CS855: Parallel Linear Algebra (II)

Marc Moreno Maza

February 19, 2008

Outline

- Triangular matrices and back substitution
- Tridiagonal systems and even-odd reduction
- Parallel direct methods for solving SLEs

Let A be a lower-triangular $n \times n$ matrix with coefficients in a field, i.e., $a_{ij} = 0$ for $1 \leq i < j \leq n$. Assume also $a_{ii} \neq 0$ for all $1 \leq i \leq n$. Consider the system of linear equations (SLE):

$$A \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

The classical method is the **back substitution**. If the number p of processors satisfies $p = n$, then there is a trivial but inefficient parallelization:

- The i-th equation

  $$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{ii}x_i = b_i$$

  is assigned to Processor P_i.

- Processor P_1 computes directly x_1 and broadcasts it to all other processors,

- every Processor P_j, for $j > 1$, updates its equation by subtracting $a_{j1}x_1$ from it.
- In general, in the i-th step, Processor P_i computes the value of x_i, broadcasts it to all processors P_j, for $j > i$, and they update their equations by subtracting $a_{ji}x_i$.

The sequential complexity is in $O(n^2)$ field operations. The parallel complexity depends on the complexity $b(p)$ of broadcasting a number to p processors, since all the other operations take constant time per one step

- If $b(p) = \Theta(p)$ (say on a 1-D mesh) then $T(n, n) = O(n^2)$ and the parallel algorithm is useless.
- If $b(p) = O(\log p)$ (hypercube) then $T(n, n) = O(n \log n)$, but the cost is not optimal.

Intuitively, we can gain a reasonable efficiency only if we reduce p. We consider two cases:

- If cyclic row-major mapping is used, then there are n steps as before.

In every step, one processor broadcasts one number in $b(p)$ time and each processor P_j performs substitutions in all of its $k_j \leq n/p$ equations, where k_j decreases by one every p-th step. Then

$$T(n,p) = n\ O(\log p\ +\ n/p) = O(n\log p) + O(n^2/p).$$

- If block row-major mapping is used, then the algorithm has only p parallel steps. In step i, Processor P_i computes its n/p variables, broadcasts them in one message to all P_j, for $j > i$, in time $O((n/p)\log p)$ and these processors perform their substitutions into their n/p equations. Then, we have

$$T(n,p) = p\ O(n^2/p^2) + p\ O((n/p)\log p) = O(n\log p) + O(n^2/p).$$

which is asymptotically comparable to the preceding case. In practice, however, the latter implementation should provide better performance due to less startups.

A **tridiagonal** system of equations has a tridiagonal coefficient matrix: all the elements except for

- those on the main diagonal and
- on the diagonals above it and below it

are zero. Formally, we have $a_{ij} = 0$ if $\mid j - i \mid > 1$. Hence, for $1 \leq i \leq n$, the i-th equation looks like

$$f_i x_{i-1} + g_i x_i + h_i x_{i+1} = b_i \tag{1}$$

where

- f_i, g_i, h_i, b_i are the coefficients,
- $x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n$ are the unknowns,
- $x_0 = x_{n+1} = 0$ are introduced for convenience.

The idea of the even-odd reduction is based on a simple observation. If

$g_i \neq 0$, then x_i can be expressed in terms of x_{i-1} and x_{i+1} by

$$x_i = \frac{1}{g_i}(b_i - f_i x_{i-1} - h_i x_{i+1}).$$

If we substitute these expressions for all x_i with odd i, we get a new tridiagonal SLE with variables $x_2, x_4, x_6, \ldots$.

- The number of equations and unknowns has dropped to one half.
- This **sequential** algorithm proceeds with substitutions so that after $n-1$ steps, a single equation in a single variable remains. We can then calculate backward the values of other variables in time $O(n)$. The algorithm is optimal, since the trivial sequential lower bound is $\Omega(n)$ (at least, $\Omega(n)$ steps are needed just to read the input).
- The algorithm needs to be refined if at some stage a coefficient g_i is null.

The parallelization of this algorithm is straightforward.

- Assume first $p = n$ and let P_i be assigned to Equation i.

- After k parallel reductions, for $1 \leq k \leq \lfloor \log n \rfloor$, we are left with variables x_i where i is a multiple of 2^k.
- So, Processor $j2^k$ needs to communicate with processor $(j+1)2^k$. Let us look at different implementations.
  - Consider first a 1-D mesh $M(n)$. The communication cost in step k is in $O(2^k)$. In particular, for the last step it is in $O(n)$. The total sum over all steps is $\Theta(n)$ which brings into the total parallel time a term of the same order as the sequential time itself.
  - Consider an Hypercube. Then any parallel communication step takes constant time.

