Genericity and efficiency in exact linear algebra with the FFLAS-FFPACK and LinBox libraries

Clément Pernet & the LinBox group

U. Joseph Fourier (Grenoble 1, Inria/LIP AriC)

Sage Days 66
Liège, 11 Mars 2015

Introduction

Computer Algebra

Computing **exactly** over $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}/p\mathbb{Z}, \mathrm{GF}(q), \mathrm{K}[X]$.

- Symbolic manipulations.
- Applications where all digits matter:

- breaking Discrete Log Pb. in quasi-polynomial time [Barbulescu & *al.* 14],
- building modular form databases to test the BSD conjecture [Stein 12],
- formal verification of Hales' proof of Kepler conjecture [Hales 05].

Efficiency mostly rely on linear algebra over $\mathbb{Z}$ and $\mathbb{Z}/p\mathbb{Z}$.

Exact linear algebra

Matrices can be

- **Dense:** store all coefficients
- **Sparse:** store the non-zero coefficients only
- **Black-box:** no access to the storage, only *apply* to a vector

Exact linear algebra

Matrices can be

Dense: store all coefficients

Sparse: store the non-zero coefficients only

Black-box: no access to the storage, only *apply* to a vector

Coefficient domains:

Word size:
- integers with a priori bounds
- $\mathbb{Z}/p\mathbb{Z}$ for p of ≈ 32 bits

Multi-precision: $\mathbb{Z}/p\mathbb{Z}$ for p of $\approx 100, 200, 1000, 2000, \ldots$ bits

Arbitrary precision: $\mathbb{Z}, \mathbb{Q}$

Polynomials: K$[X]$ for K any of the above

Exact linear algebra

Matrices can be

Dense: store all coefficients

Sparse: store the non-zero coefficients only

Black-box: no access to the storage, only *apply* to a vector

Coefficient domains:

Word size:
- integers with a priori bounds
- $\mathbb{Z}/p\mathbb{Z}$ for p of ≈ 32 bits

Multi-precision: $\mathbb{Z}/p\mathbb{Z}$ for p of $\approx 100, 200, 1000, 2000, \ldots$ bits

Arbitrary precision: $\mathbb{Z}, \mathbb{Q}$

Polynomials: K$[X]$ for K any of the above

Several implementations for the same domain: better fits FFT, LinAlg, etc

Exact linear algebra

Matrices can be

Dense: store all coefficients

Sparse: store the non-zero coefficients only

Black-box: no access to the storage, only *apply* to a vector

Coefficient domains:

Word size:
- integers with a priori bounds
- $\mathbb{Z}/p\mathbb{Z}$ for p of ≈ 32 bits

Multi-precision: $\mathbb{Z}/p\mathbb{Z}$ for p of $\approx 100, 200, 1000, 2000, \ldots$ bits

Arbitrary precision: $\mathbb{Z}, \mathbb{Q}$

Polynomials: K$[X]$ for K any of the above

Several implementations for the same domain: better fits FFT, LinAlg, etc

Requires genericity.

Exact linear algebra

Which computation?

Comp. Number Theory:	CharPoly, LinSys, Echelon, over $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}/p\mathbb{Z}$, Dense
Graph Theory:	MatMul, CharPoly, Det, over $\mathbb{Z}$, Sparse
Discrete log.:	LinSys, over $\mathbb{Z}/p\mathbb{Z}$, $p \approx 120$ bits, Sparse
Integer Factorization:	NullSpace, over $\mathbb{Z}/2\mathbb{Z}$, Sparse
Algebraic Attacks:	Echelon, LinSys, over $\mathbb{Z}/p\mathbb{Z}$, $p \approx 20$ bits, Sparse & Dense
List decoding of RS codes:	Lattice reduction, over GF$(q)[X]$, Structured

Exact linear algebra

Which computation?

Comp. Number Theory:	CharPoly, LinSys, Echelon, over $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}/p\mathbb{Z}$, Dense
Graph Theory:	MatMul, CharPoly, Det, over $\mathbb{Z}$, Sparse
Discrete log.:	LinSys, over $\mathbb{Z}/p\mathbb{Z}$, $p \approx 120$ bits, Sparse
Integer Factorization:	NullSpace, over $\mathbb{Z}/2\mathbb{Z}$, Sparse
Algebraic Attacks:	Echelon, LinSys, over $\mathbb{Z}/p\mathbb{Z}$, $p \approx 20$ bits, Sparse & Dense
List decoding of RS codes:	Lattice reduction, over GF$(q)[X]$, Structured

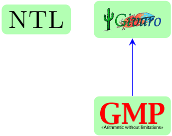
Requires high performance.

Software stack for exact linear algebra

Arithmetic

GMP, MPIR: multiprecision integers and rationals

Givaro, NTL: finite fields and polynomials



Software stack for exact linear algebra

Arithmetic

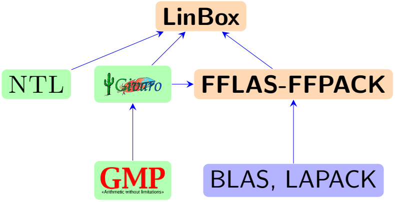
GMP, MPIR: multiprecision integers and rationals

Givaro, NTL: finite fields and polynomials

BLAS: Basic Linear Algebra Subroutines (floating point)

FFLAS-FFPACK: Basic Exact Linear Algebra over $\mathbb{Z}/p\mathbb{Z}$,

LinBox: Linear Algebra over $\mathbb{Z}, \mathbb{Z}/p\mathbb{Z}$ and K$[X]$



Software stack for exact linear algebra

Arithmetic

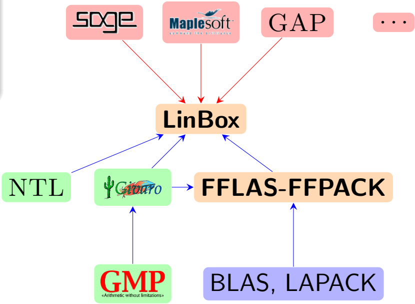
GMP, MPIR: multiprecision integers and rationals

Givaro, NTL: finite fields and polynomials

BLAS: Basic Linear Algebra Subroutines (floating point)

FFLAS-FFPACK: Basic Exact Linear Algebra over $\mathbb{Z}/p\mathbb{Z}$,

LinBox: Linear Algebra over $\mathbb{Z}, \mathbb{Z}/p\mathbb{Z}$ and K$[X]$



Outline

The LinBox project

- International collaboration: Canada, USA, France
- Strongly generic `C++` code, focus on efficiency
- Free software (LGPL 2.1+)
- ≈ 200 K loc
- `http://linalg.org/`

