Task based parallelization of recursive linear algebra routines using Kaapi

Clément PERNET
joint work with Jean-Guillaume DUMAS and Ziad SULTAN

Université Grenoble Alpes, LJK-CASYS

January 20, 2017

Journée Runtime, Paris.
Supported by OpenDreamKit Horizon 2020 European Research Infrastructures project (# 676541)
High performance algebraic computing

Domain of computation

\[ \mathbb{Z}, \mathbb{Q} \] : variable size, multi-precision
\[ \mathbb{Z}_p, \text{GF}(p^k) \] : fixed size, specific arithmetic

Common belief: Slow

- terrible complexities,
- no need for *all the precision*

Example (Linear System solving over \( \mathbb{Q} \))

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive Gauss Elim over ( \mathbb{Q} )</td>
<td>( O(2^n) )</td>
</tr>
<tr>
<td>Gauss mod det</td>
<td>( O(n^5) )</td>
</tr>
<tr>
<td>Gauss mod ( p ) + CRT</td>
<td>( O(n^4) ), ( O(n^{\omega+1}) )</td>
</tr>
<tr>
<td>( p )-adic Lifting</td>
<td>( O(n^3) ), ( O^\sim(n^\omega) )</td>
</tr>
</tbody>
</table>

And fast software: LU over \( (\mathbb{Z}/65521\mathbb{Z})^{5000 \times 5000} \) in 3.8s (21.8Gfops on 1 Haswell core)
Gaussian elimination in computer algebra

Applications

Algebraic cryptanalysis: RSA, DLP \( \Rightarrow \) LinSys, Krylov, \( \mathbb{F}_q \)

Comp. number theory: modular forms databases: Echelon over \( \mathbb{F}_q \)

Exact mixed-integer linear programming: \( \Rightarrow \) LinSys over \( \mathbb{Q} \)

Formal proof: Sums of squares \( \Rightarrow \) Cholesky over \( \mathbb{Q} \)

HPC building block

- Dense linear algebra over \( \mathbb{Z}/p\mathbb{Z} \) \( \log_2 p \approx 20 - 30 \) bits
- MatMul (\( \text{fgemm} \)) and GaussElim (\( \text{PLUQ} \))
  - triangular decomposition PLUQ (for LinSys, Det)
  - linear dependencies (Krylov, Grobner basis)
FFLAS-FFPACK library

**FFLAS-FFPACK features**

- High performance implementation of basic linear algebra routines over word size prime fields
- Exact alternative to the numerical BLAS library
- Exact triangularization, Sys. solving, Det, Inv., CharPoly

![Graph showing matrix multiplication performance](image)
Exact vs numerical Gaussian elimination

Similarities

- Reduction to $gemm$ kernel (Matrix Multiplication)
  - Blocking: slab/tiled, iterative/recursive
- Parallel blocking is constrained by pivoting
  - numeric: ensuring numerical stability
  - exact: able to reveal rank profile
Exact vs numerical Gaussian elimination

Similarities

- Reduction to \texttt{gemm} kernel (Matrix Multiplication)
  - Blocking: slab/tiled, iterative/recursive
- Parallel blocking is constrained by pivoting
  - numeric: ensuring numerical stability
  - exact: able to reveal rank profile

Specificities

- Recursive tasks (vs block iterative in numeric)
Exact vs numerical Gaussian elimination

Similarities

- Reduction to `gemm` kernel (Matrix Multiplication)
  - Blocking: slab/tiled, iterative/recursive
- Parallel blocking is constrained by pivoting
  - Numeric: ensuring numerical stability
  - Exact: able to reveal rank profile

Specificities

- Recursive tasks (vs block iterative in numeric)
  - Modular reductions
  - Strassen’s algorithm
    - Efficiency increases with the granularity
    - Tradeoff between total work and fine granularity
Exact vs numerical Gaussian elimination

Similarities

- Reduction to \texttt{gemm} kernel (Matrix Multiplication)
  \Rightarrow \text{Blocking: slab/tiled, iterative/recursive}
- Parallel blocking is constrained by pivoting
  \textit{numeric}: ensuring numerical stability
  \textit{exact}: able to reveal rank profile

