Solving Tridiagonal Systems in Parallel

Cosmin Oancea & Fritz Henglein
cosmin@diku.dk & henglein@diku.dk

Department of Computer Science
University of Copenhagen

August 15, 2012

TRIDAG Problem Statement

Parallel Algorithm for Computing X:

Given A and D, find X such that $A * X = D$,
where $A \in \mathbb{M}^{n \times n}$ is tridiagonal, and X and D vectors of size n.

```
                          A                      *     X       =     D
| b0   c0   0    0     ............... 0 |     | x0      |    | d0      |
| a0   b1   c1   0     ............... 0 |     | x1      |    | d1      |
| 0    a1   b2   c2   0   ............ 0 |     | x2      |    | d2      |
| ..................................... |  *  | ....    | =  | ....    |
| 0     0    ...    0      an-3  bn-2  cn-2 |  | xn-2    |    | dn-2    |
| 0     0    ...    0      0     an-2  bn-1 |  | xn-1    |    | dn-1    |
```

Functional Background: Parallel Prefix Sum (a.k.a. scan) and Map

Given a binary-associative operator $\odot$ with neutral element $e_\odot$, **scan**, defined below, has a well-known parallel algorithm of asymptotic complexity $O(log\ n)$!

$$\text{scan } \odot\ e_\odot\ [a_1, a_2, \ldots, a_n] = [a_1, a_1 \odot a_2, \ldots, a_1 \odot a_2 \odot \ldots \odot a_n]$$

Map is an embarrassingly-parallel functional-basic block: given f :: a -> a

$$\text{map f } [a_1, a_2, \ldots, a_n] = [\text{f } a_1, \text{f } a_2, \ldots, \text{f } a_n]$$

High-Level Solution (Sequential and Parallel)

Problem is to Find $X = [x_0, ..., x_{n-1}]^T$ such that:

$$\begin{array}{ccccc}
A & * & X & = & D \\
\left|\begin{array}{lllllll}
b_0 & c_0 & 0 & 0 & \multicolumn{2}{c}{\ldots\ldots\ldots} & 0 \\
a_0 & b_1 & c_1 & 0 & \multicolumn{2}{c}{\ldots\ldots\ldots} & 0 \\
0 & a_1 & b_2 & c_2 & 0 & \ldots\ldots & 0 \\
\multicolumn{7}{c}{\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots} \\
0 & 0 & \ldots & 0 & a_{n-3} & b_{n-2} & c_{n-2} \\
0 & 0 & \ldots & 0 & 0 & a_{n-2} & b_{n-1}
\end{array}\right|
& * &
\left|\begin{array}{l}
x_0 \\ x_1 \\ x_2 \\ \ldots \\ x_{n-2} \\ x_{n-1}
\end{array}\right|
& = &
\left|\begin{array}{l}
d_0 \\ d_1 \\ d_2 \\ \ldots \\ d_{n-2} \\ d_{n-1}
\end{array}\right|
\end{array}$$

- STEP 1: Compute the LU decomposition of A, i.e., $A = L * U$, where L and U are lower and upper-diagonal matrices $\in \mathbb{M}^{n \times n}$.
- STEP 2: Solve $L * Y = D$, i.e., compute partial solution Y.
- STEP 3: Solve $U * X = Y$, i.e., compute problem solution X.
- Each of the three steps needs to be parallel (of depth $O(logn)$)!

LU-Decomposition Recurrences

Writing the LU decomposition: $A = L * U$, where $L, U \in \mathbb{M}^{n \times n}$

$$
\underbrace{\begin{vmatrix} b_0 & c_0 & 0 & 0 & \ldots & 0 \\ a_0 & b_1 & c_1 & 0 & \ldots & 0 \\ \ldots & & & & & c_{n-2} \\ 0 & 0 & \ldots & 0 & a_{n-2} & b_{n-1} \end{vmatrix}}_{A}
=
\underbrace{\begin{vmatrix} 1 & 0 & 0 & \ldots & 0 \\ m_0 & 1 & 0 & \ldots & 0 \\ \ldots & & & & \\ 0 & \ldots & 0 & m_{n-2} & 1 \end{vmatrix}}_{L}
*
\underbrace{\begin{vmatrix} u_0 & c_0 & 0 & \ldots & 0 \\ 0 & u_1 & c_1 & \ldots & 0 \\ \ldots & & & u_{n-2} & c_{n-2} \\ 0 & \ldots & & 0 & u_{n-1} \end{vmatrix}}_{U}
$$

