Massachusetts Institute of Technology | Lecture 10
6.892: Theory of Parallel Systems | March 11, 1997
Professor Charles E. Leiserson | Scribe: Julia Khodor

Lecture 10: Systems of Linear Equations

Sorting

In this lecture we will discuss solving systems of linear equations through LU-decomposition and sorting using serial and parallel versions of Merge.

1 Systems of Linear Equations

1.1 Matrix-vector multiplication

Let $y = Ax$, where A is $n \times n$ matrix, and x and y are n-vectors. Then $y_i = \sum_{j=1}^{n} a_{ij} x_j$.

Since there are n y_i's, and calculating each of these on one processor takes $O(n)$ time (n scalar multiplications, and $n-1$ additions),

$T_1 = Work = \Theta(n^2)$

If the additions are done on a binary tree (adding up each x subtree to get a y_i leaf of the top portion of the tree),

$T_\infty = CP = \Theta(\lg n)$

Trading the short CP for better locality (which leads to less overhead work) gives $T_{\infty_1} = \Theta(\lg^2 n)$. $T_{\infty_2} = \Theta(n)$ if we do the calculation with no temporary variables.

1.2 Systems of Linear Equations

In this case, we are looking at solving $Ax = b$ for x, where A is still an $n \times n$ matrix, and x and b are n-vectors.

The particular method we are going to look at is called LU-decomposition. We will explore both the theoretical basis for the method, and practical implementation thereof.

1.2.1 Theory

1. Factor $A = L \times U$, where

   - L is unit lower triangular (Fig. 1)
   - U is upper triangular (Fig. 2)

     Note that such factorisation is always possible if A is

     - symmetric, positive definite
     - irreducible, diagonally dominant,

     where the above terms are defined as follows:

     - Symmetric: $A^T = A$

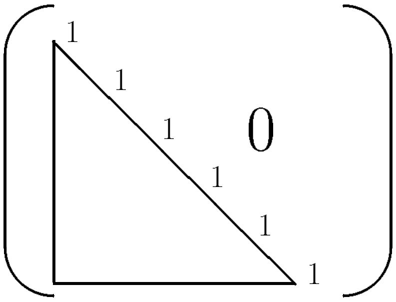

Figure 1: Unit lower triangular matrix A has all 1's on the diagonal and all 0's above the diagonal, i.e., $a_{ij} = 0 \; \forall \; i < j$, $a_{ij} = 1 \; \forall \; i = j$

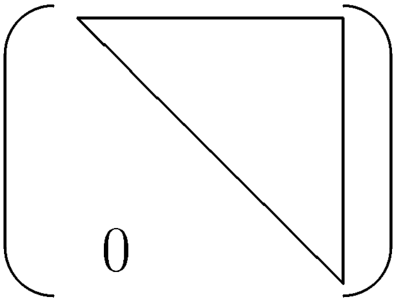

Figure 2: Upper triangular matrix A has all 0's below the diagonal, i.e. $a_{ij} = 0 \; \forall \; i > j$

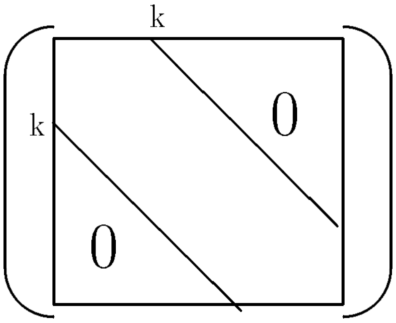

Figure 3: Band matrix A has non-zero values only within a band of width k around the diagonal, i.e. $a_{ij} = 0 \; \forall \; (i + j) \neq k$

- Positive definite: $\forall \; x, x^T Ax \geq 0$

Also note that in the above two cases no pivoting is necessary.

2. Solve $Ly = b$ by **forward substitution** (to be defined shortly) Solve $Ux = y$ by **back substitution** (see above comment)

   Note that $Ly = L(Ux) = b = Ax$.

