Cholesky(QR) factorization on multicore architectures MIMD

Tuguluke Abulitiibu
University of Colorado Denver
November 4, 2014
Motivating Example

Cholesky

QR

LAPACK

My goal
**Motivating Example**

System of linear equations

We want to approximate solutions to

\[ \overline{A} \overline{x} = \overline{b} \]

where

\[
\begin{pmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m1} & a_{m2} & \cdots & a_{mn}
\end{pmatrix}
\]

\[
\begin{pmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n
\end{pmatrix},
\]

\[
\begin{pmatrix}
  b_1 \\
  b_2 \\
  \vdots \\
  b_m
\end{pmatrix}
\]

This equation is used in almost all numerical solution of Partial Differential Equations. FDM, FEM, FVM, etc.
Solving the System

How to get $x$?

- Gaussian elimination + Back substitution
- Gaussian elimination + Pivoting + Back substitution

This would give us a good result if $A$ has a good condition number.

What if $A$ has a bad condition number?

- Preconditioning (Not today’s topic).
- Cholesky factorization
- QR factorization

The maximum value of condition number is the product of the two operator norms:

$$k(A) = \|A^{-1}\| \|A\|$$
In linear algebra, the Cholesky factorization is a decomposition of a Hermitian (Symmetric), positive-definite (Eigenvalue is positive) matrix into the product of a lower triangular matrix and its conjugate transpose.

\[ A = LL^* \]

to solve \( Ax=b \):

- decompose \( A \) into \( LU \).
- solve \( Ly = b \) for \( y \) by forward substitution.
- solve \( L^*x = y \) for \( x \) by back substitution.
In linear algebra, QR factorization of a matrix is a decomposition of a matrix into a product of an orthogonal \((Q^T Q = I \rightarrow Q^{-1} = Q^T)\) matrix \(Q\) and an upper triangular matrix \(R\)[5].

\[ A = QR \]

to solve \(Ax=b\):
- Decompose \(A\) into \(QR\).
- solve \(Qy = b\) for \(y\) by forward substitution.
- Solve \(x = R^{-1}y\) for \(x\) by back substitution.

For example, if \(A\) is full column rank, then \(R\) is invertible, so that the solution is unique, and given by \(x = R^{-1}Q^T y\).


\[ A = LDL^T = \begin{pmatrix}
1 & 0 & 0 \\
L_{21} & 1 & 0 \\
L_{31} & L_{32} & 1
\end{pmatrix}
\begin{pmatrix}
D_1 & 0 & 0 \\
0 & D_2 & 0 \\
0 & 0 & D_3
\end{pmatrix}
\begin{pmatrix}
1 & L_{21} & L_{31} \\
0 & 1 & L_{32} \\
0 & 0 & 1
\end{pmatrix}
\]

\[ = \begin{pmatrix}
D_1 \\
L_{21}D_1 & L_{21}^2D_1 + D_2 \\
L_{31}D_1 & L_{31}L_{21}D_1 + L_{32}D_2 & L_{31}^2D_1 + L_{32}^2D_2 + D_3
\end{pmatrix}
\]

(symmetric)
**Cholesky decomposition using formula**

**Real matrix (Cholesky - Banachiewicz)**

\[
L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2}.
\]

\[
L_{i,j} = \frac{1}{L_{j,j}} \left( A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} \right), \quad \text{for } i > j.
\]

**Complex matrix (Cholesky - Crout)**

\[
L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k} L_{j,k}^*}.
\]

\[
L_{i,j} = \frac{1}{L_{j,j}} \left( A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k}^* \right), \quad \text{for } i > j.
\]
% Cholesky Factorization for symmetric positive definite matrix
% Factorize A such that A = L*L',
% where L is a lower triangular matrix whose diagonal entries are not
function A=Cholesky(A)

n=size(A);

for k=1:n
    A(k,k)=sqrt(A(k,k));
    A(k+1:n,k)=A(k+1:n,k)/A(k,k);

