# Solving Triangular Systems

1. Forward Substitution Formulas
   - processing the L of the LU factorization
   - a third type of pipeline
2. Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic
3. Parallel Solving
   - using an *n*-stage pipeline
   - rewriting the formulas
   - a parallel solver with OpenMP

MCS 572 Lecture 29
Introduction to Supercomputing
Jan Verschelde, 19 March 2021

# Solving Triangular Systems

1. Forward Substitution Formulas
   - processing the L of the LU factorization
   - a third type of pipeline

2. Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic

3. Parallel Solving
   - using an $n$-stage pipeline
   - rewriting the formulas
   - a parallel solver with OpenMP

# LU factorization

To solve an $n$-dimensional linear system $A\mathbf{x} = \mathbf{b}$
we factor $A$ as a product of two triangular matrices, $A = LU$:

- $L$ is lower triangular, $L = [\ell_{i,j}]$, $\ell_{i,j} = 0$ if $j > i$ and $\ell_{i,i} = 1$.
- $U$ is upper triangular $U = [u_{i,j}]$, $u_{i,j} = 0$ if $i > j$.

Solving $A\mathbf{x} = \mathbf{b}$ is equivalent to solving $L(U\mathbf{x}) = \mathbf{b}$:

1. Forward substitution: $L\mathbf{y} = \mathbf{b}$.
2. Backward substitution: $U\mathbf{x} = \mathbf{y}$.

Factoring $A$ costs $O(n^3)$, solving triangular systems costs $O(n^2)$.

# formulas for forward substitution

Expanding the matrix-vector product $L\mathbf{y}$ in $L\mathbf{y} = \mathbf{b}$ leads to

$$\left\{ \begin{array}{lcl} y_1 & = & b_1 \\ \ell_{2,1}y_1 + y_2 & = & b_2 \\ \ell_{3,1}y_1 + \ell_{3,2}y_2 + y_3 & = & b_3 \\ & \vdots & \\ \ell_{n,1}y_1 + \ell_{n,2}y_2 + \ell_{n,3}y_3 + \cdots + \ell_{n,n-1}y_{n-1} + y_n & = & b_n \end{array} \right.$$

and solving for the diagonal elements gives

$$\begin{array}{lcl} y_1 & = & b_1 \\ y_2 & = & b_2 - \ell_{2,1}y_1 \\ y_3 & = & b_3 - \ell_{3,1}y_1 - \ell_{3,2}y_2 \\ & \vdots & \\ y_n & = & b_n - \ell_{n,1}y_1 - \ell_{n,2}y_2 - \cdots - \ell_{n,n-1}y_{n-1} \end{array}$$

# formula and algorithm

For $k = 1, 2, \ldots, n$:

$$y_k = b_k - \sum_{i=1}^{k-1} \ell_{k,i} y_i.$$

As an algorithm:

for $k$ from 1 to $n$ do
  $y_k := b_k$;
  for $i$ from 1 to $k-1$ do
    $y_k := y_k - \ell_{k,i} \star y_i$.

We count

$$1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$$

multiplications and subtractions.

# Solving Triangular Systems

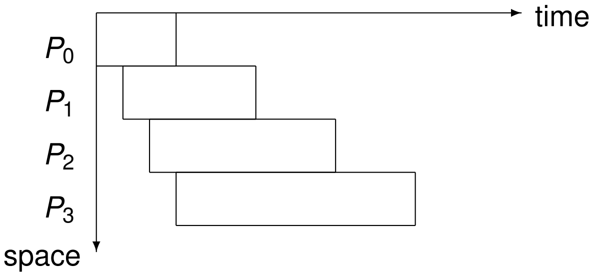
1. Forward Substitution Formulas
   - processing the L of the LU factorization
   - a third type of pipeline
2. Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic
3. Parallel Solving
   - using an $n$-stage pipeline
   - rewriting the formulas
   - a parallel solver with OpenMP

# a third type of pipeline

Three types of pipelines:

1. Speedup only if multiple instances. Example: instruction pipeline.
2. Speedup already if one instance. Example: pipeline sorting.
3. Worker continues after passing information through.
   Example: solve $L\mathbf{y} = \mathbf{b}$.