   At this point the question of what's all the fuss is about comes to mind. More precicely, why bother being clever with the LU-decomposition, when we could simply compute A inverse. There are two reasons:

   (a) Inverse may be highly ill-conditioned.

   (b) A may be a very sparsely-structured, e.g. a band matrix, as in Fig. 3

   Inverse of a band matrix fills up, while L and U of the band matrix are themseles band matrices.

   In other words, L and U tend to preserve good properties of a matrix. For more information see CLR, chapter 31.

1.2.2 Implementation

1. **Forward Substitution.** Solve $Ly = b$ by Divide and Conquer (D&C).

   We can write the problem out as:

   $$\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} L_{11}y_1 \\ L_{21}y_1 + L_{22}y_2 \end{bmatrix}$$

   Note that in the above equation L_{11} and L_{22} are unit lower triangular, while L_{21} is dense.

Algorithm:

- Solve $L_{11}y_1 = b_1$ for y_1 recursively.
- sync
- Compute $b_2' = b_2 - L_{21}y_1$, where $L_{21}y_1$ is calculated using matrix-vector multiplication
- sync
- Solve $L_{22}y_2 = b_2'$ for y_2 recursively

Analysis of the above algorithm yields

$$\begin{aligned} T_1(n) &= 2T_1(\tfrac{n}{2}) + \Theta(n^2) \text{ (work for matrix-vector multiplication from section 1)} \\ &\quad \text{by case 3 of Master's Theorem} \\ &= \Theta(n^2) \end{aligned}$$

$$\begin{aligned} T_\infty(n) &= 2T_\infty(\tfrac{n}{2}) + \Theta(\lg n) \text{ (CP for matrix-vector multiplication from section 1)} \\ &\quad \text{by case 1 of Master's Theorem} \\ &= \Theta(n) \end{aligned}$$

If we use no temps in matrix-vector multiplication,

$$\begin{aligned} T_\infty(n) &= 2T_\infty(n/2) + \Theta(n) \\ &\quad \text{By case 2 of Master's Theorem} \\ &= \Theta(n(lgn)) \end{aligned}$$

2. **Back Substitution**

Can be done similarly to forward substitution. However, this is a bit more complicated, because the diagonal of U doesn't necesserely consist of all 1's. This means that, for instance, we need to divide in the base case.

3. **LU-decomposition**

So, the success of the method depends entirely on our ability to perform the folowing factorization:

$$\begin{bmatrix} A_{11} & A_{12} \\ A21 & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{bmatrix}$$

And, upon careful consideration we note that it is not that difficult to do, and, indeed, can be done using the following:

Algorithm:

- Factor $A_{11} = L_{11}U_{11}$ recursively
- sync
- solve $L_{11}U_{12} = A_{12}$ for U_{12} by forward substitution. This involves n vector-matrix multiplications. We can get better locality by using D&C here.
- Solve $L_{21}U_{11} = A_{21}$ for L_{21} by back substitution.
- sync
- Factor $A_{22} - L_{21}U_{12} = L_{22}U_{22}$ recursively.

Analysis of the above algorithm yields

$$\begin{aligned} T_1(n) &= 2T_1(\tfrac{n}{2}) + \Theta(n^3) \text{ } (n\times(\text{work for calculating } U_{12} \text{ and } L_{21})) \\ & \text{ by case 3 of Master's Theorem} \\ &= \Theta(n^3), \text{ same as serial implementation.} \end{aligned}$$

Note: If we use no-temps version of the algorithm, data dependencies will be the same as in the serial case.

$$\begin{aligned} T_\infty(n) &= 2T_\infty(\tfrac{n}{2}) + \Theta(n) \text{ (from forward/back substitution)} \\ & \text{ by case 2 of Master's Theorem} \\ &= \Theta(n(\lg n)) \end{aligned}$$

Note that we can pretty much use the no temps version of vector-matrix multiplication here, because it doesn't cost us assymptotically $T_\infty(n) = \Theta(n(\lg^2 n))$.

