GaussPar

Parallel gaussian algorithm for finite fields

1.0.0

5 May 2021

Jendrik Brachter

Sergio Siccha

Emma Ahrens

Jendrik Brachter
Email: brachter@cs.uni-kl.de

Sergio Siccha
Email: siccha@mathematik.uni-kl.de

Emma Ahrens
Email: emma.ahrens@rwth-aachen.de

Contents

Chapter 1

The GaussPar package

1.1 Introduction

This package implements the algorithm described in "A parallel algorithm for Gaussian elimination over finite fields" (Linton, S., Nebe, G., Niemeyer, A., Parker, R. and Thackray, J. (2018)). The algorithm divides a given matrix into smaller submatrices (blocks) and carries out steps of the Gaussian elimination block by block. The paper identifies the minimal dependencies between processing different blocks. Independent steps can be performed simultaneously, allowing for high levels of parallelism. We provide two main functions `EchelonMatBlockwise` (2.1.2) and `EchelonMatTransformationBlockwise` (2.1.1), where the latter additionally computes the transformation matrix in parallel. The naming conventions are chosen in accordance with the "Gauss"-package [1] , which provides an implementation of the sequential GAUSS algorithm. Our functions do currently not support sparse matrices.

[1] https://www.gap-system.org/Packages/gauss.html

Chapter 2

Gaussian Elimination

2.1 Gaussian Elimination

This section describes the different variants of our implementation of the Gaussian algorithm.

Note that the value of the option `numberBlocks` described below has a big impact on the performance of our algorithms. For information on how to choose a suitable value for `numberBlocks` see Chapter 3.

2.1.1 EchelonMatTransformationBlockwise

▷ `EchelonMatTransformationBlockwise(mat[, options])` (function)

Returns: a record that contains information on the echelon form of *mat* and the corresponding transformation matrix.

This is the main function of the GaussPar package. It computes the reduced row echelon form (RREF) of the matrix *mat* and the corresponding transformation matrix. In a pre-processing step, *mat* is split up into a block matrix whose blocks can be processed in parallel.

The input parameters have the following meaning:

- *mat* is a matrix defined over a finite field
- *options* is a record that can be used to provide some additional parameters. Note that a specification of either *numberBlocks* or both of *numberBlocksHeight* and *numberBlocksWidth* is mandatory. The following parameters are currently supported:
  - *numberBlocksHeight* and *numberBlocksWidth*: The number of vertical and horizontal blocks in which to divide the matrix during the algorithm.
  - *numberBlocks*: Use this argument if you want the same number of vertical and horizontal blocks in the block matrix decomposition of *mat*.
  - *verify*: If set to *true*, the computation is verified at the end. That is, we check wheter *coeffs* * *mat* is in RREF. This option is only available for the function *EchelonMatTransformationBlockwise*.
  - *isChopped*: It is possible to input *mat* directly as a matrix of block matrices. In this case the parameter *isChopped* must be set to *true* and the splitting step is skipped.

The output record contains the following items:

- *vectors*: a matrix that forms the RREF of *mat* without zero rows
- *heads*: a list of integers, such that *heads[i]* gives the number of the row for which the pivot element is in column i. If no such row exists, *heads[i]* is *0*.
- *coeffs*: the corresponding transformation matrix. It holds *coeffs* * *mat* = *vectors*.
- *relations*: a matrix whose rows form a basis for the row null space of *mat*. If *relations* is not the empty list, it holds that *relations* * *mat* = *0*. Otherwise *mat* has full row rank.