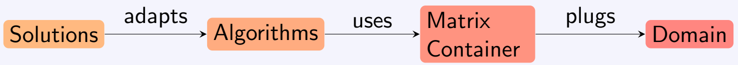
The LinBox project

- International collaboration: Canada, USA, France
- Strongly generic `C++` code, focus on efficiency
- Free software (LGPL 2.1+)
- ≈ 200 K loc
- `http://linalg.org/`

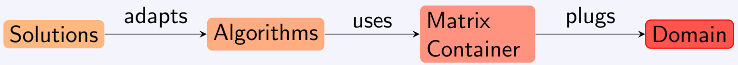
Milestones

1998	First design: Black box and sparse matrices
2003	Dense linear algebra using BLAS $\rightsquigarrow$ FFLAS-FFPACK
2005	LinBox-1.0
2008	Integration in Sage
2012-..	Parallelization
2014	SIMD & Sparse BLAS in FFLAS-FFPACK (Brice's talk)

Architecture (design)



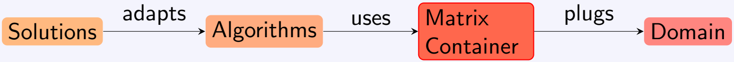
Architecture (design)



Genericity w.r.t the domain

- modular arithmetic
- finite fields
- integers, rationals
- polynomials

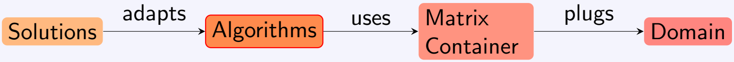
Architecture (design)



Genericity w.r.t the matrix type

- Dense
- Structured
- Blackbox ($x \rightarrow Ax$ or block $X \rightarrow AX$)
- Sparse

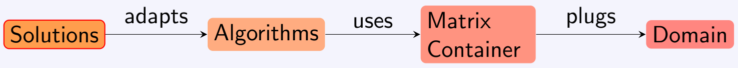
Architecture (design)



Various algorithms

- Blackbox (Lanczos, Wiedemann, block variants)
- Gaussian elimination...
- BLAS modular linear algebra (FFPACK)
- $p-$adic, CRA, early termination...

Architecture (design)



Solutions

- solve
- det
- rank
- charpoly
- ...

Architecture (Genericity)

Domain % element:

```
template <class Element>
class Modular<Element>; // Z/pZ
```

Architecture (Genericity)

Domain % element:

```
template <class Element>
class Modular<Element>; // Z/pZ
```

Matrix % domain:

```
template <class Field>
class BlasMatrix<Field>; // dense matrix
```

Architecture (Genericity)

Domain % element:

```
template <class Element>
class Modular<Element>; // Z/pZ
```

Matrix % domain:

```
template <class Field>
class BlasMatrix<Field>; // dense matrix
```

Solutions % matrix:

```
template <class Matrix>
unsigned long & rank(unsigned long & r,
                     const Matrix & A);
```

Architecture (Example)

Example : det.h

```
#include "linbox/integer.h"
#include "linbox/blackbox/blas-blackbox.h"
#include "linbox/solutions/det.h"
#include "linbox/util/matrix-stream.h"

typedef PID_integer     Domain;
Domain ZZ;
MatrixStream<Domain> ms( ZZ, input );
BlasBlackbox<Domain> A(ms);
Domain::Element det_A;
det(det_A, A);
```

Architecture (Example)

Example : det.h

```
#include "linbox/field/modular.h"
#include "linbox/blackbox/sparse.h"
#include "linbox/solutions/det.h"
#include "linbox/util/matrix-stream.h"

typedef Modular<double> Domain;
Domain F(65537) ;
MatrixStream<Domain> ms( F , input );
SparseMatrix<Domain> A(ms);
Domain::Element det_A;
det(det_A, A);
```

Outline

Black box linear algebra

Black box linear algebra

- Matrices viewed as linear operators
- algorithms based on matrix-vector apply only $\rightsquigarrow$ cost $E(n)$

Black box linear algebra

- Matrices viewed as linear operators
- algorithms based on matrix-vector apply only $\rightsquigarrow$ cost $E(n)$

Structured matrices: Fast apply (e.g. $E(n) = O(n \log n)$)

Sparse matrices: Fast apply and no fill-in

Black box linear algebra

- Matrices viewed as linear operators
- algorithms based on matrix-vector apply only $\rightsquigarrow$ cost $E(n)$

Structured matrices: Fast apply (e.g. $E(n) = O(n \log n)$)

Sparse matrices: Fast apply and no fill-in

$\rightsquigarrow$

- Iterative methods
- No access to coefficients, trace, no elimination
- Matrix **multiplication** $\Rightarrow$ Black-box **composition**

Example: blackbox composition

```
template <class Mat1, class Mat2>
class Compose {
  protected:
    Mat1 _A;
    Mat2 _B;
  public:
    Compose(Mat1& A, Mat2& B) : _A(A), _B(B) {}

    template<class InVec, class OutVec>
    OutVec& apply (const InVec& x) {
      return _A.apply(_B.apply(x));
    }
};
```

Black box linear algebra

Matrix-Vector Product: building block,
$\rightsquigarrow$ costs $E(n)$

Minimal polynomial: [Wiedemann 86]
$\rightsquigarrow$ iterative Krylov/Lanczos methods
$\rightsquigarrow O(nE(n) + n^2)$

Black box linear algebra

Matrix-Vector Product: building block,
$\rightsquigarrow$ costs $E(n)$

Minimal polynomial: [Wiedemann 86]
$\rightsquigarrow$ iterative Krylov/Lanczos methods
$\rightsquigarrow O(nE(n) + n^2)$

Rank, Det, Solve: [Chen& Al. 02]
$\rightsquigarrow$ reduces to MinPoly + preconditioners
$\rightsquigarrow \tilde{O}(nE(n) + n^2)$

Black box linear algebra

Matrix-Vector Product: building block,
$\rightsquigarrow$ costs $E(n)$

Minimal polynomial: [Wiedemann 86]
$\rightsquigarrow$ iterative Krylov/Lanczos methods
$\rightsquigarrow O(nE(n) + n^2)$

Rank, Det, Solve: [Chen& Al. 02]
$\rightsquigarrow$ reduces to MinPoly + preconditioners
$\rightsquigarrow \tilde{O}(nE(n) + n^2)$

Characteristic Poly.: [Dumas P. Saunders 09]
$\rightsquigarrow$ reduces to MinPoly, Rank, . . .