If we scale the number of processors to $p < n$, then the same idea applies. The best mapping for A is just block row-major striping. Assuming constant time communication steps, then

$$T(n,p) = O(n/p + n/(2p) + n/(4p) + \cdots) + O(\log p) = O(2n/p + \log p)$$

which indicates a very nice scalability.

Consider again a SLE $Ax = b$ of order n over a field. Assume that A is nonsingular and dense. It is natural to compute a LU decomposition of A where

- L is a lowertriangular matrix
- U is an uppertriangular matrix

such that $A = LU$ holds.

- We choose the Gauss LU decomposition, which is unique after imposing that every element on the diagonal of L is one. (Cholesky's scheme is another choise where $L_{ii} = U_{ii}$ for all $1 \leq i \leq n$.

In a parallel setting, the calculation of matrices L and U can proceed simultaneously. Assume first $r = \sqrt{p}$ and view processors as a virtual 2-D mesh $M(r, r)$.

- A common method is to calculate column i of L immediately after row i of U, successively for $i = 1, 2, \ldots, n$.

Consider below $n = r = 4$. We want:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix}$$

That is

$$A = \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} & l_{21}u_{14} + u_{24} \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} & l_{31}u_{14} + l_{32}u_{24} + u_{34} \\ l_{41}u_{11} & l_{41}u_{12} + l_{42}u_{22} & l_{41}u_{13} + l_{42}u_{23} + u_{43}u_{33} & l_{41}u_{14} + l_{42}u_{24} + l_{43}u_{3} \end{pmatrix}$$

Assume that we have n^2 processors P_{ij} for $1 \leq i \leq j \leq n$:

- Intially, each of them stores a_{ij}.
- In the end P_{ij} stores u_{ij} if $j \leq i$ otherwise P_{ij} stores l_{ij}

The principle of the algorithm.

STEP 1. Two sub-steps:

Horizontally Let $u_{1j} = a_{1j}$ for $1 \leq j \leq n$. For $2 \leq j \leq n$, Processor P_{1j} broadcasts u_{1j} to all Processor P_{ij} for $2 \leq i \leq n$.

Vertically Processor P_{11} computes the inverse of u_{11} and broadcasts it to Processors P_{i1} for $2 \leq i \leq n$. Then, for $2 \leq i \leq n$, Processor P_{i1} computes l_{i1} and broadcasts it to P_{ij} for $2 \leq j \leq n$.

STEP 2. Two sub-steps:

Horizontally For $2 \leq j \leq n$, Processor P_{2j} computes u_{1j} and broadcasts it to all Processor P_{ij} for $3 \leq i \leq n$.

Vertically Processor P_{22} computes the inverse of u_{22} and broadcasts it to Processors P_{i2} for $3 \leq i \leq n$. Then, for $3 \leq i \leq n$, Processor P_{i2} computes l_{i2} and broadcasts it to P_{ij} for $3 \leq j \leq n$.

STEP K. for $3 \leq k \leq n$. Two sub-steps:

Horizontally For $k \leq j \leq n$, Processor P_{kj} computes u_{kj} and broadcasts it to all

Processor P_{ij} for $k+1 \leq i \leq n$.

Vertically Processor P_{kk} computes the inverse of u_{kk} and broadcasts it to Processors P_{ik} for $k+1 \leq i \leq n$. Then, for $k+1 \leq i \leq n$, Processor P_{ik} computes l_{ik} and broadcasts it to P_{ij} for $k+1 \leq j \leq n$.

This is only a principle. To be a precise algorithm, one still needs:

- to state precisely the formulas computing the u_{kj}'s and the l_{ik}'s (not difficult, in fact, from the coefficients of the matrix product LU).
- to scale the algorithm for the case where each processor has to work with matrix blocks rather than coefficients.

For the case $p = n^2$, the algorithm principle above leads to

$$T(n, n^2) = nO(\log p) + nO(1) = O(n \log n),$$

where the first contribution comes from the *broadcasting* (assuming an hypercube topology) and the second one comes from the arithmetic operations (which is, indeed, a constant cost for each proceesor, during one parallel step).

References and Acknowledgments

Most of the material of these course notes are inspired by the following.

- Parallel Algorithms (WISM 459, 2005/2006) by Rob Bisseling.
  `http://www.math.uu.nl/people/bisselin/pas1.html`
- Topics in Parallel Computing (CS 838, Univ. of Wisconsin-Madison, Spring 1999) by Pavel Tvrdik.
  `http://www.cs.wisc.edu/ tvrdik/cs838.html`
- Parallel Algorithms (CS 662, San Diego Univ., Spring 1996) by Roger Whitney.
  `http://www.eli.sdsu.edu/courses/spring96/cs662/notes/`