FFPACK: Finite Field Linear Algebra Package

Jean-Guillaume Dumas, Pascal Giorgi and Clément Pernet

pascal.giorgi@ens-lyon.fr, {Jean.Guillaume.Dumas, Clément.Pernet}@imag.fr

Introduction

Motivation: Integer linear algebra

Introduction

Motivation: Integer linear algebra

- Sparse or structured matrix: specific methods (Blackbox,...)

  ⇒Otherwise: Dense methods

Introduction

Motivation: Integer linear algebra

- Sparse or structured matrix: specific methods (Blackbox,...)

  ⇒Otherwise: Dense methods

- Limit the growth of intermediate results:

  ⇒Several computations over distinct finite fields and reconstruction using Chinese Remaindering

Introduction

Motivation: Integer linear algebra

- Sparse or structured matrix: specific methods (Blackbox,...)

  ⇒Otherwise: Dense methods

- Limit the growth of intermediate results:

  ⇒Several computations over distinct finite fields and reconstruction using Chinese Remaindering

Applications: integer polynomial factorization, Gröbner basis computation, integer system solving, . . .

Exact Dense Linear Algebra Routines

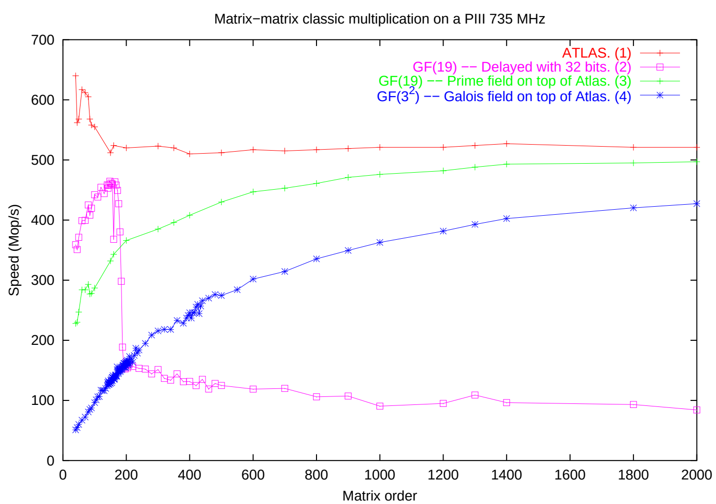
FFLAS Finite Field Linear Algebra Subroutines

- Based on a Matrix Multiplication kernel
- Using numerical BLAS through conversions
- Fast Matrix Multiplication algorithm

Exact Dense Linear Algebra Routines

FFLAS Finite Field Linear Algebra Subroutines

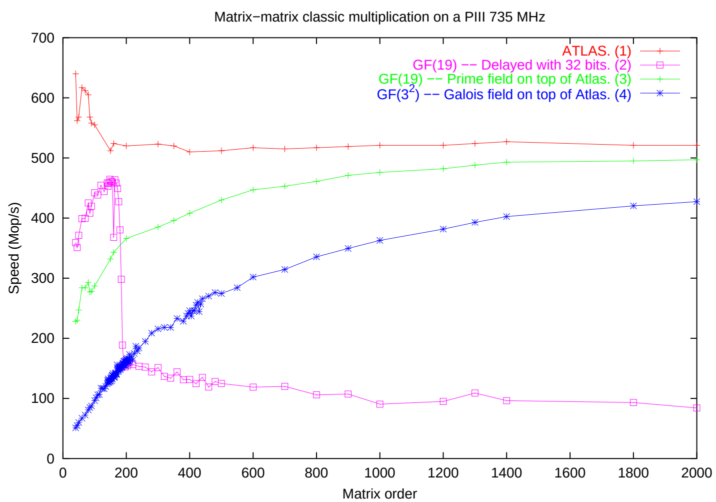
- Based on a Matrix Multiplication kernel
- Using numerical BLAS through conversions
- Fast Matrix Multiplication algorithm



Exact Dense Linear Algebra Routines

FFLAS Finite Field Linear Algebra Subroutines

- Based on a Matrix Multiplication kernel
- Using numerical BLAS through conversions
- Fast Matrix Multiplication algorithm



FFPACK Finite Field Linear Algebra Package

- Higher Level (cf LAPACK)
- Based on matrix triangularization

Contents

Base field representation

- `Modular<double>`:
  - Based on machine `double` floating point representation
  - Only using the mantissa
    $\Rightarrow$Exact representation of integer up to 2^{53}
  - Avoids conversions and extra memory storage when using FFLAS

Base field representation

- `Modular<double>`:
  - Based on machine `double` floating point representation
  - Only using the mantissa
    $\Rightarrow$Exact representation of integer up to 2^{53}
  - Avoids conversions and extra memory storage when using FFLAS
- `Givaro-ZpZ`:
  - based on machine integer (16,32 or 64 bits)
  - specialized dot-product (using delayed modulus)

Contents

Triangular System Solve: `trsm`

Compute a matrix $X \in K^{m \times n}$, s.t. $AX = B$.

Triangular System Solve: `trsm`

Compute a matrix $X \in K^{m \times n}$, s.t. $AX = B$.

- Used for numerical computations (in the BLAS)

Triangular System Solve: `trsm`

Compute a matrix $X \in K^{m \times n}$, s.t. $AX = B$.

- Used for numerical computations (in the BLAS)
- Building block for triangularization block algorithms

Triangular System Solve: `trsm`

Compute a matrix $X \in K^{m \times n}$, s.t. $AX = B$.

- Used for numerical computations (in the BLAS)
- Building block for triangularization block algorithms
- Three different approaches for exact computation over a finite field :
  1. A block recursive algorithm
  2. A wrapping of the BLAS `dtrsm`
  3. A matrix-vector based routine

Triangular System Solve: `trsm`

Compute a matrix $X \in K^{m \times n}$, s.t. $AX = B$.

- Used for numerical computations (in the BLAS)
- Building block for triangularization block algorithms
- Three different approaches for exact computation over a finite field :
  1. A block recursive algorithm
  2. A wrapping of the BLAS `dtrsm`
  3. A matrix-vector based routine
- Cascade algorithms as solution

1. The block recursive algorithm

$$\begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

1. The block recursive algorithm

$$\begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

$X_2 :=$ recursive call on (A_3, B_2).