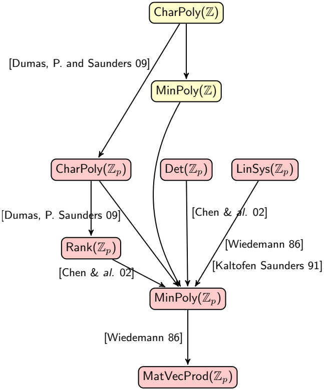
Black box linear algebra

Matrix-Vector Product: building block,
$\rightsquigarrow$ costs $E(n)$

Minimal polynomial: [Wiedemann 86]
$\rightsquigarrow$ iterative Krylov/Lanczos methods
$\rightsquigarrow O(nE(n) + n^2)$

Rank, Det, Solve: [Chen& Al. 02]
$\rightsquigarrow$ reduces to MinPoly + preconditioners
$\rightsquigarrow O\tilde{}(nE(n) + n^2)$

Characteristic Poly.: [Dumas P. Saunders 09]
$\rightsquigarrow$ reduces to MinPoly, Rank, ...



Outline

Reductions: linear algebra's arithmetic complexity

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskĭi, etc)

Reductions: linear algebra's arithmetic complexity

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskĭi, etc)

Matrix Product

[Strassen 69]: $O(n^{2.807})$

$\vdots$

[Schönhage 81] $O(n^{2.52})$

$\vdots$

[Coppersmith, Winograd 90] $O(n^{2.375})$

$\vdots$

[Le Gall 14] $O(n^{2.3728639})$

$\rightsquigarrow$ $\mathsf{MM}(n) = O(n^\omega)$

Reductions: linear algebra's arithmetic complexity

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskĭi, etc)

Matrix Product

[Strassen 69]: $O(n^{2.807})$

$\vdots$

[Schönhage 81] $O(n^{2.52})$

$\vdots$

[Coppersmith, Winograd 90] $O(n^{2.375})$

$\vdots$

[Le Gall 14] $O(n^{2.3728639})$

$\rightsquigarrow$ $\mathsf{MM}(n) = O(n^\omega)$

Other operations

[Strassen 69]: Inverse in $O(n^\omega)$

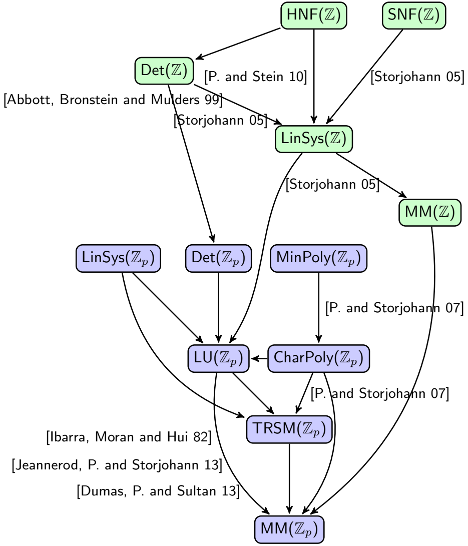
[Schönhage 72]: QR in $O(n^\omega)$

[Bunch, Hopcroft 74]: LU in $O(n^\omega)$

[Ibarra & *al.* 82]: Rank in $O(n^\omega)$

[Keller-Gehrig 85]: CharPoly in $O(n^\omega \log n)$

Reductions


HNF($\mathbb{Z}$)
SNF($\mathbb{Z}$)
Det($\mathbb{Z}$)
[P. and Stein 10]
[Storjohann 05]
[Abbott, Bronstein and Mulders 99]
[Storjohann 05]
LinSys($\mathbb{Z}$)
[Storjohann 05]
MM($\mathbb{Z}$)
LinSys($\mathbb{Z}_p$)
Det($\mathbb{Z}_p$)
MinPoly($\mathbb{Z}_p$)
[P. and Storjohann 07]
LU($\mathbb{Z}_p$)
CharPoly($\mathbb{Z}_p$)
[P. and Storjohann 07]
TRSM($\mathbb{Z}_p$)
[Ibarra, Moran and Hui 82]
[Jeannerod, P. and Storjohann 13]
[Dumas, P. and Sultan 13]
MM($\mathbb{Z}_p$)

Making theoretical reductions effective

Making theoretical reductions effective

Common mistrust

Fast linear algebra is

- ✗ never faster
- ✗ numerically unstable

Making theoretical reductions effective

Common mistrust

Fast linear algebra is

- ✗ never faster
- ✗ numerically unstable

Lucky coincidence

- ✓ building blocks **in theory** happen to be the most efficient routines **in practice**

$\leadsto$ reduction trees are still relevant

Making theoretical reductions effective

Common mistrust

Fast linear algebra is

- ✗ never faster
- ✗ numerically unstable

Lucky coincidence

- ✓ building blocks **in theory** happen to be the most efficient routines **in practice**

$\rightsquigarrow$ reduction trees are still relevant

Roadmap

1. Tune building blocks (MatMul)
2. Improve existing reductions (LU, Echelon)
   - ▷ leading constants
   - ▷ memory footprint
3. Produce new reduction schemes (CharPoly)

Matrix Multiplication over $\mathbb{Z}/p\mathbb{Z}$

Ingedients [Dumas, Gautier and P. 02]

- Compute over $\mathbb{Z}$ and delay modular reductions

$\rightsquigarrow$ $k\left(\frac{p-1}{2}\right)^2 < 2^{\text{mantissa}}$

Matrix Multiplication over $\mathbb{Z}/p\mathbb{Z}$

Ingedients [Dumas, Gautier and P. 02]

- Compute over $\mathbb{Z}$ and delay modular reductions

$$\rightsquigarrow k\left(\frac{p-1}{2}\right)^2 < 2^{\text{mantissa}}$$

- Fastest integer arithmetic: `double`, `float` (SIMD and pipeline)
- Cache optimizations

$\rightsquigarrow$ numerical BLAS

Matrix Multiplication over $\mathbb{Z}/p\mathbb{Z}$

Ingedients [Dumas, Gautier and P. 02]

- Compute over $\mathbb{Z}$ and delay modular reductions

$$\rightsquigarrow \quad 9^\ell \left\lfloor \frac{k}{2^\ell} \right\rfloor \left(\frac{p-1}{2}\right)^2 < 2^{\text{mantissa}}$$

- Fastest integer arithmetic: `double`, `float` (SIMD and pipeline)
- Cache optimizations

$\rightsquigarrow$ numerical BLAS

- Strassen-Winograd $6n^{2.807} + \dots$

Matrix Multiplication over $\mathbb{Z}/p\mathbb{Z}$

Ingedients [Dumas, Gautier and P. 02]

