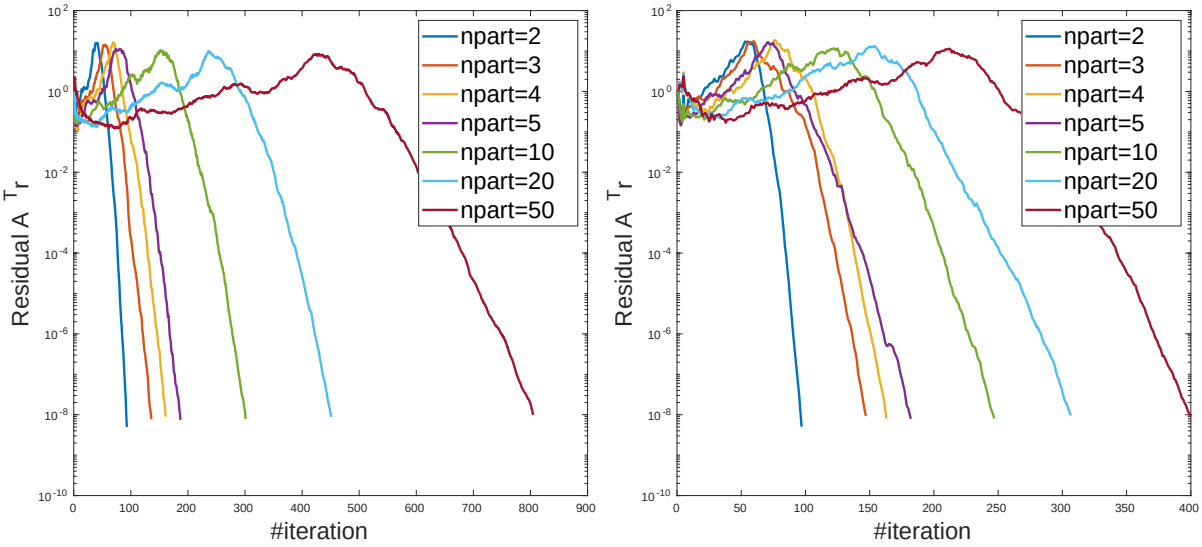
Clearly a greater number of partitions will give us more parallelism but will result in more iterations. Also, we would hope that the numerically aware partitioning [12] will do better than simply dividing the matrix uniformally into blocks of columns. We show these effects quite strongly in the results in Table 3 for runs on the first four matrices.

The stopping criterion for our algorithm is based on the value of $\|A^T r\|_2$. We also show the number of iterations for solving the system using a conjugate gradient iteration on the normal equations. Significantly less iterations are needed by our block Cimmino code than by CGNR even when using many partitions. In nearly all cases, the numerically aware partitioning results in far fewer iterations than the natural partitioning sometimes by a factor of over 5.

		Natural partitioning		Numerically aware	
Matrix name	npart	niter	$A^T r$	niter	$A^T r$
well1850	CGNR	458	9.85e-09		
	2	265	8.45e-09	48	1.27e-14
	3	428	8.51e-09	65	4.95e-10
	4	441	7.46e-09	77	6.49e-09
	5	451	9.68e-09	96	1.87e-09
	10	457	8.73e-09	115	8.26e-09
	20	456	9.76e-09	186	6.83e-09
	50	458	9.16e-09	305	9.71e-09
Kemelmacher	CGNR	3591	9.89e-09		
	2	93	4.99e-09	97	5.05e-09
	3	136	7.75e-09	147	9.36e-09
	4	161	8.96e-09	163	8.12e-09
	5	187	7.47e-09	182	7.92e-09
	10	301	7.83e-09	247	8.09e-09
	20	451	8.83e-09	307	9.76e-09
	50	805	9.92e-09	400	9.17e-09
mesh_deform	2	269	9.52e-09	45	7.50e-09
	3	306	9.89e-09	67	9.61e-09
	4	390	9.11e-09	63	7.61e-09
	5	398	9.45e-09	70	7.23e-09
	10	440	8.94e-09	67	7.95e-09
	20	474	9.15e-09	82	7.33e-09
	50	479	9.41e-09	76	8.56e-09
deltaX	CGNR	2344	8.47e-09		
	2	296	7.51e-09	269	9.99e-09
	3	421	9.49e-09	489	8.47e-09
	4	521	8.63e-09	594	9.19e-09
	5	636	9.74e-09	595	9.77e-09
	10	1001	9.72e-09	724	8.40e-09
	20	1418	8.75e-09	1549	7.92e-09
	50	2056	9.33e-09	2313	9.52e-09

Table 3: Convergence of the method on SuiteSparse matrices for a threshold of $1e-8$.