Thus, the average parallellism is:

$\bar{P}(n) \triangleq \frac{T_1(n)}{T_\infty(n)} = \Theta(\frac{n^2}{\lg n})$

4. **Sparse implementation**

Typically, sparce matrix is represented as a doubly-linked list, as in Fig 4.

This is not a great representation for the D&C algorithms, because it requires traversing the list multiple times in order to get all the elements of the subproblem. The idea for a scalable parallel implementation of the D&C algorithms is to use a quad-tree representation of a matrix instead. See Figure 5.

If a submatrix$= 0$, store nil. Note that the quad-tree has height $\lg n$. Also, when fill-in occurs, all we have to do is add another branch/subbranch to the tree.

Idea for project: Matlab toolbox for sparse matrices.

The current implementation is about 30% slower then the best serial implementation, but it works and scales.

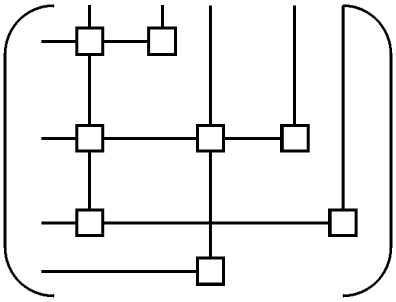

Figure 4: Representing sparce matrices as doubly-linked lists.

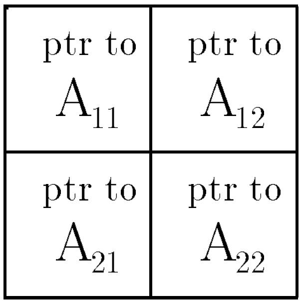

Figure 5: Representing sparce matrices as quad-trees.

2 Sorting

In this section we will consider sorting using Mergesort. We will see that the performance of the algorithm depends mostly on our choice of Merge and we will consider two implementations of Merge, and their impact on the overall performance of Mergesort.

Consider the following code for Mergesort.

```
Mergesort (A, n)
{
    if (n==1)
        return;
    spawn Mergesort (A, n/2);
    spawn Mergesort (A[n/2], n-n/2);
    sync;
    spawn Merge (A, n);
    sync;
    return;
}
```

So, the analysis of the performance of this algorithm relies heavily on our choice of Merge.

2.1 Serial merge

If we consider the serial merge, we find $T_1 = \Theta(n)$, and $T_\infty = \Theta(n)$. This leads to the following recurrence relations on the work and CP of the entire algorithm:

$$\begin{aligned} T_1(n) &= 2T_1(n/2) + \Theta(n) \\ &\quad \text{By case 2 of Master's theorem:} \\ &= \Theta(n(\lg n)) \end{aligned}$$

$$\begin{aligned} T_\infty(n) &= T_\infty(\tfrac{n}{2}) + \Theta(n) \\ &\quad \text{By case 3 of Master's theorem:} \\ &= \Theta(n) \end{aligned}$$

Which gives us $\bar{P}(n) = \Theta(\lg n)$– puny!

Therefore, we would like to do a

Figure 6: The idea of Akl & Santoro merge is to find median of two sorted sequences (which can be done in $O(\lg n)$ time), and to recursively merge the subsequences, pivoting on the medians.

2.2 Parallel Merge

We have already seen Batcher's bitonic merge. But that has work $T_1(n) = \Theta(n(\lg n))$, which would add to the sort work in the work recurrence for the Mergesort. We would like to find a merge whose work term in the Mergesort recurrence would be dominated by the sorting work. One such algorithm is due to Akl and Santoro. See Figure 6 for the idea of the algorithm.

Consider the following code for the variant of the above algorithm. Note that we can not do merge in place– we need to copy. This introduces an extra level of complexity, not shown in the code.

```
Merge(A, l, B, m)                          /* temps not shown*/
{
    if (m>l)
        {
            Merge (B, m, A, l);            /* if |B| > |A|, switch the arrays around, and */
            return;                        /* Merge */
        }
    Find j B[j] ≤ A[l/2] ≤ B[j + 1], using binary search;
    spawn Merge (A, l/2, B, j);
    Merge (&A[l/2], l-l/2, &B[j+1], m-j);
    sync;
}
```

The idea is illustrated in Figure 7.