- Compute over $\mathbb{Z}$ and delay modular reductions

$$\leadsto 9^\ell \left\lfloor \frac{k}{2^\ell} \right\rfloor \left(\frac{p-1}{2}\right)^2 < 2^{\text{mantissa}}$$

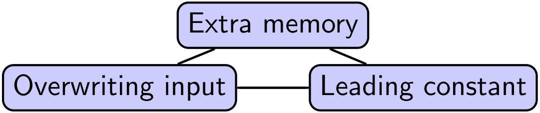
- Fastest integer arithmetic: `double`, `float` (SIMD and pipeline)
- Cache optimizations

$\leadsto$ numerical BLAS

- Strassen-Winograd $6n^{2.807} + \dots$

with memory efficient schedules [Boyer, Dumas, P. and Zhou 09]

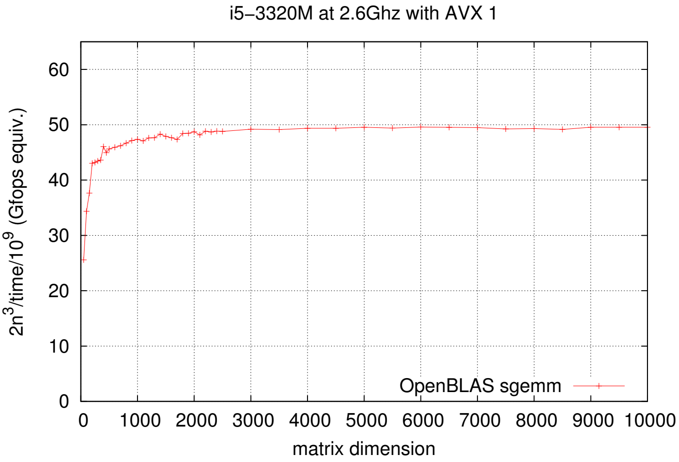
Tradeoffs:



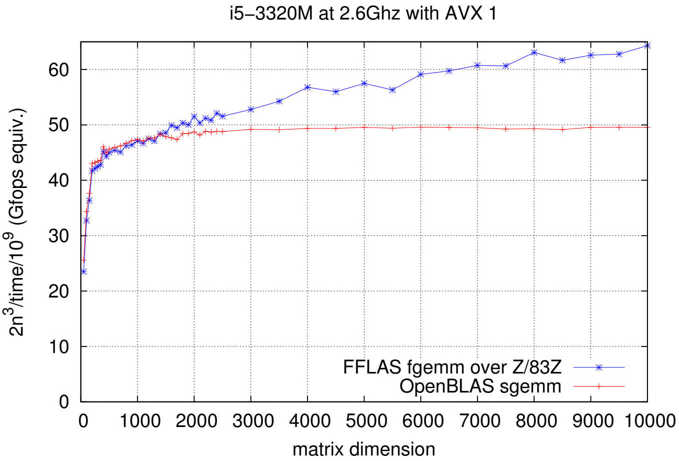
Fully in-place in

$7.2n^{2.807} + \dots$

Sequential Matrix Multiplication

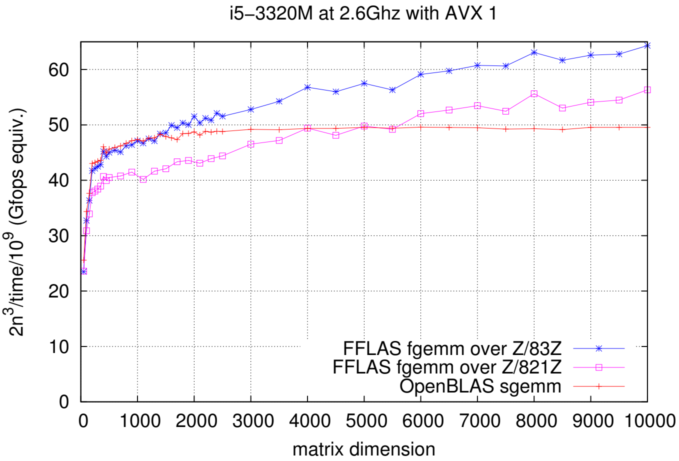


Sequential Matrix Multiplication



$p = 83$, $\leadsto$ 1 mod / 10000 mul.

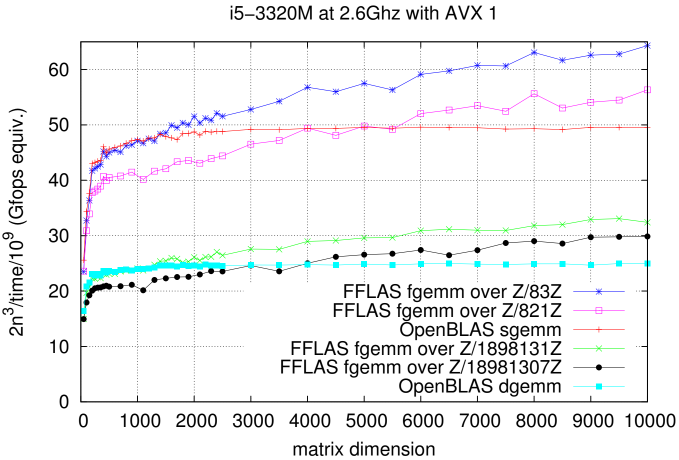
Sequential Matrix Multiplication



$p = 83$, $\rightsquigarrow$ 1 mod / 10000 mul.

$p = 821$, $\rightsquigarrow$ 1 mod / 100 mul.

Sequential Matrix Multiplication



$p = 83$, $\rightsquigarrow$ 1 mod / 10000 mul.

$p = 821$, $\rightsquigarrow$ 1 mod / 100 mul.

$p = 1898131$, $\rightsquigarrow$ 1 mod / 10000 mul.

$p = 18981307$, $\rightsquigarrow$ 1 mod / 100 mul.