1. The block recursive algorithm

$$\begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

$X_2 :=$ recursive call on (A_3, B_2).
$B_1 := B_1 - A_2 X_2$.

1. The block recursive algorithm

$$\begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

$X_2 :=$ recursive call on (A_3, B_2).
$B_1 := B_1 - A_2 X_2$.
$X_1 :=$ recursive call on (A_1, B_1).

1. The block recursive algorithm

$$\begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

$X_2 :=$ recursive call on (A_3, B_2).
$B_1 := B_1 - A_2 X_2$.
$X_1 :=$ recursive call on (A_1, B_1).

$\Rightarrow$ Reduces to matrix multiplication

1. The block recursive algorithm

$$\begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

$X_2 :=$ recursive call on (A_3, B_2).
$B_1 := B_1 - A_2 X_2$.
$X_1 :=$ recursive call on (A_1, B_1).

$\Rightarrow$ Reduces to matrix multiplication
$\rightarrow O(n^\omega)$ algebraic time complexity

1. The block recursive algorithm

$$\begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

$X_2 :=$ recursive call on (A_3, B_2).
$B_1 := B_1 - A_2 X_2$.
$X_1 :=$ recursive call on (A_1, B_1).

$\Rightarrow$ Reduces to matrix multiplication

$\longrightarrow$ $O(n^\omega)$ algebraic time complexity

$\longrightarrow$ Efficiency of FFLAS

2. Wrapping the BLAS `dtrsm`

- Same approach as for the matrix multiplication in FFLAS:
  - Conversion : Finite Field $\longrightarrow$ Real (`double`)
  - Computation over the real (using BLAS `dtrsm`)
  - Conversion : Real (`double`) $\longrightarrow$ Finite Field

2. Wrapping the BLAS `dtrsm`

- Same approach as for the matrix multiplication in FFLAS:
  - Conversion : Finite Field $\longrightarrow$ Real (`double`)
  - Computation over the real (using BLAS `dtrsm`)
  - Conversion : Real (`double`) $\longrightarrow$ Finite Field
- Two constraints:
  - No division must occur during BLAS computation
  - No overflow

2. Wrapping the BLAS `dtrsm`

First constraint: Divisions must be exact in

$$x_i = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=i+1}^{n} a_{i,j} x_j \right)$$

$\Rightarrow A$ must have a unit diagonal.

$\Rightarrow$Precondition A: $U = AD_A^{-1}$, solve $UY = B$, $X = D_A^{-1}Y$ where D_A is the diagonal of A.

2. Wrapping the BLAS `dtrsm`

Second constraint: No overflow during the BLAS computation

$\Rightarrow$Must bound the growth of the coefficients:

2. Wrapping the BLAS `dtrsm`

Second constraint: No overflow during the BLAS computation

$\Rightarrow$Must bound the growth of the coefficients:

- Naive: $(p-1)p^{n-1} < 2^m$

2. Wrapping the BLAS `dtrsm`

Second constraint: No overflow during the BLAS computation

$\Rightarrow$Must bound the growth of the coefficients:

- Naive:$(p-1)p^{n-1} < 2^m$
- Using a classical prime field repr.: $0 \leq x \leq p-1$:

$$\Rightarrow \frac{p-1}{2}\left[p^{n-1} + (p-2)^{n-1}\right] < 2^m$$

2. Wrapping the BLAS `dtrsm`

Second constraint: No overflow during the BLAS computation

$\Rightarrow$Must bound the growth of the coefficients:

- Naive: $(p-1)p^{n-1} < 2^m$
- Using a classical prime field repr.: $0 \leq x \leq p-1$:

$$\Rightarrow \frac{p-1}{2}\left[p^{n-1} + (p-2)^{n-1}\right] < 2^m$$

- Using a centered prime field repr.: $-\frac{p-1}{2} \leq x \leq \frac{p-1}{2}$:

$$\Rightarrow \frac{p-1}{2}\left(\frac{p+1}{2}\right)^{n-1} < 2^m$$

2. Wrapping the BLAS `dtrsm`

⇒Limit the matrix order for a given prime p:

2. Wrapping the BLAS `dtrsm`

$\Rightarrow$Limit the matrix order for a given prime p:

For $m = 53$:

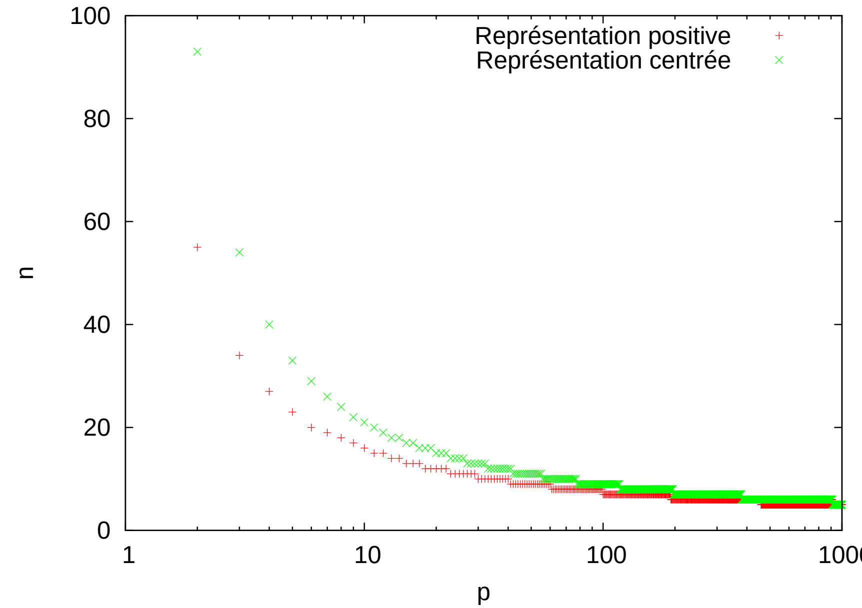
- $p = 2 \Rightarrow n \leq 55$
- $p = 9739 \Rightarrow n \leq 4$
- $p = 94906249 \Rightarrow n \leq 2$