Typical for the 3rd type of pipeline is the varying length of each job.



# Solving Triangular Systems

1. Forward Substitution Formulas
   - processing the L of the LU factorization
   - a third type of pipeline
2. Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic
3. Parallel Solving
   - using an $n$-stage pipeline
   - rewriting the formulas
   - a parallel solver with OpenMP

# ill conditioned matrices

Consider the 4-by-4 lower triangular matrix

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -2 & -2 & 1 & 0 \\ -2 & -2 & -2 & 1 \end{bmatrix}.$$

What we know from numerical analysis:

1. The condition number of a matrix magnifies roundoff errors.
2. The hardware double precision is $2^{-52} \approx 2.2 \times 10^{-16}$.
3. We get no accuracy from condition numbers larger than $10^{16}$.

# an experiment in an interactive Julia session

```
julia> using LinearAlgebra

julia> A = ones(32,32);

julia> D = Diagonal(A);

julia> L = LowerTriangular(A);

julia> LmD = L - D;

julia> L2 = D - 2*LmD;

julia> cond(L2)
2.4163163056907e16
```

The condition number is estimated at $2.4 \times 10^{16}$.

# Solving Triangular Systems

1. Forward Substitution Formulas
   - processing the L of the LU factorization
   - a third type of pipeline

2. Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic

3. Parallel Solving
   - using an $n$-stage pipeline
   - rewriting the formulas
   - a parallel solver with OpenMP

# quad double arithmetic

A quad double is an unevaluated sum of 4 doubles,
improves working precision from $2.2 \times 10^{-16}$ to $2.4 \times 10^{-63}$.

Y. Hida, X.S. Li, and D.H. Bailey: **Algorithms for quad-double precision floating point arithmetic.** In *15th IEEE Symposium on Computer Arithmetic* pages 155–162. IEEE, 2001. Software at
`http://crd.lbl.gov/~dhbailey/mpdist/qd-2.3.17.tar.gz`.

A quad double builds on `double double`, some features:

- The least significant part of a `double double` can be interpreted as a compensation for the roundoff error.
- Predictable overhead: working with `double double` is of the same cost as working with complex numbers.

# operator overloading in C++

```
#include <iostream>
#include <iomanip>
#include <qd/qd_real.h>
using namespace std;

int main ( void )
{
   qd_real q("2");
   cout << setprecision(64) << q << endl;
   for(int i=0; i<8; i++)
   {
      qd_real dq = (q*q - 2.0)/(2.0*q);
      q = q - dq; cout << q << endl;
   }
   cout << scientific << setprecision(4);
   cout << "residual : " << q*q - 2.0 << endl;
   return 0;
}
```

# compiling with a makefile

The makefile contains the entry:

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib

qd4sqrt2:
        g++ -I$(QD_ROOT)/include qd4sqrt2.cpp \
            $(QD_LIB)/libqd.a \
            -o qd4sqrt2
```

Compiling at the command prompt `$`:

```
$ make qd4sqrt2
g++ -I/usr/local/qd-2.3.17/include qd4sqrt2.cpp \
            /usr/local/lib/libqd.a \
            -o qd4sqrt2
```

# running the code