Other routines

LU decomposition

- Block recursive algorithm $\rightsquigarrow$ reduces to MatMul $\rightsquigarrow$ $O(n^\omega)$

n	1000	5000	10000	15000	20000
`LAPACK-dgetrf`	**0.024s**	**2.01s**	**14.88s**	48.78s	113.66
`fflas-ffpack`	0.058s	2.46s	16.08s	**47.47s**	**105.96s**

Intel Haswell E3-1270 3.0Ghz using OpenBLAS-0.2.9

Other routines

LU decomposition

- Block recursive algorithm $\rightsquigarrow$ reduces to MatMul $\rightsquigarrow$ $O(n^\omega)$

n	1000	5000	10000	15000	20000
`LAPACK-dgetrf`	**0.024s**	**2.01s**	**14.88s**	48.78s	113.66
`fflas-ffpack`	0.058s	2.46s	16.08s	**47.47s**	**105.96s**

Intel Haswell E3-1270 3.0Ghz using OpenBLAS-0.2.9

Characteristic Polynomial

- A new reduction to matrix multiplication in $O(n^\omega)$.

n	1000	2000	5000	10000
`magma-v2.19-9`	1.38s	24.28s	332.7s	2497s
`fflas-ffpack`	**0.532s**	**2.936s**	**32.71s**	**219.2s**

Intel Ivy-Bridge i5-3320 2.6Ghz using OpenBLAS-0.2.9

Other routines

LU decomposition

- Block recursive algorithm $\rightsquigarrow$ reduces to MatMul $\rightsquigarrow$ $O(n^\omega)$

n	1000	5000	10000	15000	20000
`LAPACK-dgetrf`	**0.024s**	**2.01s**	**14.88s**	48.78s	113.66
`fflas-ffpack`	0.058s	2.46s	16.08s	**47.47s**	**105.96s**

Intel Haswell E3-1270 3.0Ghz using OpenBLAS-0.2.9

$\times 7.63$

$\times 6.59$

Characteristic Polynomial

- A new reduction to matrix multiplication in $O(n^\omega)$.

n	1000	2000	5000	10000
`magma-v2.19-9`	1.38s	24.28s	332.7s	2497s
`fflas-ffpack`	**0.532s**	**2.936s**	**32.71s**	**219.2s**

Intel Ivy-Bridge i5-3320 2.6Ghz using OpenBLAS-0.2.9

$\times 7.5$

$\times 6.7$

Outline

Design of parallel exact linear algebra

ANR HPAC project:

1. efficient kernels for exact linear algebra on SMP
2. DSL, runtime as a plugin and composition
3. attacking large scale challenges from cryptography

Design of parallel exact linear algebra

ANR HPAC project: **Ziad Sultan PhD. Thesis**

1. efficient kernels for exact linear algebra on SMP
2. DSL, runtime as a plugin and composition
3. attacking large scale challenges from cryptography

Design of parallel exact linear algebra

ANR HPAC project: **Ziad Sultan PhD. Thesis**

1. **efficient kernels for exact linear algebra on SMP**
2. DSL, runtime as a plugin and composition
3. attacking large scale challenges from cryptography

Design of parallel exact linear algebra

ANR HPAC project: **Ziad Sultan PhD. Thesis**

1. **efficient kernels for exact linear algebra on SMP**
2. DSL, runtime as a plugin and composition
3. attacking large scale challenges from cryptography

Parallel numerical linear algebra

- cost invariant wrt. splitting
  - $O(n^3)$

  $\rightsquigarrow$ fine grain

  $\rightsquigarrow$ block iterative algorithms
- regular task load
- Numerical stability constraints

Design of parallel exact linear algebra

ANR HPAC project: **Ziad Sultan PhD. Thesis**

1. **efficient kernels for exact linear algebra on SMP**
2. DSL, runtime as a plugin and composition
3. attacking large scale challenges from cryptography

Parallel numerical linear algebra

- cost invariant wrt. splitting
  - $O(n^3)$

  $\rightsquigarrow$ fine grain

  $\rightsquigarrow$ block iterative algorithms
- regular task load
- Numerical stability constraints

Exact linear algebra specificities

- cost affected by the splitting
  - $O(n^\omega)$ for $\omega < 3$
  - modular reductions

  $\rightsquigarrow$ coarse grain

  $\rightsquigarrow$ recursive algorithms
- rank deficiencies

  $\rightsquigarrow$ unbalanced task loads

Ingredients for the parallelization

Criteria

- good performances
- portability across architectures
- abstraction for simplicity

Challenging key point: scheduling as a plugin

Program: only describes where the parallelism lies

Runtime: scheduling & mapping, depending on the context of execution

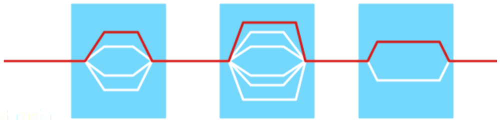
3 main models:

1. Parallel loop [data parallelism]
2. Fork-Join (independent tasks) [task parallelism]
3. Dependent tasks with data flow dependencies [task parallelism]

Data Parallelism

OMP

```
for (int step = 0; step < 2; ++step){
#pragma omp parallel for
    for (int i = 0; i < count; ++i)
        A[i] = (B[i+1] + B[i-1] + 2.0*B[i])*0.25;
}
```



Limitation: very un-efficient with recursive parallel regions

- Limited to iterative algorithms
- No composition of routines

Task parallelism with fork-Join

- Task based program: **spawn** + **sync**
- Especially suited for recursive programs

OMP (since v3)

```
void fibonacci(long* result, long n) {
  if (n < 2)
    *result = n;
  else {
    long x,y;
#pragma omp task
    fibonacci( &x, n-1 );
    fibonacci( &y, n-2 );
#pragma omp taskwait
    *result = x + y;
  }
}
```

Task parallelism with fork-join

- Task based program: **spawn** + **sync**
- Especially suited for recursive programs

Cilk+

```
long fibonacci(long n) {
  if (n < 2)
    return (n);
  else {
    long x, y;
    x = cilk_spawn fibonacci(n - 1);
    y = fibonacci(n - 2);
    cilk_sync;
    return (x + y);
  }
}
```

Task parallelism with fork Join

- Task based program: **spawn** + **sync**
- Especially suited for recursive programs

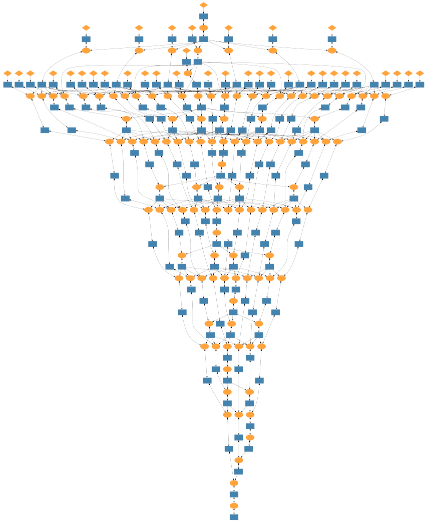
Kaapi

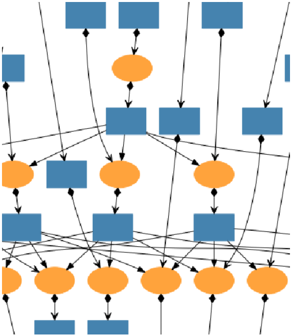
```
void fibonacci(long* result, long n) {
  if (n<2)
    *result = n;
  else {
    long x,y;
#pragma kaapi task
    fibonacci( &x, n-1 );
    fibonacci( &y, n-2 );
#pragma kaapi sync
    *result = x + y;
  }
}
```

Tasks with dataflow dependencies

- Task based model
- remove explicit synchronizations
- deduce synchronizations from the read/write specifications
- Basic definition:
  - A task is ready for execution when all its inputs variables are ready
  - A variable is ready when it has been written
- Old languages: ID, SISAL...
- New languages/libraries: Athapascan [96], Kaapi [06], StarSs [07], StarPU [08], Quark [10], OMP since v4 [14]...