2. Wrapping the BLAS `dtrsm`

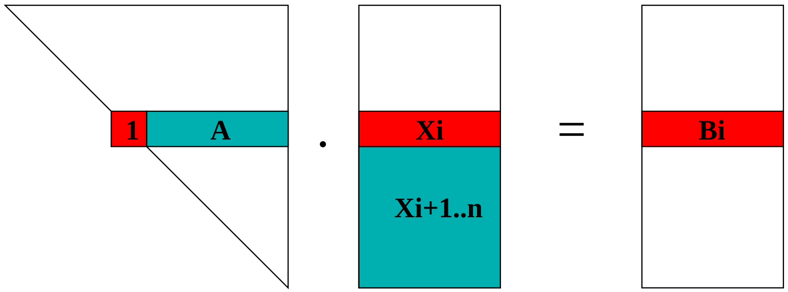
⇒Limit the matrix order for a given prime p:

For $m = 53$:

- $p = 2 \Rightarrow n \leq 55$
- $p = 9739 \Rightarrow n \leq 4$
- $p = 94906249 \Rightarrow n \leq 2$



3. Using Matrix-vector products

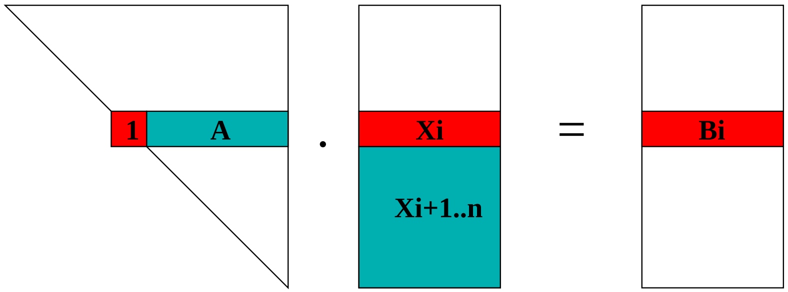


$X_i = B_i - A.X_{i+1..n}$

$\Rightarrow$ Matrix vector product

- Different implementations:

3. Using Matrix-vector products

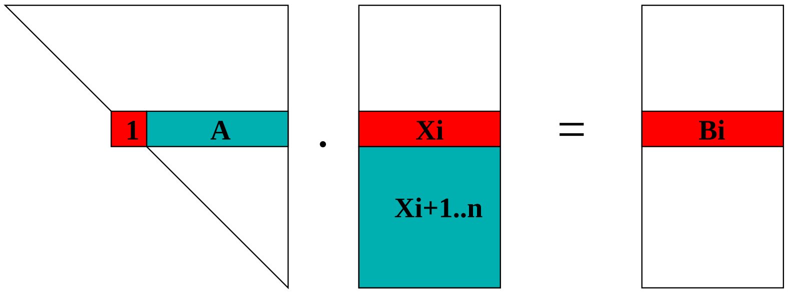


$$X_i = B_i - A.X_{i+1..n}$$

$\Rightarrow$ Matrix vector product

- Different implementations:
  - Using `modular<double>`: BLAS `gemv` and modulo

3. Using Matrix-vector products

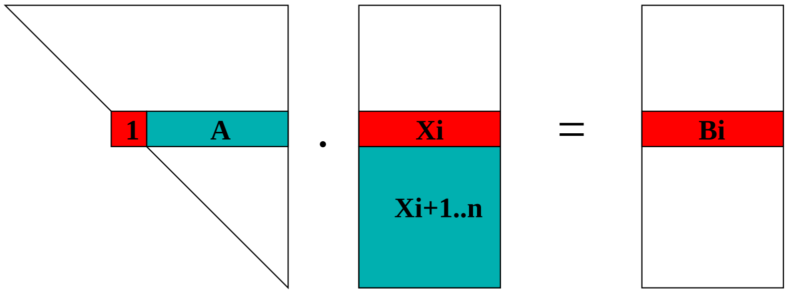


$$X_i = B_i - A.X_{i+1..n}$$

$\Rightarrow$ Matrix vector product

- Different implementations:
  - Using `modular<double>`: BLAS `gemv` and modulo
  - Using integral representations: design of specialized dot-product routines [Dumas CASC'04]

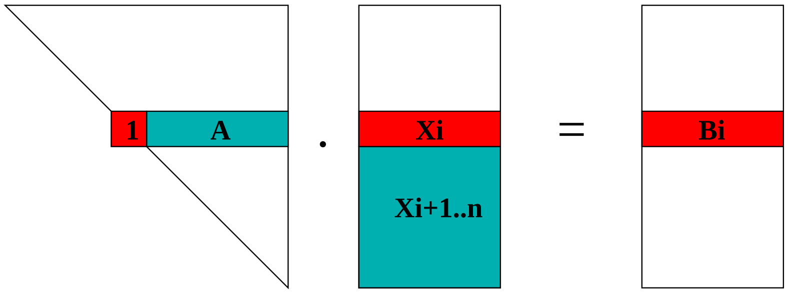
3. Using Matrix-vector products



$X_i = B_i - A.X_{i+1..n}$
$\Rightarrow$Matrix vector product

- Different implementations:
  - Using `modular<double>`: BLAS `gemv` and modulo
  - Using integral representations: design of specialized dot-product routines [Dumas CASC'04]
- Advantages:
  - Delayed modulus (1 for each row of X)