LU-decomposition Recurrences:

I. First, compute u_i, $i \in \{0 .. n-1\}$:

- I.a) $u_0 = b_0$
- I.b) $u_i = b_i - (a_{i-1} * c_{i-1}) / u_{i-1}, \quad i \in \{1 .. n-1\}$

II. Knowing the u's, we can easily compute the m's:

$m_i = a_i / u_i, \; i \in \{0 .. n-2\}$

The Difficult Recurrence of LU-Decomposition I.b)

Parallel Algorithm: **scan** with $\mathbb{M}^{2\times 2}$ multiplication operator

Recurrence:

I.a) $u_0 = b_0$

I.b) $u_i = b_i - (a_{i-1} * c_{i-1}) / u_{i-1}, \quad i \in \{1 .. n-1\}$

1) Substitute in I.b) $u_i \leftarrow q_{i+1} / q_i$,

where $q_1 = b_0$ and $q_0 = 1.0$, i.e., $u_0 == b_0$ still holds! We obtain:

I.c) $\begin{vmatrix} q_{i+1} \\ q_i \end{vmatrix} = \begin{vmatrix} b_i & - a_{i-1} * c_{i-1} \\ 1.0 & 0.0 \end{vmatrix} * \begin{vmatrix} q_i \\ q_{i-1} \end{vmatrix} \quad i \in \{1 .. n-1\}$

2) Denoting the above 2×2 matrix by $S_i \in \mathbb{M}^{2\times 2}$ we obtain by induction:

I.d) $| q_{i+1} \;\; q_i |^T = (S_i * S_{i-1} * S_1) * | b_0 \;\; 1.0 |^T$

3) We can compute all such matrices, i.e., $S_i * S_{i-1} * S_1$ with $i \in \{1..n-1\}$, by applying **scan** with the (associative) matrix-multiplication operator (*)

I.e) matrices = scan (*) $[S_{n-1}, S_{n-2}, \ldots, S_1]$

4) Finally, we compute each $[q_{i+1}, q_i]^T$ by multiplying its corresponding matrix with $[b_0, 1.0]^T$, and compute $u_i = q_{i+1} / q_i$. Both are PARALLEL:

I.f) q_pairs = map (matVectMult $[b_0, 1.0]$) matrices

I.g) u = map (lambda [x,y] $\rightarrow$ x/y) q_pairs

Implementing The LU-Decomposition Recurrences

5) Recurrence II, i.e., $\mathtt{m}_i = \mathtt{a}_i \; / \; \mathtt{u}_i$, $i \in \{0..n-2\}$ is written in parallel as: `m = map (lambda(x,y) -> x/y) (zip a u)`

Haskell Code for Parallel LU Decomposition:

```
computeLU_us_par :: (Fractional a, Ord a) => [a] -> [a] -> [a] -> [a]
computeLU_us_par a b c =
    let matMult2by2 [a1, a2, a3, a4] [b1, b2, b3, b4] =
            [ (b1*a1+b2*a3), (b1*a2+b2*a4), (b3*a1+b4*a3), (b3*a2+b4*a4) ]
        matVectMult b0 [a1, a2, a3, a4] = (a1*b0 + a2) / (a3*b0 + a4)

        b0              = head b
        abc             = zip3 a (tail b) c
        matrices_inp  = map (lambda (a,b,c)->[b, (- (a * c)), 1.0, 0.0]) abc
        matrices_out  = scanl matMult2by2 [1.0, 0.0, 0.0, 1.0] matrices_inp
    in map (matVectMult b0) matrices_out

computeLU_ms_par :: (Fractional a, Ord a) => [a] -> [a] -> [a]
computeLU_ms_par a u = map (lambda(x,y) -> x/y) (zip a u)

decompLU :: (Fractional a, Ord a) => [a] -> [a] -> [a] -> ([a],[a])
decompLU a b c = (computeLU_us_par a b c, computeLU_ms_par a us)
```