We show convergence curves for the matrix Kemelmacher in Figure 5. These show that the numerically aware partitioning is in general much better than the natural partitioning in terms of number of iterations for convergence. Although our measure for convergence is not at all monotonic, it does eventually convergence at a fast rate.



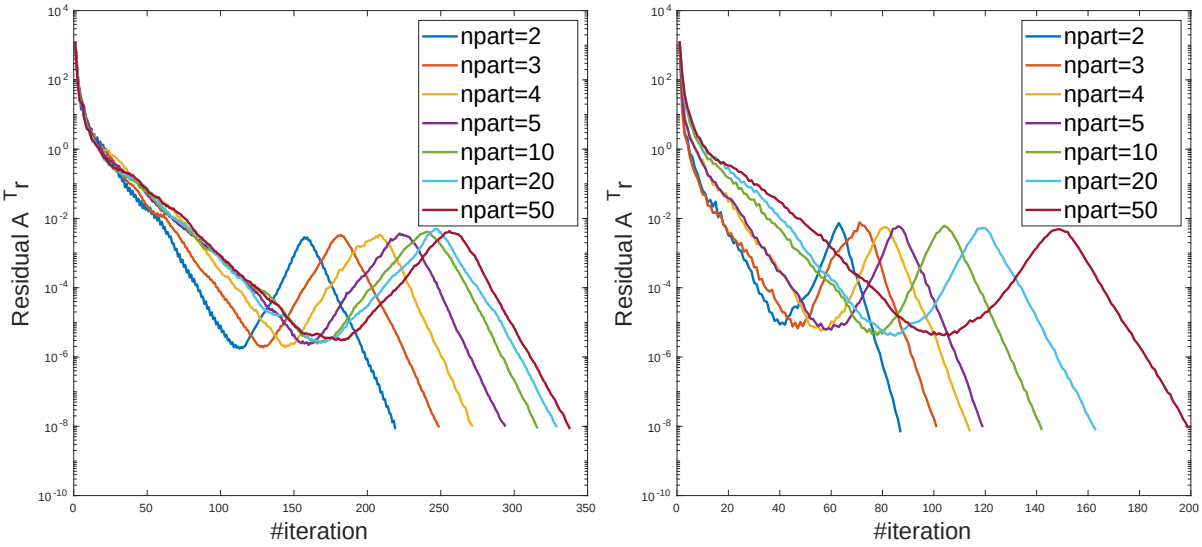
(a) Natural partitioning **(b)** Numerically aware partitioning

Figure 5: Convergence of the two partitioning schemes on Kemelmacher for differing numbers of partitions.

The results on the PIGS matrices in Table 4 show a similar pattern to the earlier runs. In every instance the numerically aware partitioning does better and for the largest problem it beats CGNR by a factor of more than eight even when using 50 partitions. Finally we show convergence curves for the matrix PIGS_medium1 in Figure 6 that again are similar to our results in Figure 5.