3. Using Matrix-vector products

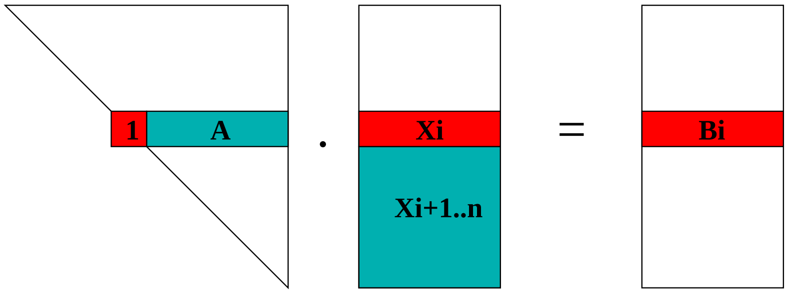


$X_i = B_i - A.X_{i+1..n}$

$\Rightarrow$ Matrix vector product

- Different implementations:
  - Using `modular<double>`: BLAS `gemv` and modulo
  - Using integral representations: design of specialized dot-product routines [Dumas CASC'04]
- Advantages:
  - Delayed modulus (1 for each row of X)
  - nicer bound: $n(p-1)^2 < 2^m$

3. Using Matrix-vector products



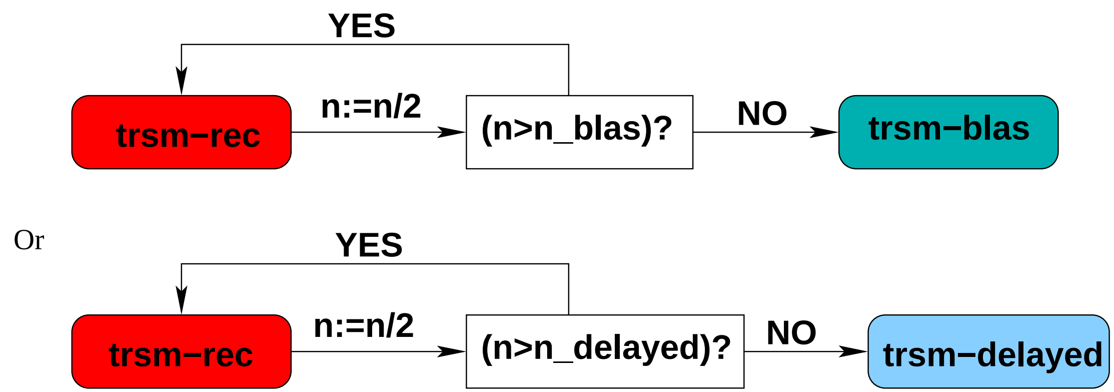
$$X_i = B_i - A.X_{i+1..n}$$

$\Rightarrow$Matrix vector product

- Different implementations:
  - Using modular<double>: BLAS gemv and modulo
  - Using integral representations: design of specialized dot-product routines [Dumas CASC'04]
- Advantages:
  - Delayed modulus (1 for each row of X)
  - nicer bound: $n(p-1)^2 < 2^m$
- Drawback: less efficient for large matrices ($n \geq 100$)

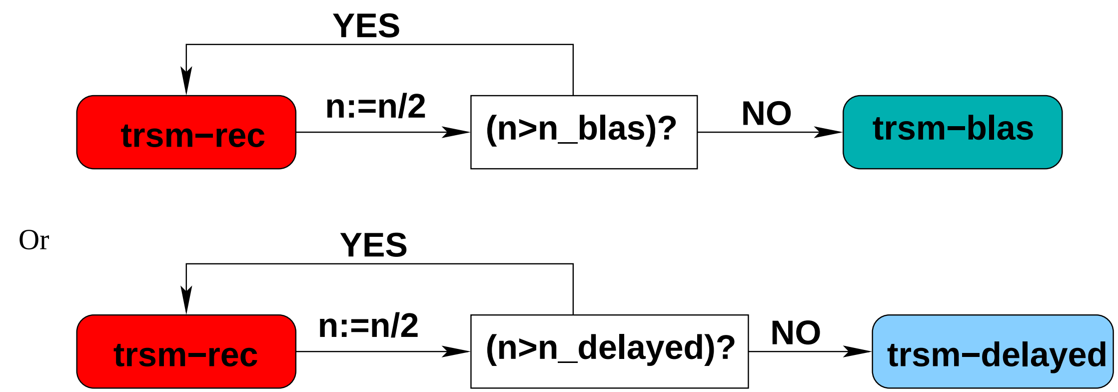
Two cascade algorithms

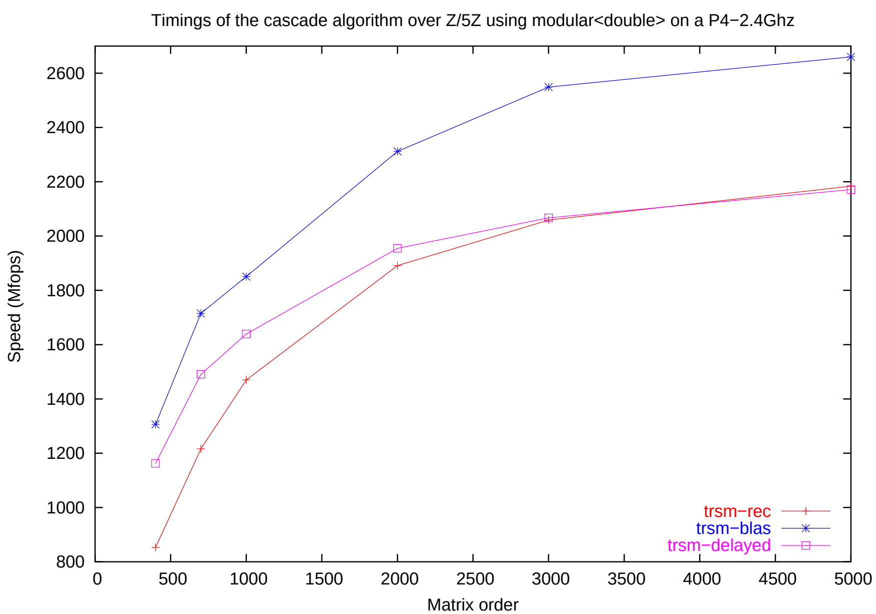
Idea:

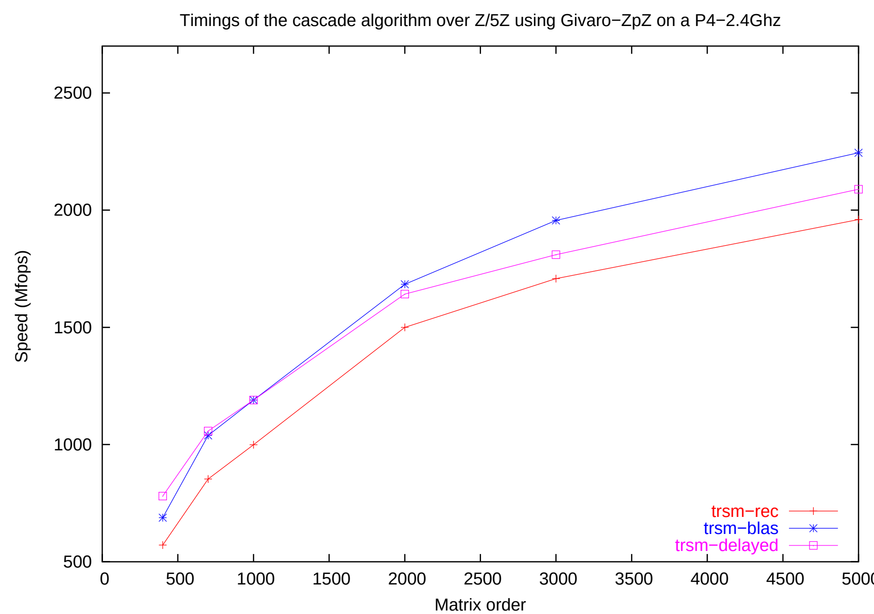


Two cascade algorithms

Idea:



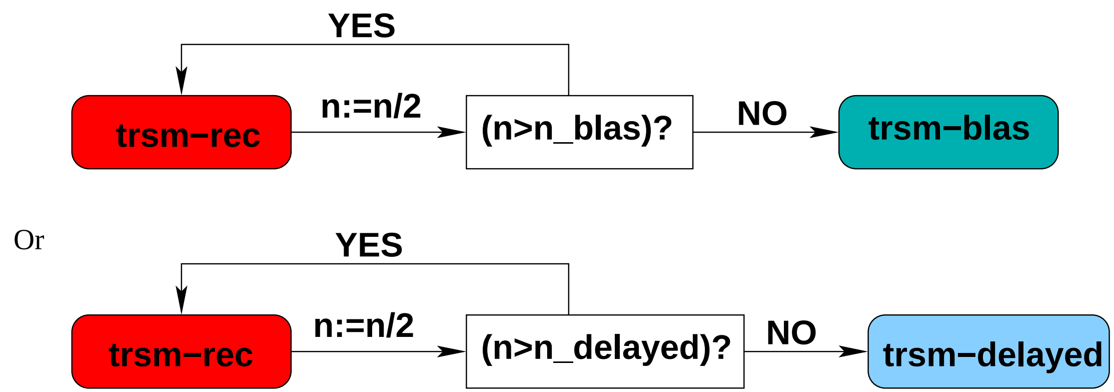




`modular<double>` $p = 5$ `Givaro-ZpZ` $p = 5$

Two cascade algorithms

Idea:



Conclusion

1. - `trsm-blas` highly depends on the prime
   - `trsm-delayed` do not

Conclusion

1. - `trsm-blas` highly depends on the prime
   - `trsm-delayed` do not
2. If the field representation can be chosen
   $\Rightarrow$Use `Modular<double>` with `trsm-blas`

Conclusion

1. - `trsm-blas` highly depends on the prime
   - `trsm-delayed` do not
2. If the field representation can be chosen
   ⇒Use `Modular<double>` with `trsm-blas`
3. Otherwise
   ⇒For some cases, a specialization of dot-product can slightly outperform `trsm-blas`

Contents

Triangularization

Specific issues:

- Have to deal with singular matrices
- Memory requirements

Triangularization

Specific issues:

- Have to deal with singular matrices
- Memory requirements

Provide a better analysis of the algebraic time complexity:

- improves the constant of the dominant term
- giving $T = \frac{2}{3}n^3 + O(n^2)$ in the nonsingular case with classic matrix multiplication

Triangularization

Specific issues:

- Have to deal with singular matrices
- Memory requirements

Provide a better analysis of the algebraic time complexity:

- improves the constant of the dominant term
- giving $T = \frac{2}{3}n^3 + O(n^2)$ in the nonsingular case with classic matrix multiplication

We will compare 3 implementations:

- `LSP`: a block recursive algorithm [Ibara & Al.]
- `LUdivine`: `LSP` with lesser memory requirements
- `LQUP` : Fully in-place triangularization

LSP algorithms

LSP [Ibara]:

- Split the row dimension

LSP algorithms

LSP [Ibara]:

- Split the row dimension
- recursive call on A_1

LSP algorithms

LSP [Ibara]:

- Split the row dimension
- recursive call on A_1
- $G \leftarrow A_{21}U_1^{-1}$

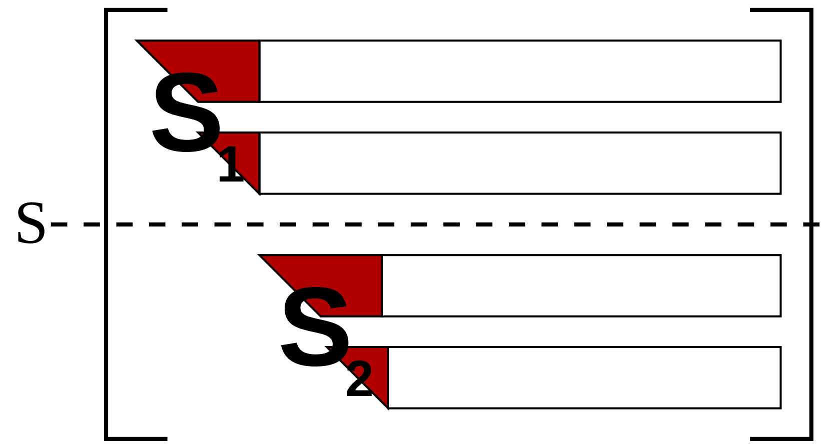
LSP algorithms

LSP [Ibara]:

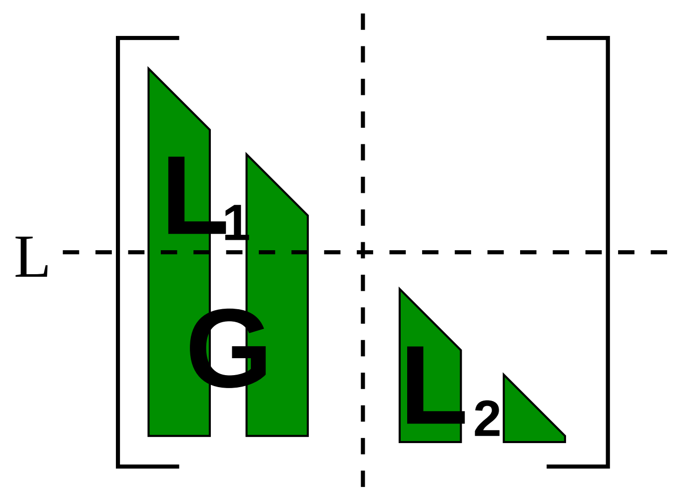
- Split the row dimension
- recursive call on A_1
- $G \leftarrow A_{21}U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GV$

LSP algorithms

LSP [Ibara]:



- Split the row dimension
- recursive call on A_1
- $G \leftarrow A_{21}U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GV$
- recursive call on A_{22}



LSP algorithms

LUdivine: *result is in place*

- Split the row dimension

LSP algorithms

LUdivine: *result is in place*

- Split the row dimension
- recursive call on A_1

LSP algorithms

LUdivine: *result is in place*

- Split the row dimension
- recursive call on A_1
- $G \leftarrow A_{21} U_1^{-1}$

LSP algorithms

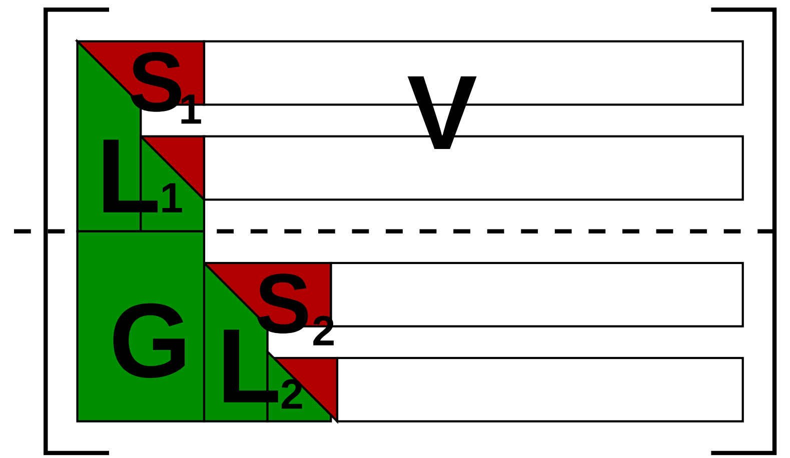
LUdivine: *result is in place*

- Split the row dimension
- recursive call on A_1
- $G \leftarrow A_{21} U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GW$

LSP algorithms

LUdivine: *result is in place*

- Split the row dimension
- recursive call on A_1
- $G \leftarrow A_{21} U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GW$
- recursive call on A_{22}



LSP algorithms

LQUP: *fully in place*

- Split the row dimension

LSP algorithms

LQUP: *fully in place*

- Split the row dimension
- recursive call on A_1

LSP algorithms

LQUP: *fully in place*

- Split the row dimension
- recursive call on A_1
- $G \leftarrow A_{21}U_1^{-1}$

LSP algorithms

LQUP: *fully in place*

- Split the row dimension
- recursive call on A_1
- $G \leftarrow A_{21} U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GV$

LSP algorithms

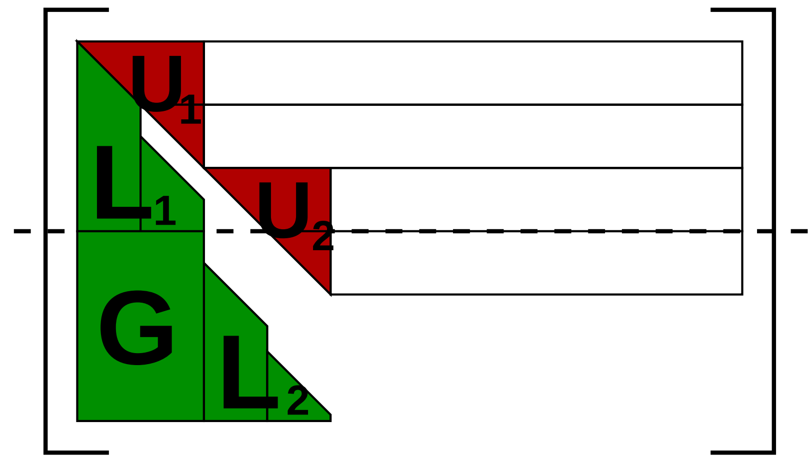
LQUP: *fully in place*

- Split the row dimension
- recursive call on A_1
- $G \leftarrow A_{21}U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GV$
- recursive call on A_{22}

LSP algorithms

LQUP: *fully in place*

- Split the row dimension
- recursive call on A_1
- $G \leftarrow A_{21} U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GV$
- recursive call on A_{22}
- row permutations



Comparisons

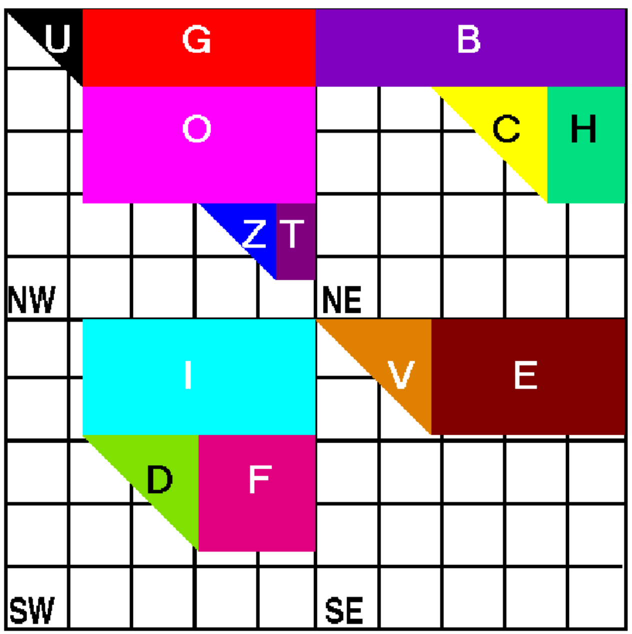
n	*1000*	*3000*	*5000*	*8000*	*10000*
`LSP`	0.48	8.01	32.54	404.8	1804
`LUdivine`	0.47	7.79	30.27	403.9	1691
`LQUP`	0.45	7.59	29.90	201.7	1090

