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)

Séminaire Performance et Généricité,
LRDE EPITA, Paris,
11 Mars 2015
Computing exactly over $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}/p\mathbb{Z}, \text{GF}(q), K[X]$.  

- 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].
Introduction

Computer Algebra

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

- 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 $\approx$ 32 bits
- **Multi-precision**: $\mathbb{Z}/p\mathbb{Z}$ for $p$ of $\approx$ 100, 200, 1000, 2000, ... 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 $\approx 32$ 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 \( \approx 32 \) 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 implemenations for the same domain: better fits FFT, LinAlg, etc

Requires genericity.
## Exact linear algebra

### Which computation?

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

Requires high performance.
Software stack for exact linear algebra

**Arithmetic**

- **GMP**: multiprecision integers and rationals
- **MPIR**: multiprecision integers and rationals
- **GAP**: finite fields and polynomials
- **NTL**: finite fields and polynomials
- **FFLAS-FFPACK**: Basic Exact Linear Algebra over \( \mathbb{Z}/p\mathbb{Z} \)
- **LinBox**: Linear Algebra over \( \mathbb{Z} \), \( \mathbb{Z}/p\mathbb{Z} \) and \( \mathbb{K}[X] \)
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 $K[X]$
Software stack for exact linear algebra

**Arithmetic**

- **GMP**: multiprecision integers and rationals
- **MPIR**: multiprecision integers and rationals
- **Fina**: 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

1. The LinBox library
2. Blackbox linear algebra
3. Dense linear algebra
4. Parallelization
The LinBox project

- International collaboration: Canada, USA, France
- Strongly generic C++ code, focus on efficiency
- Free software (LGPL 2.1+)
- $\approx 200$ 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 \( \leadsto \) FFLAS-FFPACK
- 2005 LinBox-1.0
- 2008 Integration in Sage
- 2012-.. Parallelization
- 2014 SIMD & Sparse BLAS in FFLAS-FFPACK (Brice’s talk)
Architecture (design)

Solutions adapts Algorithms uses Matrix Container plugs Domain
The LinBox library

Architecture (design)

Solutions adapts Algorithms uses Matrix Container plugs Domain

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 \((x \rightarrow Ax \text{ or block } X \rightarrow AX)\)
- Sparse
The LinBox library

Architecture (design)

---

Various algorithms

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

Solutions adapts Algorithms uses Matrix Container plugs Domain

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

C. Pernet (UJF)
The LinBox library

Architecture (Genericity)

Domain % element:

```cpp
template <class Element>
class Modular<Element>; // Z/pZ
```
Architecture (Genericity)

Domain % element:

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

Matrix % domain:

```cpp
template <class Field>
class BlasMatrix<Field>; // dense matrix
```
The LinBox library

Architecture (Genericity)

Domain % element:

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

Matrix % domain:

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

Solutions % matrix:

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

Example: det.h

```c
#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);
```
The LinBox library

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

1. The LinBox library
2. Blackbox linear algebra
3. Dense linear algebra
4. Parallelization
Black box linear algebra

- Matrices viewed as linear operators
- Algorithms based on matrix-vector apply only
- Cost \( E(n) \)
- Structured matrices: Fast apply (e.g. \( E(n) = O(n \log n) \))
- Sparse matrices: Fast apply and no fill-in

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

C. Pernet (UJF)
Black box linear algebra

- Matrices viewed as linear operators
- Algorithms based on matrix-vector apply only \( \sim \) cost \( E(n) \)

\[ A \in \mathbb{K}^{n \times n} \]

\[ v \in \mathbb{K}^n \quad \rightarrow \quad Av \in \mathbb{K}^n \]
Matrices viewed as linear operators

- algorithms based on matrix-vector apply only $\leadsto$ cost $E(n)$

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

Sparse matrices: Fast apply and no fill-in
Matrices viewed as linear operators

- algorithms based on matrix-vector apply only \( \sim \) cost \( E(n) \)

\[ A \in \mathbb{K}^{n \times n} \]
\[ v \in \mathbb{K}^n \quad \rightarrow \quad Av \in \mathbb{K}^n \]

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

Sparse matrices: Fast apply and no fill-in

- Iterative methods
- No access to coefficients, trace, no elimination
- Matrix multiplication \( \Rightarrow \) Black-box composition
Example: blackbox composition

```cpp
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));
        }
};
```
Matrix-Vector Product: building block, $ightarrow$ costs $E(n)$

Minimal polynomial: [Wiedemann 86]
$ightarrow$ iterative Krylov/Lanczos methods
$ightarrow 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
\( \rightsquigarrow O(\tilde{n}E(n) + n^2) \)
Matrix-Vector Product: building block,
\[ \sim \text{costs } E(n) \]

Minimal polynomial: [Wiedemann 86]
\[ \sim \text{iterative Krylov/Lanczos methods} \]
\[ \sim O(nE(n) + n^2) \]