```
$ ./qd4sqrt2
2.0000000000000000000000000000000000000000000000000000000000000000e+00
1.5000000000000000000000000000000000000000000000000000000000000000e+00
1.4166666666666666666666666666666666666666666666666666666666666667e+00
1.4142156862745098039215686274509803921568627450980392156862745098e+00
1.4142135623746899106262955788901349101165596221157440445849050192e+00
1.4142135623730950488016896235025302436149819257761974284982894987e+00
1.4142135623730950488016887242096980785696718753772340015610131332e+00
1.4142135623730950488016887242096980785696718753769480731766797380e+00
1.4142135623730950488016887242096980785696718753769480731766797380e+00
residual : 0.0000e+00
$
```

General multiple double arithmetic is available:

M. Joldes, J.-M. Muller, V. Popescu, W. Tucker.
**CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications.**
In *Mathematical Software – ICMS 2016, the 5th International Conference on Mathematical Software*, pages 232–240, Springer-Verlag, 2016.

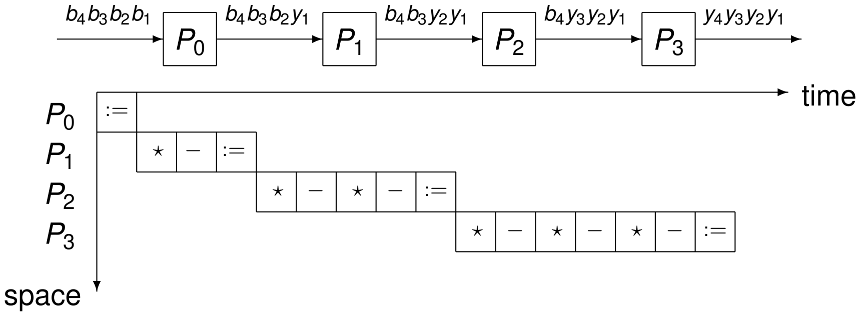
# Solving Triangular Systems

1. Forward Substitution Formulas
   - processing the L of the LU factorization
   - a third type of pipeline
2. Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic
3. Parallel Solving
   - using an $n$-stage pipeline
   - rewriting the formulas
   - a parallel solver with OpenMP

# using an $n$-stage pipeline

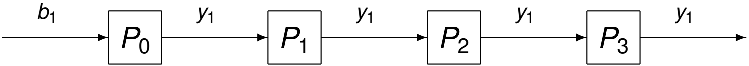
We assume that $L$ is available on every processor.

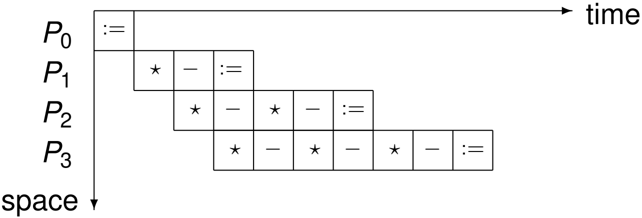
$$\text{for } n = 4 = p: \quad \begin{aligned} y_1 &:= b_1 \\ y_2 &:= b_2 - \ell_{2,1} \star y_1 \\ y_3 &:= b_3 - \ell_{3,1} \star y_1 - \ell_{3,2} \star y_2 \\ y_4 &:= b_4 - \ell_{4,1} \star y_1 - \ell_{4,2} \star y_2 - \ell_{4,3} \star y_3 \end{aligned}$$

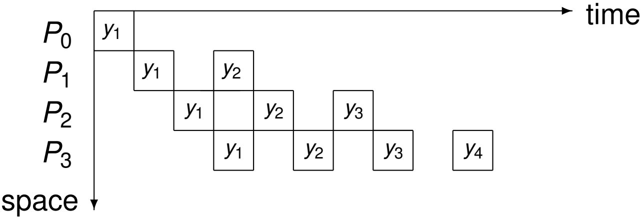


# type 3 pipelining

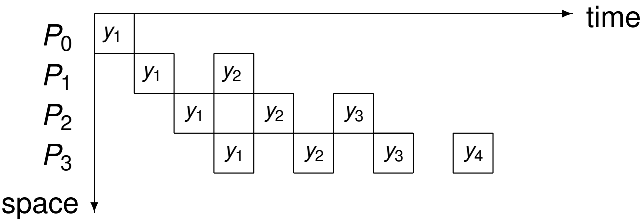
Make $y_1$ available in the next pipeline cycle:



# counting the steps




We count the steps for $p = 4$ or in general, for $p = n$:

1. The latency takes 4 steps for $y_1$ to be at $P_4$, or in general: $n$ steps for $y_1$ to be at $P_n$.
2. It takes then 6 additional steps for $y_4$ to be computed by $P_4$, or in general: $2n - 2$ additional steps for $y_n$ to be computed by $P_n$.

So it takes $n + 2n - 2 = 3n - 2$ steps to solve
an $n$-dimensional triangular system by an $n$-stage pipeline.

# Solving Triangular Systems

1. Forward Substitution Formulas