Analysis of the above algorithm:

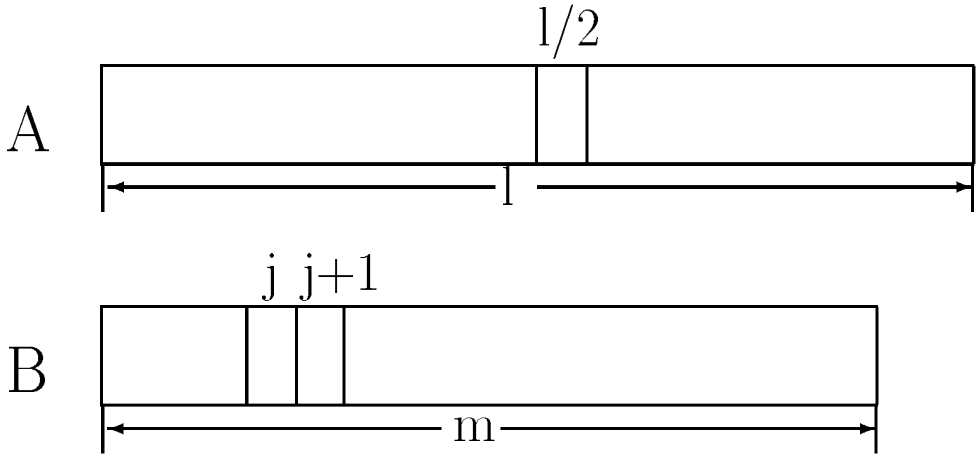

Figure 7: This variant of Akl& Santoro merges lists based on the value of the $\frac{l}{2}$-th element of the longer list, and the elements of the shorter list that will flank the above element in the final list.

Let the size of the problem be $n = l + m$. Then, in the worst case we would have to merge half the list A with all of list B, which, at most is length $\frac{3}{4}n$, and we have to do the $O(\lg n)$ work to find the appropriate j. Thus:

$$\begin{aligned} T_\infty(n) &\leq T_\infty(\tfrac{3}{4}n) + O(\lg n) \\ &\quad \text{By case 2 of Master's theorem:} \\ &= O(\log^2 n) \end{aligned}$$

And for work, we have

$$T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + O(\lg n),\ \forall\ \tfrac{1}{4} \leq \alpha \leq \tfrac{3}{4} \qquad (1)$$

It turns out that the worst case is when $\alpha = \frac{3}{4}$. But this doesn't actually matter, since, as we will prove directly the work in all cases is essencially the same as if we were cutting both lists in half.

Proof by substitution:

(a) Guess $T_1(n) = \theta(n)$

(b) Then

$$\begin{aligned} T_1(n) &\leq c\alpha n + c(1-\alpha)n + O(\lg n) \\ &\leq c(\alpha + 1 - \alpha)n + O(\lg n) \\ &= cn + O(lgn) \\ &= \Theta(n) \end{aligned}$$

Thus, by case 1 of Master's theorem,
$(1) = O(n)$

Back to the original Mergesort code, substituting the above analysis of the parallel Merge, we get for the Mergesort:

$$\begin{array}{rcl} T_1(n) & = & \Theta(n(\lg n)) \\ T_\infty & = & T_\infty(\frac{n}{2}) + \Theta(\lg^2 n) \\ & & \text{by case 2 of Master's theorem:} \\ & = & \Theta(\lg^3 n) \end{array}$$

And, the average parallelism is:

$\bar{P}(n) = \Omega(\frac{n}{\lg^2 n})$

By being slightly more clever, can get $\bar{P}(n) = \Omega(\frac{n}{\lg n})$, but this adds constant factor to the work, so the algorithm discussed in detail above is actually the one preffered in practice.

Project idea: efficient sorting algorithm.