Specificities

- Recursive tasks (vs block iterative in numeric)
  \textit{Modular reductions}
  \textit{Strassen’s algorithm} \quad \text{efficiency increases with the granularity}
  \Rightarrow \text{tradeoff between total work and fine granularity}
- Pivoting strategies: no stability constraints, but rank profiles
Exact vs numerical Gaussian elimination

Similarities

- Reduction to `gemm` kernel (Matrix Multiplication)
  - Blocking: slab/tiled, iterative/recursive
- Parallel blocking is constrained by pivoting
  - numeric: ensuring numerical stability
  - exact: able to reveal rank profile

Specificities

- Recursive tasks (vs block iterative in numeric)
- Modular reductions
- Strassen’s algorithm
  - efficiency increases with the granularity
  - tradeoff between total work and fine granularity
- Pivoting strategies: no stability constraints, but rank profiles
- Rank deficiencies:
  - blocks have unpredictable size (and positions)
  - unbalanced task load
Block algorithms

Tiled Iterative

getrf: $A \rightarrow L, U$

Slab Recursive

Tiled Recursive
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

\[ \text{trsm: } B \leftarrow B U^{-1}, B \leftarrow L^{-1} B \]

\[ \text{gemm: } C \leftarrow C - A \times B \]
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

\[ \text{getrf: } A \rightarrow L, U \]
\[ \text{trsm: } B \leftarrow BU^{-1}, B \leftarrow L^{-1}B \]
\[ \text{gemm: } C \leftarrow C - A \times B \]
Block algorithms

getrf: $A \rightarrow L, U$
trsm: $B \leftarrow BU^{-1}, B \leftarrow L^{-1}B$
gemm: $C \leftarrow C - A \times B$
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

$\text{getrf}: A \rightarrow L, U$
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

\[
\text{trsm: } B \leftarrow BU^{-1}, B \leftarrow L^{-1}B
\]

\[
\text{gemm: } C \leftarrow C - A \times B
\]
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

getrf: $A \rightarrow L, U$
Block algorithms

**Tiled Iterative**

**Slab Recursive**

**Tiled Recursive**

**Equations**

\[ \text{trsm: } B \leftarrow B U^{-1}, B \leftarrow L^{-1} B \]

\[ \text{gemm: } C \leftarrow C - A \times B \]
Block algorithms

**Tiled Iterative**

**Slab Recursive**

**Tiled Recursive**

\[
\text{getrf}: A \rightarrow L, U
\]
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

\text{trsm}: \begin{align*}
B & \leftarrow BU^{-1}, \\
B & \leftarrow L^{-1}B
\end{align*}

\text{gemm}: \begin{align*}
C & \leftarrow C - A \times B
\end{align*}
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

getrf: $A \rightarrow L, U$
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

getrf: $A \rightarrow L, U$
**Block algorithms**

- **Tiled Iterative**
- **Slab Recursive**
- **Tiled Recursive**

\[
\text{trsm: } B \leftarrow BU^{-1}, B \leftarrow L^{-1}B \\
\text{gemm: } C \leftarrow C - A \times B
\]
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

\( \text{getrf}: A \rightarrow L, U \)
Block algorithms

- **Tiled Iterative**
- **Slab Recursive**
- **Tiled Recursive**

**trsm:** $B \leftarrow BU^{-1}, B \leftarrow L^{-1}B$

**gemm:** $C \leftarrow C - A \times B$
Block algorithms

getr: $A \rightarrow L, U$

trsm: $B \leftarrow BU^{-1}, B \leftarrow L^{-1}B$

gemm: $C \leftarrow C - A \times B$
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

\texttt{getrf}: A \rightarrow L, U
Need for a high level parallel programming environment

Features required

Portability, Performance and Scalability. But more precisely:

- Runtime system with good performance for recursive tasks.
- Dataflow *task* synchronization

- Handle efficiently unbalanced workloads.
- Efficient range cutting for parallel for.
Need for a high level parallel programming environment

Features required

Portability, Performance and Scalability. But more precisely:

- Runtime system with good performance for recursive tasks.
- Dataflow **task** synchronization

- Handle efficiently unbalanced workloads.
- Efficient range cutting for parallel for.