Example

```
gap> A := RandomMat(8, 5, GF(5)) * RandomMat(5, 8, GF(5));;
gap> Display(A);
1 4 3 2 4 4 3 4
4 1 2 4 2 . . 4
2 3 1 4 3 3 1 3
3 . 4 3 3 2 4 .
4 1 3 2 3 3 . 2
2 1 3 3 1 1 2 3
. 3 3 . 1 1 3 .
4 1 4 1 4 3 1 1
gap> res := EchelonMatTransformationBlockwise(A, rec(numberBlocks := 2));;
gap> Display(res.vectors);
1 . . . . . . 3
. 1 . . . 3 2 3
. . 1 . . . . 3
. . . 1 . 2 1 1
. . . . 1 2 2 2
gap> res.coeffs * A=res.vectors;
true
```

The transformation matrix can be easily obtained from the output record as follows:

Example

```
gap> trafo := Concatenation(res.coeffs, res.relations);;
gap> Display(trafo * A);
1 . . . . . . 3
. 1 . . . 3 2 3
. . 1 . . . . 3
. . . 1 . 2 1 1
. . . . 1 2 2 2
. . . . . . . .
. . . . . . . .
. . . . . . . .
```

It is also possible to directly input a block matrix.

Example

```
gap> A := RandomMat(8, 5, GF(5)) * RandomMat(5, 8, GF(5));;
gap> res1 := EchelonMatTransformationBlockwise(A, rec(numberBlocks := 2));;
gap> A1 := A{[1..4]}{[1..4]};;
gap> A2 := A{[1..4]}{[5..8]};;
gap> A3 := A{[5..8]}{[1..2]};;
gap> A4 := A{[5..8]}{[5..8]};;
gap> A_blockwise := [[A1,A2],[A3,A4]];;
```

```
gap> res3 := EchelonMatTransformationBlockwise(A_blockwise,
  rec(numberBlocks := 2, isChopped := true));;
gap> res3 = res1;
true
```

2.1.2 EchelonMatBlockwise

▷ `EchelonMatBlockwise(`*`mat[, options]`*`)` (function)

Returns: a record that contains information on the echelon form of *mat*.

This is a version of the main function that computes the reduced row echelon form (RREF) of the matrix *mat* but doesn't compute the corresponding transformation matrix. In a pre-processing step, *mat* is split up into a block matrix whose blocks can be processed in parallel.

The input parameters have the following meaning:

- *mat* is a matrix defined over a finite field
- *options* is a record that is used to provide additional parameters. For more information see *EchelonMatTransformationBlockwise* (See `EchelonMatTransformationBlockwise` (2.1.1)).

The output record contains the items *vectors* and *heads*. For their meaning, see `EchelonMatTransformationBlockwise` (2.1.1).

Chapter 3

Finding a suitable number of blocks

Experiments with matrices over fields of sizes 2 to 11 and dimensions 500 to 10.000 have found values for *numberBlocks* from 6 to 15 to be acceptable. Note though, that this highly depends on the calculation, the number of used threads and the machine itself.

3.1 Measure Contention

3.1.1 GAUSS_MeasureContention

▷ GAUSS_MeasureContention(*numberBlocks, q, A[, showOutput]*) (function)

This function helps with finding a suitable value for *numberBlocks*, which has a big influence on the performance of the EchelonMatTransformationBlockwise (2.1.1) function. For inputs *numberBlocks*, a field size *q*, and a matrix *A* over GF(q), this function does the following:

- it calls the parallel function EchelonMatTransformationBlockwise (2.1.1),
- it calls the sequential function EchelonMatTransformation from the package Gauss,
- it prints the wall time, that is the elapsed real time, each call took,
- for the parallel function, it prints an estimate of the lock contenation ratio, a short explanation of lock contention is given below. A high lock contention ratio deteriorates the performance of the algorithm.

The input *showOutput* can be used to suppress printing of messages.
The influence of *numberBlocks* on the performance is as follows:

- if *numberBlocks* is too small, then not enough calculations can happen in parallel
- if *numberBlocks* is too big then:
  - the lock contention ratio increases
  - HPC-GAP's task framework generates a big overhead since it is not optimized.

In a parallel computation several threads may try to access an object at the same time. HPC-GAP needs to prevent situations in which one thread writes into an object that another thread is currently

reading, since that could lead to corrupted data. To this end, HPC-GAP can synchronize access to objects via locks.