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

Characteristic Poly.: [Dumas P. Saunders 09]
\[ \sim \text{reduces to MinPoly, Rank, …} \]
Matrix-Vector Product: building block, \( \sim \) costs \( E(n) \)

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

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

Characteristic Poly.: [Dumas P. Saunders 09]
\( \sim \) reduces to MinPoly, Rank, \ldots
Outline

1. The LinBox library
2. Blackbox linear algebra
3. Dense linear algebra
4. Parallelization
Reductions: linear algebra’s arithmetic complexity

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskii, etc)
Reductions: linear algebra’s arithmetic complexity

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

Matrix Product

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

[Schönhage 81]: $O(n^{2.52})$

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

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

$\sim \text{MM}(n) = O(n^\omega)$
Dense linear algebra

Reductions: linear algebra’s arithmetic complexity

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

Matrix Product

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

[Schönhage 81]: $O(n^{2.52})$

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

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

$\leadsto$ 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 $O(n^\omega \log n)$
Dense linear algebra

Reductions

[Abbott, Bronstein and Mulders 99]
[Storjohann 05]

Det(Z) [P. and Stein 10]

LinSys(Z) [Storjohann 05]

LinSys(Z_p) [Ibarra, Moran and Hui 82]
[Jeannerod, P. and Storjohann 13]
[Dumas, P. and Sultan 13]

MM(Z)

HNF(Z) [P. and Storjohann 07]

SNF(Z) [Storjohann 05]

Det(Z_p) [Storjohann 05]

LinSys(Z_p) [P. and Storjohann 07]

MinPoly(Z_p) [P. and Storjohann 07]

CharPoly(Z_p)

LU(Z_p) [P. and Storjohann 07]

TRSM(Z_p)

MM(Z_p) [Storjohann 05]
Dense linear algebra

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

⇝ 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
⇝ 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 $\mathbb{Z}/p\mathbb{Z}$

**Inredients [Dumas, Gautier and P. 02]**

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

$$k \left( \frac{p-1}{2} \right)^2 < 2^\text{mantissa}$$
Matrix Multiplication over $\mathbb{Z}/p\mathbb{Z}$

**Ingredients [Dumas, Gautier and P. 02]**

- Compute over $\mathbb{Z}$ and delay modular reductions
  \[ k \left( \frac{p-1}{2} \right)^2 < 2^{\text{mantissa}} \]
- Fastest integer arithmetic: double, float (SIMD and pipeline)
- Cache optimizations
  \[ \Rightarrow \text{numerical BLAS} \]
Matrix Multiplication over $\mathbb{Z}/p\mathbb{Z}$

Ingredients [Dumas, Gautier and P. 02]

- Compute over $\mathbb{Z}$ and delay modular reductions
  \[ 9\ell \lfloor \frac{k}{2\ell} \rfloor \left( \frac{p-1}{2} \right)^2 < 2^{\text{mantissa}} \]
- Fastest integer arithmetic: double, float (SIMD and pipeline)
- Cache optimizations
  \[ \sim \text{numerical BLAS} \]
- Strassen-Winograd $6n^{2.807} + \ldots$
Matrix Multiplication over $\mathbb{Z}/p\mathbb{Z}$

**Ingredients [Dumas, Gautier and P. 02]**

- Compute over $\mathbb{Z}$ and delay modular reductions
  \[ 9^\ell \left\lfloor \frac{k}{2^\ell} \right\rfloor \left( \frac{p-1}{2} \right)^2 < 2^{\text{mantissa}} \]
- Fastest integer arithmetic: double, float (SIMD and pipeline)
- Cache optimizations \(\leadsto\) numerical BLAS
- Strassen-Winograd $6n^{2.807} + \ldots$

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

**Tradeoffs:**

- Extra memory
- Overwriting input
- Leading constant
- Fully in-place in $7.2n^{2.807} + \ldots$
Sequential Matrix Multiplication

\[ 2n^3 / \text{time} / 10^9 \text{ (Gflops equiv.)} \]

matrix dimension

\[ \text{i5-3320M at 2.6Ghz with AVX 1} \]

OpenBLAS sgemm

\[ p = 83, \Rightarrow 1 \mod 1000 \text{ mul.} \]

\[ p = 821, \Rightarrow 1 \mod 10 \text{ mul.} \]
Sequential Matrix Multiplication

\[ p = 83, \sim 1 \mod / 10000 \text{ mul.} \]
Sequential Matrix Multiplication

$p = 83, \sim 1 \mod / 10000 \text{ mul.}$

$p = 821, \sim 1 \mod / 100 \text{ mul.}$

C. Pernet (UJF)
Dense linear algebra

Sequential Matrix Multiplication

![Graph showing sequential matrix multiplication performance](image)