→ Wish to design a code independently from the runtime system
→ Using runtime systems as a plugin
Outline

1. Runtime systems
2. Matrix Multiplication
3. TRSM
4. Parallel exact Gaussian elimination
Runtime systems to be supported

OpenMP 3.x and 4.0 supported directives: (using libgomp)

- **Data sharing attributes:**
  - OMP3 `shared`: data visible and accessible by all threads
  - OMP3 `firstprivate`: local copy of original value
  - OMP4 `depend`: set data dependencies

- **Synchronization clauses:** `#pragma omp taskwait`

xKaapi: via the libkomp [BDG12] library:

- OpenMP directives → xKaapi tasks.
- Re-implem. of task handling and management.
- Better recursive tasks execution.

TBB: designed for nested and recursive parallelism

- `parallel_for`
- `tbb::task_group`, `wait()`, `run()` using C++11 lambda functions
Parallel Algebraic Linear Algebra Dedicated Interface

Mainly macro-based keywords
- No function call runtime overhead when using macros.
- No important modifications to be done to original program.
- Macros can be used also for C-based libraries.

Complementary C++ template functions
- Implement the different cutting strategies.
- Store the iterators
Task parallelization: **fork-join** and **dataflow** models

- **PAR_BLOCK**: opens a parallel region.
- **SYNCH_GROUP**: Group of tasks synchronized upon exit.
- **TASK**: creates a task.
  - **REFERENCE**(args...): specify variables captured by reference. By default all variables accessed by value.
  - **READ**(args...): set var. that are read only.
  - **WRITE**(args...): set var. that are written only.
  - **READWRITE**(args...): set var. that are read then written.

Example:

```c
void axpy(const Element a, const Element b, Element &y)
{ y += a*x; }
SYNCH_GROUP(
    TASK(MODE(READ(a, x) READWRITE(y)),
    axpy(a, x, y));
); 
```
Parallel matrix multiplication

**Iterative variants**
- **Fixed block size** *(FIXED, GRAIN)*
  - Better control of data mapping in memory
  - Complexity: $O(n^3)$
- **Fixed number of tasks** *(THREADS)*
  - Less control of data mapping in memory
  - Complexity: $O(n^\omega)$

**Recursive variants**
- Almost no control of data mapping in memory
- Complexity: $O(n^\omega)$ or $O(n^3)$
Performance of pfgemm

Figure: Speed of MatMul variants using OpenMP tasks
Performance of pfgemm

pfgemm on 32 cores Xeon E4620 2.2Ghz with TBB

**Figure:** Speed of MatMul variants using IntelTBB tasks
Performance of pfgemm

Figure: Speed of MatMul variants using XKaapi tasks
Parallel Matrix Multiplication: State of the art

HPAC server: 32 cores Xeon E4620 2.2Ghz (4 NUMA sockets)

Comparison of our best implementations with the state of the art numerical libraries:

- MKL dgemm
- OpenBlas dgemm
- PLASMA-QUARK dgemm
- BensonBallard (Strassen)
Parallel Matrix Multiplication: State of the art

Effective Gfops = \frac{\text{# of field ops using classic matrix product}}{\text{time}}.

Comparison of our best implementations with the state of the art numerical libraries:

- WinogradPar\to classicPar\langle double\rangle
- ClassicPar\to WinogradSeq\langle double\rangle
- MKL dgemm
- OpenBlas dgemm
- PLASMA-QUARK dgemm
- BensonBallard (Strassen)
Outline

1. Runtime systems
2. Matrix Multiplication
3. TRSM
4. Parallel exact Gaussian elimination
Parallel Triangular Solving Matrix

Iterative variant:

\[
\begin{bmatrix}
  X_1 & \cdots & X_k
\end{bmatrix}
\leftarrow
L^{-1}
\begin{bmatrix}
  B_1 & \cdots & B_k
\end{bmatrix}.
\]

- The computation of each \(X_i \leftarrow L^{-1}B_i\) is independent
- \(k\) sequential tasks set as the number of available threads

Recursive variant:

1: Split 
\[
\begin{bmatrix}
  X_1 \\
  X_2
\end{bmatrix} = 
\begin{bmatrix}
  L_1 & L_2 \\
  L_2 & L_3
\end{bmatrix}^{-1}
\begin{bmatrix}
  B_1 \\
  B_2
\end{bmatrix}
\]

2: \(X_1 \leftarrow L_1^{-1}B_1\)

3: \(X_2 \leftarrow B_2 - L_2X_1 \ // \ \text{Parallel MatMul}\)

4: \(X_2 \leftarrow L_3^{-1}BX_2\)
Parallel Triangular Solving Matrix

**Iterative variant:**

\[
\begin{bmatrix}
X_1 & \ldots & X_k
\end{bmatrix}
\leftarrow L^{-1}
\begin{bmatrix}
B_1 & \ldots & B_k
\end{bmatrix}.
\]

- The computation of each \(X_i \leftarrow L^{-1}B_i\) is independent
- \(k\) sequential tasks set as the number of available threads

**Recursive variant:**

1: Split

\[
\begin{bmatrix}
X_1 \\
X_2
\end{bmatrix}
= \begin{bmatrix}
L_1 \\
L_2 \\
L_3
\end{bmatrix}^{-1}
\begin{bmatrix}
B_1 \\
B_2
\end{bmatrix}
\]

2: \(X_1 \leftarrow L_1^{-1}B_1\)

3: \(X_2 \leftarrow B_2 - L_2X_1 \quad \text{// Parallel MatMul}\)

4: \(X_2 \leftarrow L_3^{-1}BX_2\)

**Hybrid PFTRSM:** column dimension of \(B\) small

- use iterative splitting in priority
- when \(\#cols(X) < \#proc\): save some threads for recursive calls
Parallel Triangular Solving Matrix Experiments

Figure: Comparing the Iterative and the Hybrid variants for parallel FTRSM using libkomp and libgomp. Only the outer dimension varies: $B$ and $X$ are $10000 \times n$. 
Outline

1. Runtime systems
2. Matrix Multiplication
3. TRSM
4. Parallel exact Gaussian elimination
Reducing to MatMul: block versions

→ Asymptotically faster \( O(n^\omega) \)
→ Better cache efficiency

Variants of block versions

Split on one dimension:
→ Row or Column slab cutting

Split on 2 dimensions:
→ Tile cutting
Gaussian elimination design

Reducing to MatMul: block versions

→ Asymptotically faster ($O(n^\omega)$)
→ Better cache efficiency

Variants of block versions

Iterative:
- Static → better data mapping control
- Dataflow parallel model → less sync

Recursive:
- Adaptive
- sub-cubic complexity
- No Dataflow → more sync
Gaussian elimination design

Reducing to MatMul: block versions

→ Asymptotically faster \( O(n^\omega) \)
→ Better cache efficiency

Variants of block versions

Iterative:
- Static → better data mapping control
- Dataflow parallel model → less sync

Recursive:
- Adaptive
- sub-cubic complexity
- No Dataflow → more sync
Parallel tile recursive PLUQ algorithm

$2 \times 2$ block splitting
Parallel tile recursive PLUQ algorithm

Recursive call
Parallel tile recursive PLUQ algorithm

\[ p_{\text{TRSM}}: B \leftarrow BU^{-1} \]

\[
\text{TASK(MODE(READ(A), READWRITE(B))),}
\]
\[
p_{\text{trsm}}(\ldots, A, \text{lda}, B, \text{ldb});
\]
Parallel tile recursive PLUQ algorithm

\[ p_{\text{TRSM}}: B \leftarrow L^{-1}B \]

\[
\text{TASK(MODE(READ(A) READWRITE(B)),}
\]
\[
pftrsm(\ldots, A, \text{lda}, B, \text{ldb}));
\]
Parallel tile recursive PLUQ algorithm

\[
pf\text{gemm}: C \leftarrow C - A \times B
\]

\[
\text{TASK(MODE(READ(A,B) READWRITE(C))),}
\]

\[
pf\text{gemm}(\ldots, A, \text{lda}, B, \text{ldb});
\]
Parallel tile recursive PLUQ algorithm

\[ \text{pfgemm: } C \leftarrow C - A \times B \]

\[
\text{TASK(MODE(READ(A,B) READWRITE(C)) }, \\
\text{ pfgemm(\ldots, A, lda, B, ldb));}
\]
Parallel tile recursive PLUQ algorithm

\[ \text{pfgemm: } C \leftarrow C - A \times B \]

\[
\text{TASK(MODE(READ(A,B) READWRITE(C)),}
\]
\[
\text{pfgemm(..., A, lda, B, ldb));}
\]
Parallel tile recursive PLUQ algorithm