   - processing the L of the LU factorization
   - a third type of pipeline

2. Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic

3. Parallel Solving
   - using an $n$-stage pipeline
   - rewriting the formulas
   - a parallel solver with OpenMP

# rewriting the formulas

Solving $L\mathbf{y} = \mathbf{b}$ for $n = 5$:

1. $y := \mathbf{b}$;
2. $y_2 := y_2 - \ell_{2,1} \star y_1$;
   $y_3 := y_3 - \ell_{3,1} \star y_1$;
   $y_4 := y_4 - \ell_{4,1} \star y_1$;
   $y_5 := y_5 - \ell_{5,1} \star y_1$;
3. $y_3 := y_3 - \ell_{3,2} \star y_2$;
   $y_4 := y_4 - \ell_{4,2} \star y_2$;
   $y_5 := y_5 - \ell_{5,2} \star y_2$;
4. $y_4 := y_4 - \ell_{4,3} \star y_3$;
   $y_5 := y_5 - \ell_{5,3} \star y_3$;
5. $y_5 := y_5 - \ell_{5,4} \star y_4$;

$\mathbf{y} := \mathbf{b}$;
for $i$ from 2 to $n$ do
  for $j$ from $i$ to $n$ do
    $y_j := y_j - \ell_{j,i-1} \star y_{i-1}$;

$\Rightarrow$ all instructions in the $j$ loop are independent from each other!

# data parallelism

Consider the inner loop in the algorithm to solve $L\mathbf{y} = \mathbf{b}$:

$$
\begin{array}{l}
\mathbf{y} := \mathbf{b};\\
\text{for } i \text{ from 2 to } n \text{ do}\\
\quad \boxed{\begin{array}{l}\text{for } j \text{ from } i \text{ to } n \text{ do}\\ \quad y_j := y_j - \ell_{j,i-1} \star y_{i-1};\end{array}}
\end{array}
$$

We distribute the update of $y_i, y_{i+1}, \ldots, y_n$ among $p$ processors.

If $n \gg p$, then we expect a close to optimal speedup.

# Solving Triangular Systems

1. Forward Substitution Formulas
   - processing the L of the LU factorization
   - a third type of pipeline
2. Multiple Double Arithmetic
   - ill conditioned matrices
   - quad double arithmetic
3. Parallel Solving
   - using an $n$-stage pipeline
   - rewriting the formulas
   - a parallel solver with OpenMP

# a parallel solver

For our parallel solver for triangular systems:

- For $L = [\ell_{i,j}]$, we generate random numbers for $\ell_{i,j} \in [0, 1]$.
  The exact solution $\mathbf{y}$: $y_i = 1$, for $i = 1, 2, \ldots, n$.
  We compute the right hand side $\mathbf{b} = L\mathbf{y}$.

- Even already in small dimensions,
  the condition number may grow exponentially.

  Hardware double precision is insufficient.

  Therefore, we use quad double arithmetic.

- We use a straightforward OpenMP implementation.

# solving random lower triangular systems

Relying on hardware doubles is problematic:

```
$ time ./trisol 10
last number : 1.0000000000000009e+00

real    0m0.003s    user    0m0.001s    sys    0m0.002s

$ time ./trisol 100
last number : 9.9999999999974221e-01

real    0m0.005s    user    0m0.001s    sys    0m0.002s

$ time ./trisol 1000
last number : 2.7244600009080568e+04

real    0m0.036s    user    0m0.025s    sys    0m0.009s
```

# a matrix of quad doubles

Allocating data in the main program:

```
{
   qd_real b[n],y[n];

   qd_real **L;
   L = (qd_real**) calloc(n,sizeof(qd_real*));
   for(int i=0; i<n; i++)
      L[i] = (qd_real*) calloc(n,sizeof(qd_real));

   srand(time(NULL));
   random_triangular_system(n,L,b);
```

# a random triangular system

```
void random_triangular_system
 ( int n, qd_real **L, qd_real *b )
{
   for(int i=0; i<n; i++)
   {
      L[i][i] = 1.0;
      for(j=0; j<i; j++)
      {
         double r = ((double) rand())/RAND_MAX;
         L[i][j] = qd_real(r);
      }
      for(int j=i+1; j<n; j++)
         L[i][j] = qd_real(0.0);
   }
   for(int i=0; i<n; i++)
   {
      b[i] = qd_real(0.0);
      for(int j=0; j<n; j++)
         b[i] = b[i] + L[i][j];
   }
}
```

# solving the system

```
void solve_triangular_system_swapped
 ( int n, qd_real **L, qd_real *b, qd_real *y )
{
   for(int i=0; i<n; i++) y[i] = b[i];

   for(int i=1; i<n; i++)
   {
      for(int j=i; j<n; j++)
         y[j] = y[j] - L[j][i-1]*y[i-1];
   }
}
```

# using OpenMP

```
void solve_triangular_system_swapped
 ( int n, qd_real **L, qd_real *b, qd_real *y )
{
   int j;

   for(int i=0; i<n; i++) y[i] = b[i];

   for(int i=1; i<n; i++)
   {
      #pragma omp parallel shared(L,y) private(j)
      {
         #pragma omp for
         for(j=i; j<n; j++)
            y[j] = y[j] - L[j][i-1]*y[i-1];
      }
   }
}
```

# experimental timings

running `time ./trisol_qd_omp n p`
On dimension $n = 8,000$ for varying number $p$ of cores.

| $p$ | cpu time | real | user | sys |
|---|---|---|---|---|
| 1 | 21.240s | 35.095s | 34.493s | 0.597s |
| 2 | 22.790s | 25.237s | 36.001s | 0.620s |
| 4 | 22.330s | 19.433s | 35.539s | 0.633s |
| 8 | 23.200s | 16.726s | 36.398s | 0.611s |
| 12 | 23.260s | 15.781s | 36.457s | 0.626s |

The serial part is the generation of the random numbers for $L$ and the computation of $\mathbf{b} = L\mathbf{y}$. Recall Amdahl's Law.

We can compute the serial time, subtracting for $p = 1$, from the real time the cpu time spent in the solver, i.e.: $35.095 - 21.240 = 13.855$.
For $p = 12$, time spent on the solver is $15.781 - 13.855 = 1.926$.
Compare 1.926 to 21.240/12 = 1.770.

# Summary + Exercises

We ended chapter 5 in the book of Wilkinson and Allen.

Exercises:

1. Consider the upper triangular system $U\mathbf{x} = \mathbf{y}$, with $U = [u_{i,j}]$, $u_{i,j} = 0$ if $i > j$. Derive the formulas and general algorithm to compute the components of the solution $\mathbf{x}$.
   For $n = 4$, draw the third type of pipeline.
2. Write a parallel solver with OpenMP to solve $U\mathbf{x} = \mathbf{y}$.
   Take for $U$ a matrix with random numbers in $[0, 1]$, compute $\mathbf{y}$ so all components of $\mathbf{x}$ equal one. Test the speedup of your program, for large enough values of $n$ and a varying number of cores.
3. Describe a parallel solver for upper triangular systems $U\mathbf{y} = \mathbf{b}$ for distributed memory computers. Write a prototype implementation using MPI and discuss its scalability.