STEP 2 & 3: Solving For/Backward Recurrences

STEP 2: Parallel Algorithm for $L * Y = D$ (forward recurrence)

1) By doing the arithmetics, it comes down to solving the recurrence:
   II.a) $y_0 = d_0$
   II.b) $y_i = d_i - m_{i-1} * y_{i-1}, \quad i \in 1 .. n-1$

2) We shall use **scan** with linear-function-composition operator:
   i) Consider $f_1(x) = a_1 + b_1*x$ and $f_2(x) = a_2 + b_2*x$
   ii) $(f_2 \diamond f_1)(x) = (a_2+b_2*a_1) + (b_1*b_2)*x$
   iii) We represent such a function f as a pair (a, b), and implement function application via operator: apply x (a, b) = a + b*x

3) Denoting $f_i = (d_i, -m_{i-1} * y_{i-1}), \ i \in 1 .. n-1$
   i) It follows by induction that $y_i = (f_i \diamond f_{i-1} \diamond \ldots \diamond f_1)(d_0)$
   ii) We do a **scan** with $\diamond$ and neutral element (0.0, 1.0):
       f_out = scan ($\diamond$) (0.0, 1.0) [f_1, ..., f_{n-1}]
   iii) Finally, apply the f_out's to d_0 to compute all y_i, $i \in \{1 .. n-1\}$
       y = map (apply d_0) f_out

STEP 3: Solving $U * X = Y$ is similar to STEP 2 (but *backwards*):

III.a) $x_{n-1} = y_{n-1}/u_{n-1}$ b) $x_i = y_i/u_i - c_i*x_{i+1}/u_i, \ i \in \{n-2..0\}$

Putting The Implementation Together

Haskell Code for For/Backward Recurrence and TRIDAG

```
◇ :: (Fractional a, Ord a) => (a,a) -> (a,a) -> (a,a)
(a_1, b_1) ◇ (a_2, b_2) = (a_2 + b_2*a_1, b_2*b_1)

forward_rec :: (Fractional a, Ord a) => [a] -> [a] -> [a]
forward_rec d m = let d0 = head d
                      fs = scanl (◇) (0.0,1.0) (zip (tail d) (map (0.0 -) m))
                  in map (lambda(a,b) -> a + b*d0) fs

backward_rec :: (Fractional a, Ord a) => [a] -> [a] -> [a] -> [a]
backward_rec y u c = let yuc = reverse (zip3 y u (c ++ [0.0]) )
                         (y_nm1, u_nm1, c_nm1) = head yuc
                         fs_inp = map (lambda(y,u,c)->(y/u,-c/u)) (tail yuc)
                         fs_out = scanl (◇) (0.0, 1.0) fs_inp
                         x_nm1 = y_nm1 / u_nm1
                     in reverse (map (lambda(a,b) -> a + b*x_nm1) fs_out)

tridag :: (Fractional a, Ord a) => [a] -> [a] -> [a] -> [a] -> [a]
tridag a b c d = let (u, m) = decompLU a b c
                     y = forward_rec d m
                 in      backward_rec y u c
```

Resources

[1] Harold S. Stone, "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations", *In Journal of the ACM*, Vol 20(1), pp. 27–38, January 1973.

- The above paper can be found in the `DOC` folder.
- Haskell-source code can be found in the `CODE/SourceCodeHaskell` folder.
  Compile with: "ghc -O2 -o test TRIDAG.hs"
  and run with: "./test"
- Target sequential-C and GPGPU code can be found in the `CODE/SourceCodeHaskell` folder.
  The OpenCL (target) implementation solves 131072 tridiagonal systems ($N = 128$). My mobile GPGPU version is about 18x faster than the (sequentially-hand-optimized) C code (on CPU).
  Limitations: N needs to be a power of two and $N \leq 1024$.