Solving Triangular Systems

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

2. Parallel Solving
   - using an \( n \)-stage pipeline
   - rewriting the formulas
   - a parallel solver with OpenMP
Solving Triangular Systems

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

2. **Parallel Solving**
   - using an $n$-stage pipeline
   - rewriting the formulas
   - a parallel solver with OpenMP
LU factorization

To solve an $n$-dimensional linear system $Ax = b$ 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 $Ax = b$ is equivalent to solving $L(Ux) = b$:

1. Forward substitution: $Ly = b$.
2. Backward substitution: $Ux = y$.

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

Expanding the matrix-vector product $Ly$ in $Ly = b$ leads to

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

and solving for the diagonal elements gives

$$
\begin{align*}
    y_1 & = b_1 \\
    y_2 & = b_2 - \ell_{2,1} y_1 \\
    y_3 & = b_3 - \ell_{3,1} y_1 - \ell_{3,2} y_2 \\
    \vdots \\
    y_n & = b_n - \ell_{n,1} y_1 - \ell_{n,2} y_2 - \cdots - \ell_{n,n-1} y_{n-1}
\end{align*}
$$
For $k = 1, 2, \ldots, n$:

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

As an algorithm:

```python
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} \times 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. 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: adding numbers.
2. Speedup already if one instance. Example: pipeline sorting.
3. Worker continues after passing information through.
   Example: solve $Ly = b$.

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. 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.

For \( n = 4 = p \):

\[
\begin{align*}
  y_1 &:= b_1 \\
y_2 &:= b_2 - \ell_{2,1} \star y_1 \\
y_3 &:= b_3 - \ell_{3,1} \star y_1 - \ell_{3,2} \star y_2 \\
y_4 &:= b_4 - \ell_{4,1} \star y_1 - \ell_{4,2} \star y_2 - \ell_{4,3} \star y_3
\end{align*}
\]
**type 3 pipelining**

Make $y_1$ available in the next pipeline cycle:

![Diagram of type 3 pipelining](image)
We count the pipeline cycles for $p = 4$ or in general, for $p = n$:

1. The latency takes 4 cycles for $y_1$ to be at $P_4$, or in general: $n$ cycles for $y_1$ to be at $P_n$.

2. It takes then 6 additional cycles for $y_4$ to be computed by $P_4$, or in general: $2n - 2$ additional cycles for $y_n$ to be computed by $P_n$.

So it takes $n + 2n - 2 = 3n - 2$ pipeline cycles to solve an $n$-dimensional triangular system by an $n$-stage pipeline.
1. **Forward Substitution Formulas**
   - processing the L of the LU factorization
   - a third type of pipeline

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

$\Rightarrow$ all instructions in the $j$ loop are independent from each other!
Consider the inner loop in the algorithm to solve $Ly = b$:

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

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. 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 $y$: $y_i = 1$, for $i = 1, 2, \ldots, n$. We compute the right hand side $b = Ly$.

- For dimensions $n > 800$, hardware doubles are insufficient. With hardware doubles, the accumulation of round off is such that we lose all accuracy in $y_n$. We recall that condition numbers of $n$-dimensional triangular systems can grow as large as $2^n$. Therefore, we use quad double arithmetic.

- We use OpenMP, because it is Friday and OpenMP is easy.
hardware doubles are too inaccurate

Relying on hardware doubles is problematic:

$ time /tmp/trisol 10
last number : 1.0000000000000009e+00
real 0m0.003s user 0m0.001s sys 0m0.002s

$ time /tmp/trisol 100
last number : 9.9999999999974221e-01
real 0m0.005s user 0m0.001s sys 0m0.002s

$ time /tmp/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:

```c
{
    qd_real b[n], y[n];
    int i, j;

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

    srand(time(NULL));
    random_triangular_system(n, L, b);
}
```
void random_triangular_system
( int n, qd_real **L, qd_real *b )
{
    int i, j;
    for(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(j=i+1; j<n; j++)
        {
            L[i][j] = qd_real(0.0);
        }
    }
    for(i=0; i<n; i++)
    {
        b[i] = qd_real(0.0);
        for(j=0; j<n; j++)
        {
            b[i] = b[i] + L[i][j];
        }
    }
}
void solve_triangular_system_swapped
( int n, qd_real **L, qd_real *b, qd_real *y )
{
    int i, j;

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

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

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

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

    for(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 /tmp/trisol_qd_omp n p

On dimension $n = 8,000$ for varying number $p$ of cores.

<table>
<thead>
<tr>
<th>$p$</th>
<th>cpu time</th>
<th>real</th>
<th>user</th>
<th>sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21.240s</td>
<td>35.095s</td>
<td>34.493s</td>
<td>0.597s</td>
</tr>
<tr>
<td>2</td>
<td>22.790s</td>
<td>25.237s</td>
<td>36.001s</td>
<td>0.620s</td>
</tr>
<tr>
<td>4</td>
<td>22.330s</td>
<td>19.433s</td>
<td>35.539s</td>
<td>0.633s</td>
</tr>
<tr>
<td>8</td>
<td>23.200s</td>
<td>16.726s</td>
<td>36.398s</td>
<td>0.611s</td>
</tr>
<tr>
<td>12</td>
<td>23.260s</td>
<td>15.781s</td>
<td>36.457s</td>
<td>0.626s</td>
</tr>
</tbody>
</table>

The serial part is the generation of the random numbers for $L$ and the computation of $b = Ly$. 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 $21.240/12 = 1.770$. 

Introduction to Supercomputing (MCS 572) Solving Triangular Systems L-17 21 February 2014 22 / 23
Summary + Exercises

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

Exercises:

1. Consider the upper triangular system $Ux = y$, 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 $x$. For $n = 4$, draw the third type of pipeline.

2. Write a parallel solver with OpenMP to solve $Ux = y$. Take for $U$ a matrix with random numbers in $[0,1]$, compute $y$ so all components of $x$ 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 $Ly = b$ for distributed memory computers. Write a prototype implementation using MPI and discuss its scalability.