    for j=k+1:n
        A(j:n,j)=A(j:n,j)-A(j,k)*A(j:n,k);
    end
end

Matlab style code for Sequential Cholesky decomposition
The Gram-schmidt process

\[ \mathbf{u}_1 = \mathbf{v}_1, \]
\[ \mathbf{u}_2 = \mathbf{v}_2 - \text{proj}_{\mathbf{u}_1}(\mathbf{v}_2), \]
\[ \mathbf{u}_3 = \mathbf{v}_3 - \text{proj}_{\mathbf{u}_1}(\mathbf{v}_3) - \text{proj}_{\mathbf{u}_2}(\mathbf{v}_3), \]
\[ \mathbf{u}_4 = \mathbf{v}_4 - \text{proj}_{\mathbf{u}_1}(\mathbf{v}_4) - \text{proj}_{\mathbf{u}_2}(\mathbf{v}_4) - \text{proj}_{\mathbf{u}_3}(\mathbf{v}_4), \]
\[ \vdots \]
\[ \mathbf{u}_k = \mathbf{v}_k - \sum_{j=1}^{k-1} \text{proj}_{\mathbf{u}_j}(\mathbf{v}_k), \]
\[ \mathbf{e}_1 = \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|} \quad (1) \]
\[ \mathbf{e}_2 = \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|} \quad (2) \]
\[ \mathbf{e}_3 = \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|} \quad (3) \]
\[ \mathbf{e}_4 = \frac{\mathbf{u}_4}{\|\mathbf{u}_4\|} \quad (4) \]
\[ \vdots \quad (5) \]
\[ \mathbf{e}_k = \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}. \quad (6) \]
% Input: V is an m by n matrix of full rank m<=n
% Output: an m-by-n upper triangular matrix R
% and an m-by-m unitary matrix Q so that A = Q*R.

function [Q,R]=gschmidt(V)

[m,n]=size(V);
R=zeros(n);
R(1,1)=norm(V(:,1));
Q(:,1)=V(:,1)/R(1,1);

for k=2:n
    R(1:k-1,k)=Q(:,1:k-1)'*V(:,k);
    Q(:,k)=V(:,k)-Q(:,1:k-1)*R(1:k-1,k);
    R(k,k)=norm(Q(:,k));
    Q(:,k)=Q(:,k)/R(k,k);
end

Matlab style code for Sequential QR factorization
Matlab

- Command for $x = A^{-1}b : A \backslash b$
- Command for Cholesky decomposition : $R = \text{chol}(A)$
- Command for QR factorization : $[Q,R] = \text{qr}(A)$

But, what is really happening behind these function?

- Matlab
- LAPACK (Linear Algebra Package)
- BLAS (Basic Linear Algebra Subprograms)

What is the 'magic' trick?
What is LAPACK?

LAPACK (Linear Algebra Package) is a standard software library for numerical linear algebra[4]. It provides routines for solving systems of linear equations and linear least squares, eigenvalue problems, and singular value decomposition. It also has routines to implement matrix factorizations such as LU, QR, Cholesky.

LAPACK was originally written in FORTRAN 77 (later in FORTRAN 90), it can handle both real and complex matrices in both single and double precision. It can effectively exploit the caches on modern cache-based architectures, by given a well-tuned BLAS implementation. In its later packages such as ScaLAPACK and PLAPACK, it has been extended to run on distributed-memory systems.
**Functions/Subroutines**

### manual

- **S, D** stand for real floating point arithmetic respectively in single and double precision.
- **C** and **Z** stand for complex arithmetic with respectively single and double precision.
- **GE** expects an $n \times n$ array containing the entries of the matrix.
- **MM** is a two-letter code denoting the kind of matrix expected by the algorithm.

So, e.g., DGEMM means real floating point in double precision, containing the entries of the matrix algorithm.

DGEMM performs one of the matrix-matrix operations
\[ C := \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C, \]
where \( \text{op}(X) \) is one of
\[ \text{op}(X) = X \text{ or } \text{op}(X) = X^{**T}, \]
alpha and beta are scalars, and A, B and C are matrices, with \( \text{op}(A) \) an m by k matrix, \( \text{op}(B) \) a k by n matrix and C an m by n matrix.
The expression is:

\[
A \times B = \begin{bmatrix}
A_{11} & A_{12} & \cdots & A_{1s} \\
A_{21} & A_{22} & \cdots & A_{2s} \\
\vdots & \vdots & \ddots & \vdots \\
A_{q1} & A_{q2} & \cdots & A_{qs}
\end{bmatrix}
\begin{bmatrix}
B_{11} & B_{12} & \cdots & B_{1r} \\
B_{21} & B_{22} & \cdots & B_{2r} \\
\vdots & \vdots & \ddots & \vdots \\
B_{s1} & B_{s2} & \cdots & B_{sr}
\end{bmatrix},
\]

Where

\[
A_{11} = \begin{bmatrix}
A_{(1,1)} & A_{(1,2)} & \cdots & A_{(1,n-1)} & A_{(1,n)} \\
A_{(2,1)} & A_{(1,1)} & A_{(1,2)} & \cdots & \cdots & \cdots \\
\vdots & \vdots & \ddots & \ddots & \ddots & \ddots \\
A_{(n-1,1)} & A_{(1,1)} & A_{(1,2)} & \cdots & \cdots & \cdots \\
A_{(n,1)} & A_{(n-1,1)} & \cdots & A_{(2,1)} & A_{(1,1)} & A_{(1,2)}
\end{bmatrix}.
\]
### Matrix Partition