- Similar timings when matrix fit in the RAM
- `LQUP` is slightly faster
- `LQUP` is fully in-place $\Rightarrow$ no swap for $n = 8000$

Dealing with data Locality

- Application: parallelism, out of core computations
- Use square recursive blocked data structure

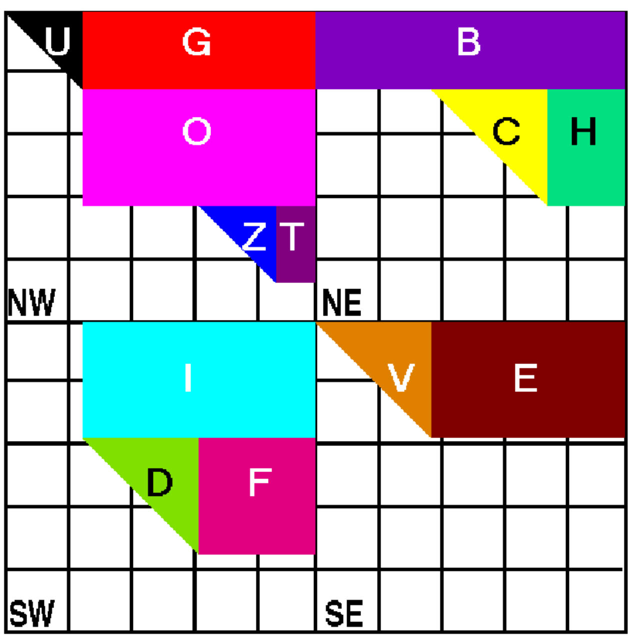
A triangularization algorithm: TURBO

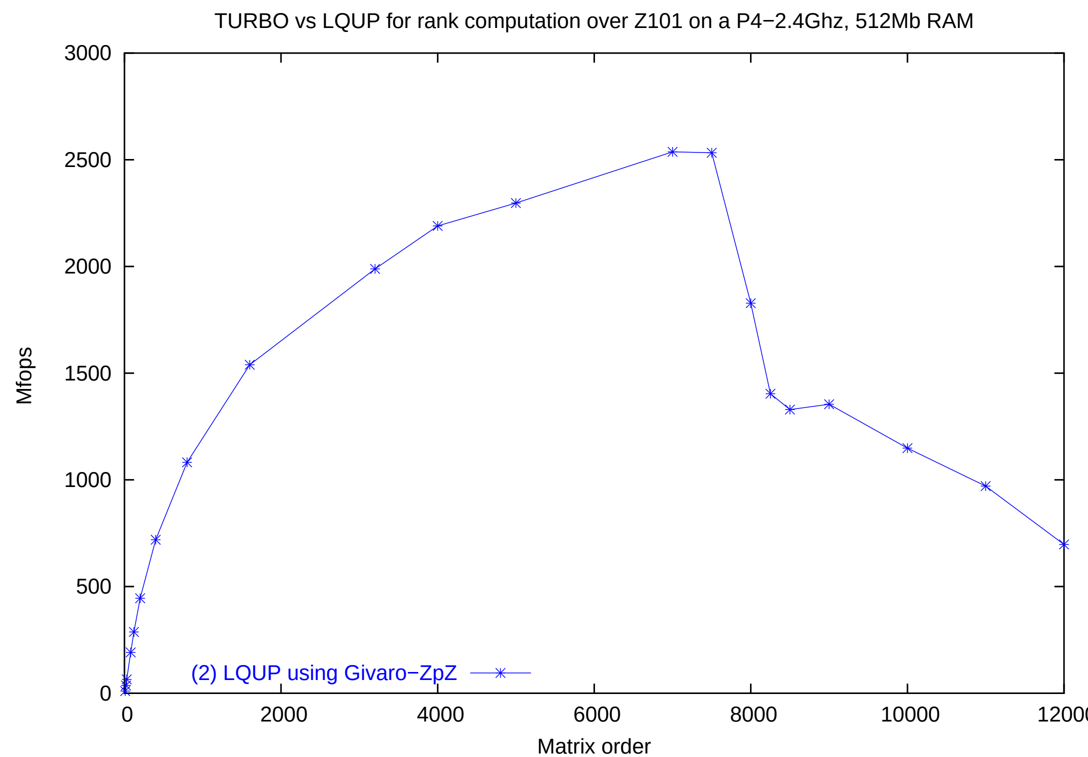


Dealing with data Locality

- Application: parallelism, out of core computations
- Use square recursive blocked data structure