Data flow graph: Cholesky factorization





SmpSS

```
#pragma smpss task write(array)
extern void compute( double* array, int count);
#pragma smpss task read(array)
extern void print( double* array, int count);
int main() {
#pragma smpss start
    compute( array, count);
    print( array, count);    // Read after write dependency
#pragma smpss sync
#pragma smpss finish
}
```

Kaapi

```
int main() {
#pragma kaapi parallel
  {
#  pragma kaapi task write(array[0..count])
    compute( array, count);
#  pragma kaapi task read(array[0..count])
    print( array, count);     // Read after write dependency
  } // implicit barrier at the end of Kaapi parallel region
}
```

Existing solutions

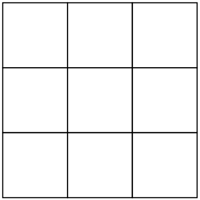
	// prog model	Architecture	Target app.
OMP 1.0 [97] OMP 3.0 [08] OMP 4.0 [14]	Parallel loop Fork-join Rec. Data Flow	Multi-CPUs Multi-CPUs Multi-CPUs	ForEach + Divide&Conquer
Cilk[96]	Fork-join	Multi-CPUs	Divide&Conquer
Athapascan[98]	Rec. Data flow	Clusters+multi-CPU	D&C, LinAlg
TBB[06]	Parallel loop Fork-join	Multi-CPU	D&C, LinAlg
Kaapi[06-12]	Rec. Data flow Parallel loop	Multi-CPUs & GPUs	D&C, LinAlg ForEach,
StarSs [07]	Flat data flow Flat data flow Flat data flow Flat data flow	multi-CPUs (SMPSs) multi-CPUs (SMPSs) Cell (CellSs) Grid (GridSs)	LinAlg LinAlg LinAlg LinAlg
StarPU [09]	Flat data flow	multi-CPUs&GPUs	LinAlg
Quark[10]	Flat data flow	Multi-CPUs	LinAlg

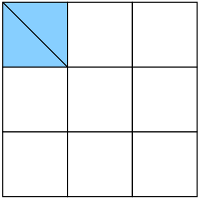
Illustration: Cholesky factorization

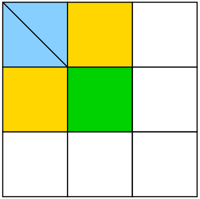
```
void Cholesky( double* A, int N, size_t NB ) {

  for (size_t k=0; k < N; k += NB)
  {
    clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

    for (size_t m=k+ NB; m < N; m += NB)
    {

      cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
        NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
    }

    for (size_t m=k+ NB; m < N; m += NB)
    {

      cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
        NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

      for (size_t n=k+NB; n < m; n += NB)
      {

        cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,
          NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
      }
    }

  }
}
```

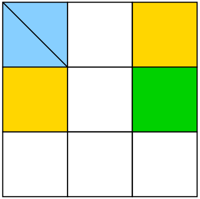
Illustration: Cholesky factorization

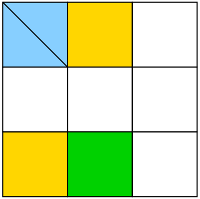
```
void Cholesky( double* A, int N, size_t NB ) {
#pragma omp parallel
#pragma omp single nowait
  for (size_t k=0; k < N; k += NB)
  {
    clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma omp task firstprivate(k, m) shared(A)
      cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
        NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
    }
#pragma omp taskwait // Barrier: no concurrency with next tasks
    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma omp task firstprivate(k, m) shared(A)
      cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
        NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

      for (size_t n=k+NB; n < m; n += NB)
      {
#pragma omp task firstprivate(k, m) shared(A)
        cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,
          NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
      }
    }
#pragma omp taskwait // Barrier: no concurrency with tasks at iteration k+1
  }
}
```

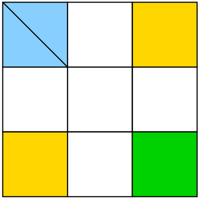












SYNC.

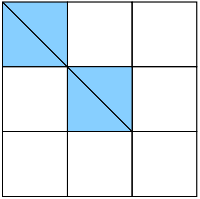
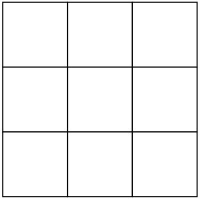
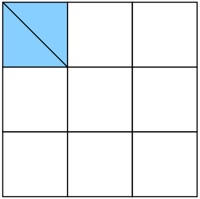
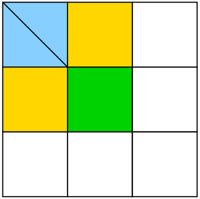


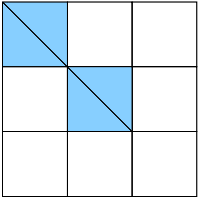
Illustration: Cholesky factorization

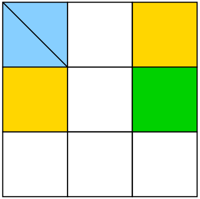
```
void Cholesky( double* A, int N, size_t NB ){
#pragma kaapi parallel
  for (size_t k=0; k < N; k += NB)
  {
#pragma kaapi task readwrite(&A[k*N+k]{ld=N; [NB][NB]})
    clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma kaapi task read(&A[k*N+k]{ld=N;[NB][NB]}) readwrite(&A[m*N+k]{ld=N; [NB][NB]})
      cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
        NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
    }

    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma kaapi task read(&A[m*N+k]{ld=N;[NB][NB]}) readwrite(&A[m*N+m]{ld=N; [NB][NB]})
      cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
        NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

      for (size_t n=k+NB; n < m; n += NB)
      {
#pragma kaapi task read(&A[m*N+k]{ld=N; [NB][NB]}, &A[n*N+k]{ld=N; [NB][NB]})\
                   readwrite(&A[m*N+n]{ld=N; [NB][NB]})
        cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,
          NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
      }
    }
  }
  // Implicit barrier only at the end of Kaapi parallel region
}
```

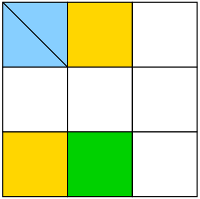


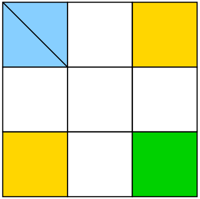












A DSL for parallel FFLAS-FFPACK

Difficult choice for a parallel language and runtime

OpenMP:

- Data parallelism (limited: no composition nor recursion)
- Fork-Join model satisfactory (was slow until v4.0)
- Dataflow dependencies: only recently (v4.0). Limited language for LinAlg data.