2 independent recursive calls (concurrent → tasks)

```c
TASK(MODE(READWRITE(A)),
ppluq(..., A, lda));
```
Parallel tile recursive PLUQ algorithm

\[
p_{\text{TRSM}}: \quad B \leftarrow BU^{-1}
\]

\[
\text{TASK(MODE(READ(A) READWRITE(B)),}
\]
\[
p_{\text{ftrsm}}(\ldots, A, \text{lda}, B, \text{ldb});
\]
Parallel tile recursive PLUQ algorithm

\[ p_{\text{TRSM}}: B \leftarrow L^{-1}B \]

\[
\text{TASK(MODE(READ(A) READWRITE(B))}, \\
\text{pftrsm(..., A, lda, B, ldb));}
\]
Parallel tile recursive PLUQ algorithm

\[ \text{pfgemm: } C \leftarrow C - A \times B \]

\[ \text{TASK(MODE(READ(A,B) READWRITE(C)),} \]
\[ \text{pfgemm(..., A, lda, B, ldb))}; \]
Parallel tile recursive PLUQ algorithm

\[ \text{pfgemm: } C \leftarrow C - A \times B \]

\[
\text{TASK(MODE(READ(A,B) \text{ READWRITE(C))},}
\]
\[
\text{ pfgemm(\ldots, A, lda, B, ldb ));}
\]
Parallel tile recursive PLUQ algorithm

\[ \text{pfgemm}: C \leftarrow C - A \times B \]

\[
\text{TASK(MODE(READ(A,B) READWRITE(C)),}
\text{ pfgemm(..., A, lda, B, ldb));}
\]
Parallel tile recursive PLUQ algorithm

Recursive call
Parallel tile recursive PLUQ algorithm

Puzzle game (block permutations)
Tile rec: better data locality and more square blocks for M.M.
State of the art: exact vs numerical linear algebra

State of the art comparison:
- Exact PLUQ using PALADIn language: best performance with xKaapi
- Numerical LU (dgetrf) of PLASMA-Quark and MKL dgetrf

parallel dgetrf vs parallel PLUQ on full rank matrices

- explicit synch pluq<double>
- MKL dgetrf
- PLASMA-Quark dgetrf tiled storage (k=212)
- PLASMA-Quark dgetrf (k=212)
Performance of parallel PLUQ decomposition

Low impact of modular reductions in parallel → Efficient SIMD implementation

Performance of tile PLUQ recursive vs iterative on full rank matrices

![Graph showing performance comparison](image)

- **explicit synch pluq rec<double>**
- **explicit synch pluq rec<131071>**

**Effective Gflops** vs **Matrix Dimension**
Performance of task parallelism: dataflow model

Performance of tile PLUQ recursive vs iterative on full rank matrices
Performance of task parallelism: dataflow model

Possible improvement: implementation of the delegation of recursive tasks dependencies

(Postpone access mode in the parallel programming environments)
Outline

1. Runtime systems
2. Matrix Multiplication
3. TRSM
4. Parallel exact Gaussian elimination
Conclusion

Lessons learnt for the parallelization of LU over $\mathbb{Z}/p\mathbb{Z}$

- Blocking impacts arithmetic cost $\Rightarrow$ fine granularity hurts
- Rank deficiency can offer more parallelism (cf. separators)
- sub-cubic perfs in parallel
- requires a runtime efficient for recursive tasks (XKaapi)
Perspectives

- already at use in tiled iterative algorithms (XKaaip)
- new challenges for recursive tasks:
  - Recursive inclusion of sub-matrices
  - Postponed modes (removing fake dependencies)
- Distributed on small sized clusters
Thank you