If a thread reads an object, that thread can acquire a lock for that object, that is no other thread can write into it until the lock is released. If a thread tries to write into said object before the lock is released we say that the lock was contended. In such a situation the contending thread needs to wait until the lock is released. This leads to waiting times or unnecessary context-switches and deteriorates performance if it happens too often.

Example

```
gap> n := 4000;; numberBlocks := 8;; q := 5;;
gap> A := RandomMat(n, n, GF(q));;
gap> GAUSS_MeasureContention(numberBlocks, q, A);
Make sure you called GAP with sufficient preallocated memory via ‘-m‘ if you
try non-trivial examples! Otherwise garbage collection will be a big
overhead.

Starting the parallel algorithm
EchelonMatTransformationBlockwise.

Wall time  parallel execution (ms): 33940.
CPU  time  parallel execution (ms): 2
Lock statistics(estimates):
acquired - 181502, contended - 1120, ratio - 1.%
Locks acquired and contention counters per thread
[ thread, locks acquired, locks contended ]:
[ 5, 54093, 248 ]
[ 6, 51004, 228 ]
[ 7, 37102, 389 ]
[ 8, 39303, 255 ]

Starting the sequential algorithm
EchelonMatTransformation

Wall time Gauss pkg execution (ms): 66778.

Speedup factor (sequential / parallel wall time):
1.96
```

Chapter 4

Using the task framework provided by HPC-GAP

To implement our parallel version of the Gauss algorithm we use the task framework provided by HPC-GAP. The structure of the source files reflects this by grouping our functions depending on how they make use of HPC-GAP's shared memory model.

4.1 The Package's Structure

- `main.gi`: Contains the main function `DoEchelonMatTransformationBlockwise`, which is wrapped by `EchelonMatBlockwise` (2.1.2) and `EchelonMatTransformationBlockwise` (2.1.1). It is the function which schedules all tasks.

- `dependencies.g`: The functions in this file compute the dependencies of the algorithm's sub-programs between each other.

- `tasks.g`: The functions in this file are scheduled as tasks by the main routine. They need to make sure that they only write read-only objects into the "shared" atomic lists.

- `thread-local.g`: The functions in this file are called by functions from "tasks.g". In principle, these functions only work in a single thread-local region and don't need to know anything about other threads. These functions may only access read-only objects or objects from the executing thread's thread-local region. They may only emit or write into thread-local objects.

Chapter 5

Utility Functions

5.1 Utility Functions

The following functions were created while implementing the high-level functions, but we found that they could be useful outside of our functions, too. You can find the specifications of the functions and some small examples here.

5.1.1 IsMatrixInRREF

▷ IsMatrixInRREF(*mat*) (function)

Returns: bool

Checks whether the matrix *mat* is in RREF

Example

```
gap> M := RandomMat( 3, 3 );;
gap> Display( M );
[ [   1,   0,  -1 ],
  [  -1,  -1,   1 ],
  [  -1,   1,  -2 ] ]
gap> IsMatrixInRREF( M );
false
gap> L := [ [ 1, 0, 3 ], [ 0, 1, 5 ] ];;
gap> Display( L );
[ [  1,  0,  3 ],
  [  0,  1,  5 ] ]
gap> IsMatrixInRREF( L );
true
```

5.1.2 RandomEchelonMat

▷ RandomEchelonMat(*height, width, rank, randomSeed, field*) (function)

Returns: mat

Constructs a random matrix in RREF of the given height (number of rows), width (number of columns) and rank over the given field. Using the same random seed will recreate the same matrix.

Example

```
gap> # We use Mersenne twister as a random seed here.
gap> randomSeed := RandomSource(IsMersenneTwister, 42);;
```

```
gap> M := RandomEchelonMat(10, 10, 5, randomSeed, GF(5));;
gap> Display(M);
1 . . . . . 1 4 4 .
. 1 . . . 1 3 2 4 3
. . 1 . . . 3 . 1 .
. . . 1 . 2 2 3 4 2
. . . . 1 2 2 4 1 4
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

Index