Cilk, TBB:

- Fork-join task model

Kaapi:

- Efficient tasks (lightweight)
- Replacement implementation for OMPv3 (`libkomp`).
- Better dataflow semantic, but still not accessible through OMP
- still protypical

DSL for FFLAS-FFPACK

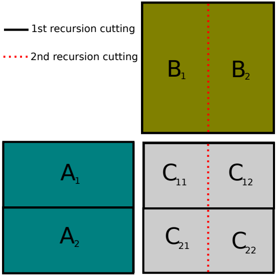
A unique programming language for parallelization

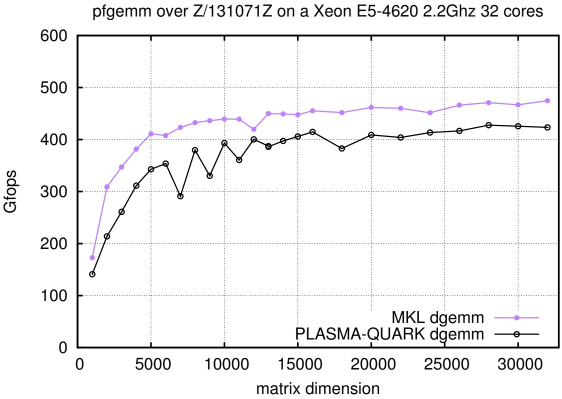
- Annotation (using macros)
- Supporting tasks with data flow dependencies
- fall back to fork-join model
- addresses: OMP v3,4, Kaapi, Cilk

```
// G = P3 [ L3 ] [ U3 V3 ] Q3
//        [ M3 ]
TASK (MODE (CONSTREFERENCE (Fi, G, Q3, P3, R3)
            WRITE (R3, P3, Q3) READWRITE(G[0])),
      R3 = pPLUQ (Fi, Diag, M-M2, N2-R1, G, lda, P3, Q3,nt/2));
// H <- A4 - ED
TASK( MODE (CONSTREFERENCE (Fi, A3, A2, A4, pWH)
            READ (M2, N2, R1, A3[0], A2[0])
            READWRITE(A4[0])),
      fgemm (Fi, FFLAS::FflasNoTrans, FFLAS::FflasNoTrans, M-M2, N-N2, R1,
             Fi.mOne, A3, lda, A2, lda, Fi.one, A4, lda, pWH));
CHECK_DEPENDENCIES;
// [ H1 H2 ] <- P3^T H Q2^T
// [ H3 H4 ]
TASK( MODE(READ(P3, Q2)
           CONSTREFERENCE (Fi, A4, Q2, P3)
           READWRITE (A4[0])),
      papplyP (Fi, FFLAS::FflasRight, FFLAS::FflasTrans, M-M2, 0, N-N2, A4, lda, Q2);
      papplyP (Fi, FFLAS::FflasLeft, FFLAS::FflasNoTrans, N-N2, 0, M-M2, A4, lda, P3););
CHECK_DEPENDENCIES;
```