A triangularization algorithm: TURBO

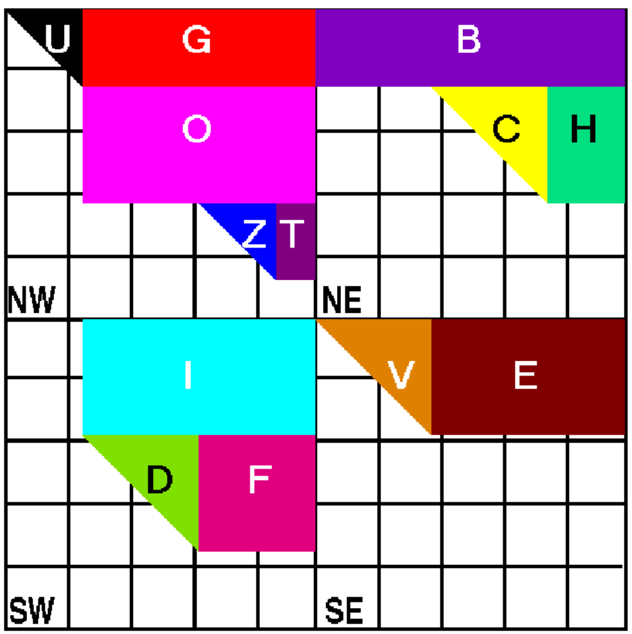


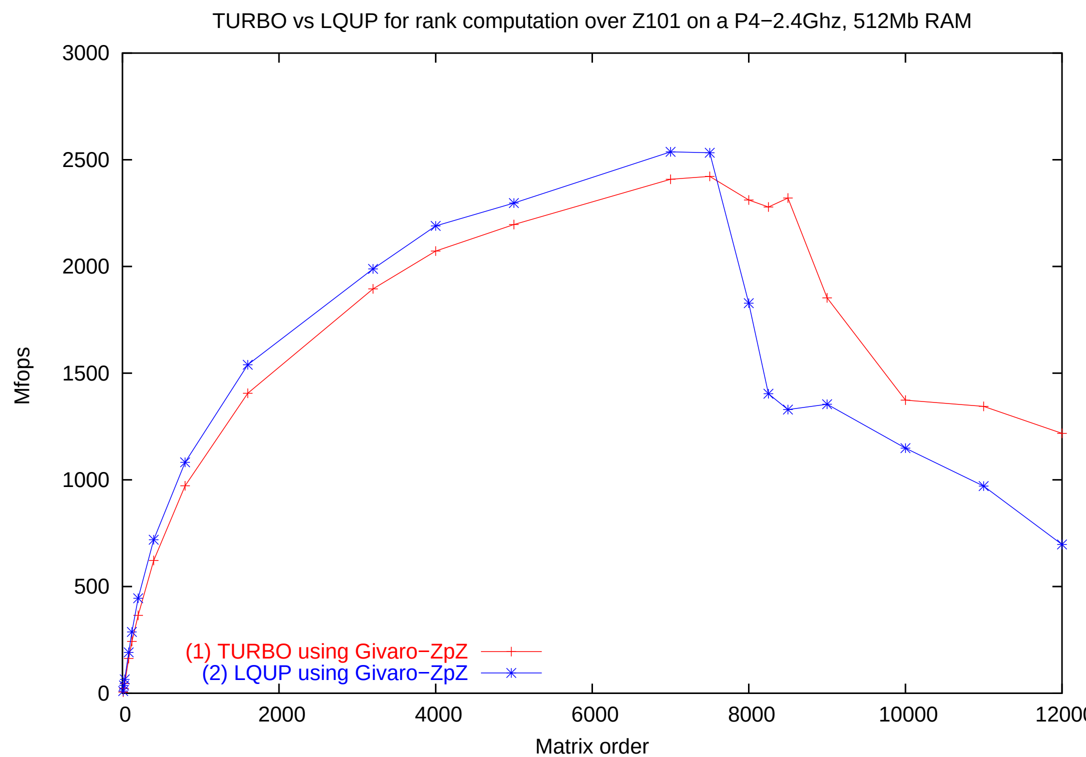


Dealing with data Locality

- Application: parallelism, out of core computations
- Use square recursive blocked data structure

A triangularization algorithm: TURBO





Conclusion

Results:

- Approach the timings of numerical routines:
  - $6.5s$ for a numeric LUP of a 3000×3000 matrix
  - $7.6s$ for a symbolic LQUP of a 3000×3000 matrix

Conclusion

Results:

- Approach the timings of numerical routines:
  - $6.5s$ for a numeric LUP of a 3000×3000 matrix
  - $7.6s$ for a symbolic LQUP of a 3000×3000 matrix
- Improved memory management of LSP factorization
- Further analysis of LSP time complexity.

Conclusion

Results:

- Approach the timings of numerical routines:
  - $6.5s$ for a numeric LUP of a 3000×3000 matrix
  - $7.6s$ for a symbolic LQUP of a 3000×3000 matrix
- Improved memory management of LSP factorization
- Further analysis of LSP time complexity.
- Optimal bounds for the coefficient growth in `trsm`

Conclusion

Results:

- Approach the timings of numerical routines:
  - $6.5s$ for a numeric LUP of a 3000×3000 matrix
  - $7.6s$ for a symbolic LQUP of a 3000×3000 matrix
- Improved memory management of LSP factorization
- Further analysis of LSP time complexity.
- Optimal bounds for the coefficient growth in `trsm`
- Part of the LinBox library [`http://linalg.org`]

Conclusion

Conclusion:

- Again: Wrapping numerical routines as much as possible appears to be the best choice
- When not possible (ex LSP)
  $\Rightarrow$block recursive algorithms

Conclusion

Conclusion:

- Again: Wrapping numerical routines as much as possible appears to be the best choice
- When not possible (ex LSP) $\Rightarrow$block recursive algorithms
- BLAS $\Rightarrow$No modulo
  $\Rightarrow$Need to control the coefficient growth
  $\Rightarrow$Bounds for correctness

Conclusion

Conclusion:

- Again: Wrapping numerical routines as much as possible appears to be the best choice
- When not possible (ex LSP)
  $\Rightarrow$block recursive algorithms
- BLAS $\Rightarrow$No modulo
  $\Rightarrow$Need to control the coefficient growth
  $\Rightarrow$Bounds for correctness
- Cascade structure
  $\Rightarrow$Switches between algorithms due to
  - Correctness constraints (theoretical thresholds)
  - Performance constraints (experimental thresholds)

Further developments

- Self adapting software: automatic setup of optimal experimental thresholds

Further developments

- Self adapting software: automatic setup of optimal experimental thresholds
- Apply of the factorization to other applications: characteristic polynomial, null space, . . .

FFPACK: Finite Field Linear Algebra Package

Jean-Guillaume Dumas, Pascal Giorgi and Clément Pernet

pascal.giorgi@ens-lyon.fr, {Jean.Guillaume.Dumas, Clément.Pernet}@imag.fr