- **FFLAS fgemm over Z/83Z**
- **FFLAS fgemm over Z/821Z**
- **OpenBLAS sgemm**
- **FFLAS fgemm over Z/1898131Z**
- **FFLAS fgemm over Z/18981307Z**
- **OpenBLAS dgemm**

### Performance Metrics

- **$2n^3 / \text{time} / 10^9$ (Gflops equiv.)**

### Matrix Dimensions

- **$p = 83$, \( \equiv 1 \bmod / 10000 \) mul.**
- **$p = 821$, \( \equiv 1 \bmod / 100 \) mul.**
- **$p = 1898131$, \( \equiv 1 \bmod / 10000 \) mul.**
- **$p = 18981307$, \( \equiv 1 \bmod / 100 \) mul.**
## Other routines

### LU decomposition

- **Block recursive algorithm** reduces to **MatMul** in $O(n^\omega)$.

<table>
<thead>
<tr>
<th>n</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
<th>15000</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACK-dgetrf</td>
<td>0.024s</td>
<td>2.01s</td>
<td>14.88s</td>
<td>48.78s</td>
<td>113.66s</td>
</tr>
<tr>
<td>fflas-ffpack</td>
<td>0.058s</td>
<td>2.46s</td>
<td>16.08s</td>
<td>47.47s</td>
<td>105.96s</td>
</tr>
</tbody>
</table>

Intel Haswell E3-1270 3.0Ghz using OpenBLAS-0.2.9
Other routines

LU decomposition

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
<th>15000</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACK-dgetrf</td>
<td>0.024s</td>
<td>2.01s</td>
<td>14.88s</td>
<td>48.78s</td>
<td>113.66s</td>
</tr>
<tr>
<td>fflas-ffpack</td>
<td>0.058s</td>
<td>2.46s</td>
<td>16.08s</td>
<td>47.47s</td>
<td>105.96s</td>
</tr>
</tbody>
</table>

Intel Haswell E3-1270 3.0Ghz using OpenBLAS-0.2.9

Characteristic Polynomial

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

<table>
<thead>
<tr>
<th>( n )</th>
<th>1000</th>
<th>2000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>magma-v2.19-9</td>
<td>1.38s</td>
<td>24.28s</td>
<td>332.7s</td>
<td>2497s</td>
</tr>
<tr>
<td>fflas-ffpack</td>
<td>0.532s</td>
<td>2.936s</td>
<td>32.71s</td>
<td>219.2s</td>
</tr>
</tbody>
</table>

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

LU decomposition

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

<table>
<thead>
<tr>
<th>n</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
<th>15000</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACK-dgetrf</td>
<td>0.024s</td>
<td>2.01s</td>
<td>14.88s</td>
<td>48.78s</td>
<td>113.66s</td>
</tr>
<tr>
<td>fflas-ffpack</td>
<td>0.058s</td>
<td>2.46s</td>
<td>16.08s</td>
<td>47.47s</td>
<td>105.96s</td>
</tr>
</tbody>
</table>

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 \( O(n^\omega) \).

<table>
<thead>
<tr>
<th>n</th>
<th>1000</th>
<th>2000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>magma-v2.19-9</td>
<td>1.38s</td>
<td>24.28s</td>
<td>332.7s</td>
<td>2497s</td>
</tr>
<tr>
<td>fflas-ffpack</td>
<td>0.532s</td>
<td>2.936s</td>
<td>32.71s</td>
<td>219.2s</td>
</tr>
</tbody>
</table>

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

\( \times 7.5 \) \( \times 6.7 \)
Outline

1. The LinBox library
2. Blackbox linear algebra
3. Dense linear algebra
4. Parallelization
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
  - $O(n^3)$
  - $\leadsto$ fine grain
  - $\leadsto$ block iterative algorithms
- regular task load
- Numerical stability constraints
Parallelization

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
  - $O(n^3)$
  - $\Rightarrow$ fine grain
  - $\Rightarrow$ block iterative algorithms
- regular task load
- Numerical stability constraints

Exact linear algebra specificities

- cost affected by the splitting
  - $O(n^\omega)$ for $\omega < 3$
  - $\Rightarrow$ modular reductions
  - $\Rightarrow$ coarse grain
  - $\Rightarrow$ recursive algorithms
- rank deficiencies
  - $\Rightarrow$ unbalanced task loads
Parallelization

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**

```c
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

```c
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;
    }
}
```

OMP (since v3)
Task parallelism with fork-join

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

Cilk+

```c
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

```c
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