Parallel matrix multiplication

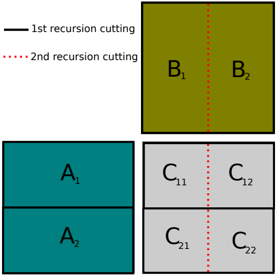
Dumas, Gautier, P. and Sultan 14

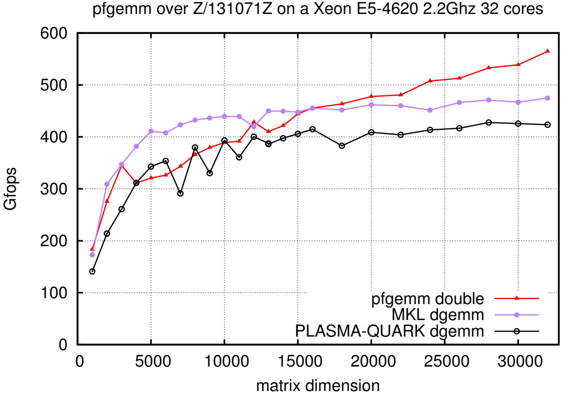




Parallel matrix multiplication

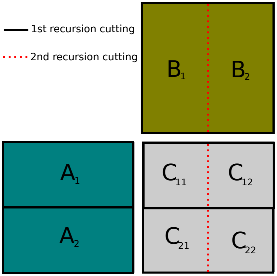
Dumas, Gautier, P. and Sultan 14

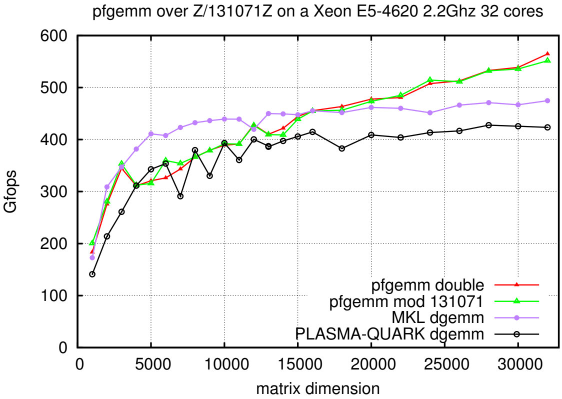




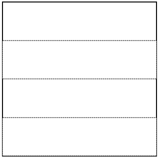
Parallel matrix multiplication

Dumas, Gautier, P. and Sultan 14



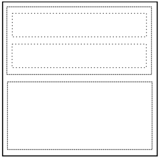


Gaussian elimination



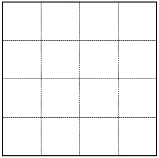
Slab iterative

LAPACK



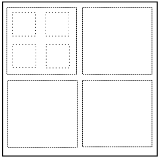
Slab recursive

FFLAS-FFPACK



Tile iterative

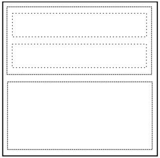
PLASMA



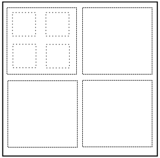
Tile recursive

FFLAS-FFPACK

Gaussian elimination



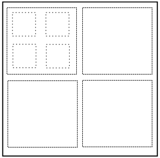
Slab recursive
FFLAS-FFPACK



Tile recursive
FFLAS-FFPACK

- Prefer recursive algorithms

Gaussian elimination



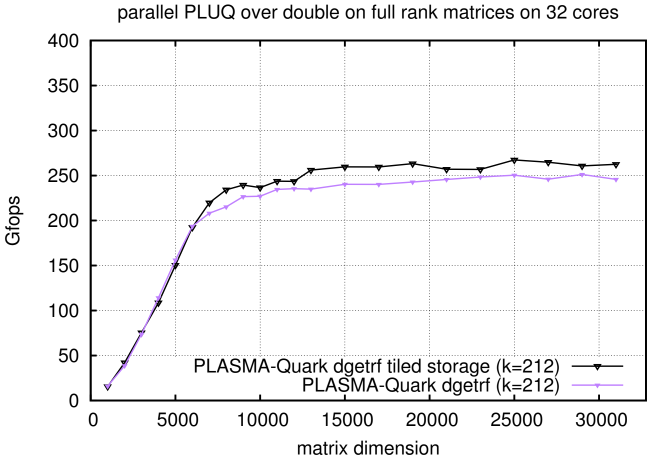
Tile recursive
FFLAS-FFPACK

- Prefer recursive algorithms
- Better data locality

Full rank Gaussian elimination

Dumas, Gautier, P. and Sultan 14

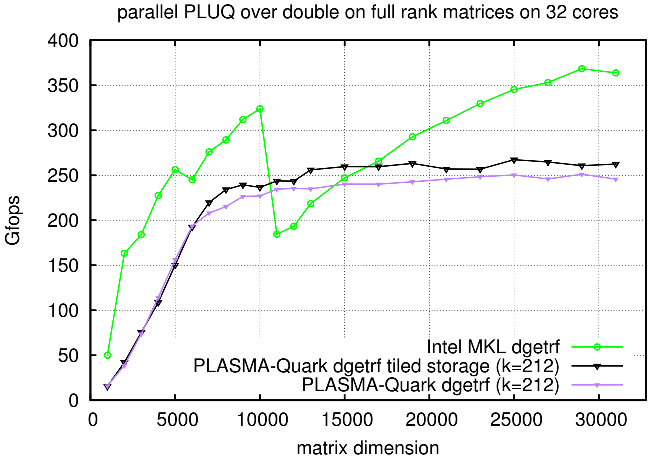
Comparing numerical efficiency (no modulo)



Full rank Gaussian elimination

Dumas, Gautier, P. and Sultan 14

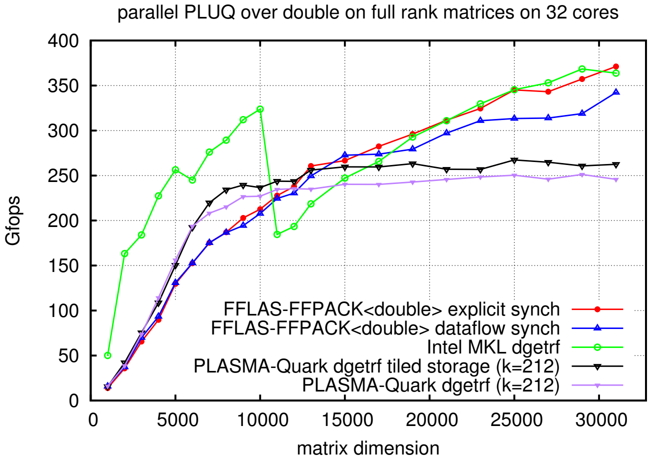
Comparing numerical efficiency (no modulo)



Full rank Gaussian elimination

Dumas, Gautier, P. and Sultan 14

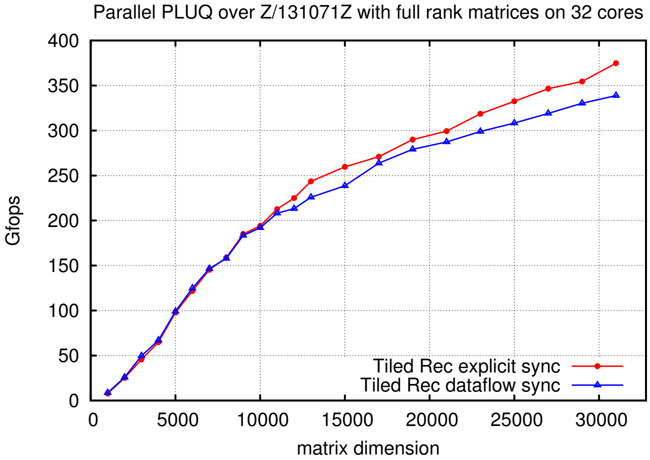
Comparing numerical efficiency (no modulo)



Full rank Gaussian elimination

Dumas, Gautier, P. and Sultan 14

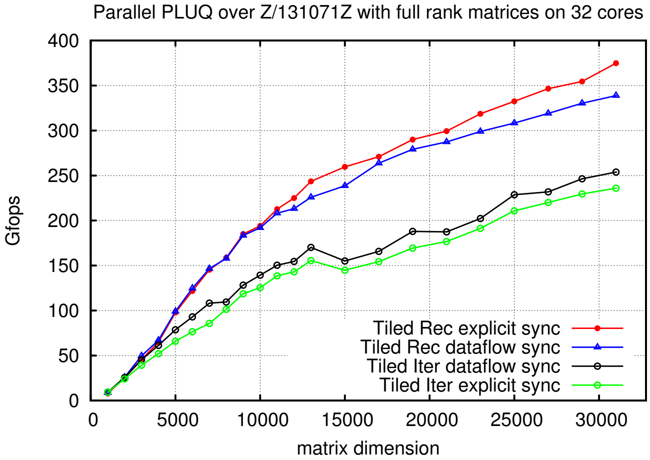
Over the finite field $\mathbb{Z}/131071\mathbb{Z}$



Full rank Gaussian elimination

Dumas, Gautier, P. and Sultan 14

Over the finite field $\mathbb{Z}/131071\mathbb{Z}$



Thank You.