<table>
<thead>
<tr>
<th></th>
<th>A11</th>
<th>A12</th>
<th>A13</th>
<th>A14</th>
</tr>
</thead>
<tbody>
<tr>
<td>A21</td>
<td>A22</td>
<td>A23</td>
<td>A24</td>
<td></td>
</tr>
<tr>
<td>A31</td>
<td>A32</td>
<td>A33</td>
<td>A34</td>
<td></td>
</tr>
<tr>
<td>A41</td>
<td>A42</td>
<td>A43</td>
<td>A44</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B11</th>
<th>B12</th>
<th>B13</th>
<th>B14</th>
</tr>
</thead>
<tbody>
<tr>
<td>B21</td>
<td>B22</td>
<td>B23</td>
<td>B24</td>
<td></td>
</tr>
<tr>
<td>B31</td>
<td>B32</td>
<td>B33</td>
<td>B34</td>
<td></td>
</tr>
<tr>
<td>B41</td>
<td>B42</td>
<td>B43</td>
<td>B44</td>
<td></td>
</tr>
</tbody>
</table>
Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software, LAPACK for example.[4]

What is BLAS?
The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations.

- The Level 1 BLAS perform scalar, vector and vector-vector operations.
- The Level 2 BLAS perform matrix-vector operations.
- The Level 3 BLAS perform matrix-matrix operations.

First published as a Fortran library in 1979

More will be study in the next presentation.
Dynamically scheduled Cholesky factorization

Functions

- **DPOTRF**: which performs the Cholesky factorization of a diagonal tile.
- **DTRSM**: which performs a triangular solve.
- **DSYRK**: which executes a symmetric rank-k update.
- **DGEMM**: which applies a matrix multiplication

```
For k=0..TILES-1 A[k][k] := DPOTRF(A[k][k])

For m=k+1..TILES-1 A[m][k] := DTRSM(A[k][k],A[m][k])

For n=k+1..TILES-1 A[n][n] := DSYRK(A[n][k],A[n][n])

For m=n+1..TILES-1 A[m][n] := DGEMM(A[m][k],A[n][k],A[m][n])
```
TASK GRAPH OF TILE CHOLESKY FACTORIZATION
Wish me Luck
Colibri

- This cluster is funded by NSF grant Julien Langou, Jack Dongarra, Jan Mandel, Daniel Connors, Jonathan Beezley, GPU Cluster for Computing Research.
- The cluster consists of 24 nodes each equipped with two CPUs and two NVIDIA Fermi GPUs, connected by Infiniband.
Head Node

Node 1: 16 cores

Node 2: 16 cores

Node 3: 16 cores

Node 4: 16 cores

Node 24: 16 cores

Figure: A 16x16 element block matrix with 1212, 1224, 24x12, and 2424 sub-Matrices.
How to make is faster?

- parallelization is ‘good’.
- communication between nodes is ‘bad’.

- QR factorization algorithms that are optimized to avoid communication.
- Communication includes both messages between processors and data movement between memory.

here should be an animation for Communication-avoiding parallel QR factorizations.
16 Cores Shared Memory. GPU in Distributed Memory.
Development language

- C.
- MPI.

Message Passing interface

Message Passing Interface (MPI) is a standardized and portable message-passing system. It has become a de facto standard for communication among processes that model a parallel program running on a distributed memory system[3].
GENERAL MPI PROGRAM STRUCTURE. FROM HTTPS://COMPUTING.LLNL.GOV/TUTORIALS/MPI/

MPI include file

*Declarations, prototypes, etc.*

Program Begins

*Serial code*

Initialize MPI environment

*Parallel code begins*

Do work & make message passing calls

Terminate MPI environment

*Parallel code ends*

*Serial code*

Program Ends
/* C Example */
#include <stdio.h>
#include <mpi.h>

int main (argc, argv)
    int argc;
    char *argv[];
{
    int rank, size;

    MPI_Init (&argc, &argv); /* starts MPI */
    MPI_Comm_rank (MPI_COMM_WORLD, &rank); /* get current process id */
    MPI_Comm_size (MPI_COMM_WORLD, &size); /* get number of processes */
    printf( "Hello world from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
- Get numerical results and comparison.
- Add one section dedicated to BLAS.
- Better Graphic explanation with animation.
- Do more depth study on MPI.
- Combination of Cholesky and QR. (If time allowed)
Harry F. Jordan, Gita Alaghband; *Fundamentals of Parallel Processing*.

Emmanuel Agullo, Cedric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Jean Roman, Samuel Thibault, Stanimire Tomov; *Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators*.


Leslie Hogben; *Handbook of Linear Algebra (Discrete Mathematics and Its Applications)*.

Lloyd N. Trefethen, David Bau III; *Numerical Linear Algebra*.

David S. Watkins; *Fundamentals of Matrix Computations*.
Questions?
Thank you!