```c
#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

```c
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

<table>
<thead>
<tr>
<th>// prog model</th>
<th>Architecture</th>
<th>Target app.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OMP 1.0 [97]</strong></td>
<td>Parallel loop</td>
<td>Multi-CPUs</td>
</tr>
<tr>
<td><strong>OMP 3.0 [08]</strong></td>
<td>Fork-join</td>
<td>Multi-CPUs</td>
</tr>
<tr>
<td><strong>OMP 4.0 [14]</strong></td>
<td>Rec. Data Flow</td>
<td>Multi-CPUs</td>
</tr>
<tr>
<td><strong>Cilk [96]</strong></td>
<td>Fork-join</td>
<td>Multi-CPUs</td>
</tr>
<tr>
<td><strong>Athapascan [98]</strong></td>
<td>Rec. Data flow</td>
<td>Clusters+multi-CPU</td>
</tr>
<tr>
<td><strong>TBB [06]</strong></td>
<td>Parallel loop</td>
<td>Multi-CPU</td>
</tr>
<tr>
<td></td>
<td>Fork-join</td>
<td></td>
</tr>
<tr>
<td><strong>Kaapi [06-12]</strong></td>
<td>Rec. Data flow</td>
<td>Multi-CPUs &amp; GPUs</td>
</tr>
<tr>
<td></td>
<td>Parallel loop</td>
<td></td>
</tr>
<tr>
<td><strong>StarSs [07]</strong></td>
<td>Flat data flow</td>
<td>multi-CPUs (SMPSs)</td>
</tr>
<tr>
<td></td>
<td>Flat data flow</td>
<td>multi-CPUs (SMPSs)</td>
</tr>
<tr>
<td></td>
<td>Flat data flow</td>
<td>Cell (CellSs)</td>
</tr>
<tr>
<td></td>
<td>Flat data flow</td>
<td>Grid (GridSs)</td>
</tr>
<tr>
<td><strong>StarPU [09]</strong></td>
<td>Flat data flow</td>
<td>multi-CPUs&amp;GPUs</td>
</tr>
<tr>
<td><strong>Quark [10]</strong></td>
<td>Flat data flow</td>
<td>Multi-CPUs</td>
</tr>
</tbody>
</table>
Illustration: Cholesky factorization

```c
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) {
            clblas_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) {
            clblas_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) {
                clblas_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

```c
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)
      clblas_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)
    clblas_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)
      clblas_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
}
```
Parallelization

C. Pernet (UJF)

The FFLAS-FFPACK and LinBox libraries

LRDE EPITA
SYNC.
Illustration: Cholesky factorization

```c
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[k*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
}
```
Parallelization

C. Pernet (UJF)

The FFLAS-FFPACK and LinBox libraries
Parallelization

The FFLAS-FFPACK and LinBox libraries
Parallelization

C. Pernet (UJF)

The FFLAS-FFPACK and LinBox libraries
Parallelization

C. Pernet (UJF)

The FFLAS-FFPACK and LinBox libraries

LRDE EPITA
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 prototypical
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

Dumas, Gautier, P. and Sultan 14

pfgemm over $\mathbb{Z}/131071\mathbb{Z}$ on a Xeon E5-4620 2.2Ghz 32 cores

MKL dgemm
PLASMA-QUARK dgemm
Parallel matrix multiplication

Dumas, Gautier, P. and Sultan 14

Parallelization

pfgemm over $\mathbb{Z}/131071\mathbb{Z}$ on a Xeon E5-4620 2.2Ghz 32 cores

C. Pernet (UJF)
The FFLAS-FFPACK and LinBox libraries

LRDE EPITA
Parallel matrix multiplication

Dumas, Gautier, P. and Sultan 14

Parallelization

pfgemm over $\mathbb{Z}/131071\mathbb{Z}$ on a Xeon E5-4620 2.2Ghz 32 cores

Gflops

matrix dimension

pfgemm double
pfgemm mod 131071
MKL dgemm
PLASMA-QUARK dgemm

C. Pernet (UJF)
Gaussian elimination

- Slab iterative
  - LAPACK
- Slab recursive
  - FFLAS-FFPACK
- Tile iterative
  - PLASMA
- Tile recursive
  - FFLAS-FFPACK
Gaussian elimination

- Prefer recursive algorithms
Parallelization

Gaussian elimination

- Prefer recursive algorithms
- Better data locality

Tile recursive
FFLAS-FFPACK
Full rank Gaussian elimination

Dumas, Gautier, P. and Sultan 14
Comparing numerical efficiency (no modulo)

parallel PLUQ over double on full rank matrices on 32 cores
Full rank Gaussian elimination

Dumas, Gautier, P. and Sultan 14
Comparing numerical efficiency (no modulo)
Parallelization

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 $\mathbb{Z}/131071\mathbb{Z}$

Parallel PLUQ over $\mathbb{Z}/131071\mathbb{Z}$ with full rank matrices on 32 cores

- Tiled Rec explicit sync
- Tiled Rec dataflow sync
- Tiled Iter dataflow sync
- Tiled Iter explicit sync

Gfops vs. matrix dimension graph
Thank You.