Matrix name	npart	Natural partitioning niter		niter	Numerically aware $A^T r$	
small1	CGNR	562	9.62e-09			
	2	202	9.39e-09		83	7.08e-09
	3	239	8.47e-09		104	7.14e-09
	4	250	9.35e-09		120	9.68e-09
	5	260	8.45e-09		130	6.81e-09
	10	278	9.39e-09		159	8.35e-09
	20	286	7.21e-09		173	7.49e-09
	50	289	9.24e-09		202	9.38e-09
small2	CGNR	705	9.96e-09			
	2	183	9.36e-09		64	9.63e-09
	3	217	9.23e-09		78	9.99e-09
	4	232	9.70e-09		103	7.20e-09
	5	236	9.16e-09		108	8.32e-09
	10	263	8.72e-09		129	8.50e-09
	20	273	9.09e-09		148	9.76e-09
	50	278	8.92e-09		176	8.83e-09
medium1	CGNR	770	9.50e-09			
	2	219	8.25e-09		87	6.70e-09
	3	249	9.43e-09		101	9.45e-09
	4	272	9.86e-09		114	7.14e-09
	5	294	9.82e-09		119	9.47e-09
	10	316	8.68e-09		142	7.77e-09
	20	329	9.36e-09		163	7.69e-09
	50	338	8.30e-09		199	8.95e-09
medium2	CGNR	1475	8.28e-09			
	2	191	9.56e-09		104	6.67e-09
	3	218	9.98e-09		120	7.94e-09
	4	365	8.60e-09		134	7.94e-09
	5	399	9.37e-09		141	8.78e-09
	10	433	8.73e-09		179	7.13e-09
	20	448	9.02e-09		213	8.87e-09
	50	455	9.05e-09		252	8.06e-09
large1	CGNR	1208	8.89e-09			
	2	235	7.85e-09		84	6.62e-09
	3	277	9.17e-09		104	7.88e-09
	4	290	7.83e-09		115	9.96e-09
	5	312	9.01e-09		122	7.59e-09
	10	344	9.57e-09		143	8.13e-09
	20	358	9.17e-09		175	8.73e-09
	50	362	9.84e-09		213	8.57e-09
large2	CGNR	2225	9.64e-09			
	2	325	8.72e-09		92	9.89e-09
	3	383	8.78e-09		111	6.83e-09
	4	397	9.73e-09		127	8.47e-09
	5	436	9.05e-09		144	8.79e-09
	10	477	8.67e-09		177	9.08e-09
	20	490	9.16e-09		221	8.85e-09
	50	492	9.72e-09		274	8.84e-09

Table 4: Convergence of the method on the PIGS problems for a threshold of $1e-8$.



(a) Natural partitioning **(b)** Numerically aware partitioning

Figure 6: Convergence of the two partitioning schemes on PIGS_medium1 for differing numbers of partitions.

5 Acknowledgements

This project is funded from the European Union's Horizon 2020 research and innovation program under the NLAFET grant agreement No 671633. We also acknowledge the contribution of our collaborators, Philippe Leleux, Daniel Ruiz, and Sukru Torun from CERFACS and ENSEEIHT-IRIT in Toulouse.

References

[1] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Analysis and Applications*, 23(1):15–41, 2001.

[2] Patrick R. Amestoy, Iain S. Duff, and Chiara Puglisi. Multifrontal QR factorization in a multiprocessor environment. *Numerical Linear Algebra with Applications*, 3(4):275–300, 1996.

[3] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*, volume 9. SIAM, 1999.

[4] Mario Arioli, Iain S. Duff, Joseph Noailles, and Daniel Ruiz. A block projection method for sparse matrices. *SIAM J. Scientific and Statistical Computing*, 13:47–70, 1992.

[5] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1–25, 2011.

[6] Iain S. Duff, Ronan Guivarch, Daniel Ruiz, and Mohamed Zenadi. The augmented block Cimmino distributed method. *SIAM J. Scientific Computing*, 37(3):A1248–A1269, 2015.

[7] T. Elfving. Block-iterative methods for consistent and inconsistent linear equations. *Numerische Mathematik*, 35(1):1–12, 1980.

[8] M. Hegland. On the computation of breeding values. In H. Burkhart, editor, *Proceedings of CONPAR 90-VAPP IV Joint International Conference on Vector and Parallel Processing*, volume 457 of *Lecture Notes in Comput. Sci*, pages 232–242, Berlin, 1990. Springer-Verlag.

[9] M. Hegland. Description and use of animal breeding data for large least squares problems. Technical Report TR/PA/93/50, CERFACS, Toulouse, France, 1993.

[10] George Karypis and Vipin Kumar. METIS –unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[11] Daniel F. Ruiz. *Solution of large sparse unsymmetric linear systems with a block iterative method in a multiprocessor environment.* PhD Thesis, Institut National Polytechnique de Toulouse, 1992. CERFACS Technical Report, TH/PA/92/06.

[12] F. S. Torun, M. Manguoglu, and C. Aykanat. A novel partitioning method for accelerating the block Cimmino algorithm. *SIAM J. Scientific Computing*, 40(6):C827–C850, 2018.

[13] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.

[14] Mohamed Zenadi. *The solution of large sparse linear systems on parallel computers using a hybrid implementation of the block Cimmino method.* Thése de Doctorat, Institut National Polytechnique de Toulouse, Toulouse, France, décembre 2013.