ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

LU factorization

- systems of linear equations are ubiquitous in numerical analysis
- let A be an $n \times n$ matrix, b a known n-vector; we are looking for x such that

$$Ax = b \tag{1}$$

- a commonly used direct method for solving this system is to convert A into the product of a lower triangular matrix L with an upper triangular matrix U

$$A = LU \tag{2}$$

  known as *LU* factorization
- Eq. 1 becomes

$$LUx = b \tag{3}$$

  which we can now solve in two simpler steps

$$Ly = b \tag{4}$$

$$Ux = y \tag{5}$$

  where we first solve the lower triangular system by forward substitution, followed by solving the upper triangular system by back substitution to obtain x

LU by Gaussian elimination

- we can compute the *LU* factorization of *A* using Gaussian elimination

Algorithm 1: LU(A)

```
1 for k = 1 to n − 1 do
2     for i = k + 1 to n do
3         L_ik = A_ik / A_kk
4     for j = k + 1 to n do
5         for i = k + 1 to n do
6             A_ij = A_ij − L_ik A_kj
```

  which encodes *L* and *U* in place by overwriting *A*
- Alg. 1 requires roughly $n^3/3$ multiply-adds and $n^2/2$ divisions
- we may also need *pivoting* to ensure numerical stability (and existence)
- Alg. 1 is one of many algorithms expressed essentially as a triply nested loop
  - the three indices can be ordered in any of 3! ways, with totally different memory access patterns
  - in parallel, the *kij* and *kji* forms may be the most efficient

Parallel LU decomposition

- number fine grain tasks as (i,j) with $i,j = 1,\ldots,n$; each task
  - stores A_{ij}
  - computes and stores U_{ij}, if $i \leq j$
  - computes and stores L_{ij}, if $i > j$

  yielding a two dimensional array of n^2 tasks
- no need to compute and store
  - the zeroes in the lower triangle of U
  - the unit diagonal and the zeroes in the upper triangle of L
- in order to create p coarse grain tasks we could combine
  - n/p rows or columns of fine grain tasks
  - $(n/\sqrt{p}) \times (n/\sqrt{p})$ blocks of tasks

  and map each one to a process

Communication patterns for parallel LU decomposition

Algorithm 2: LU(A, task=(i,j))

1 **for** $k = 1$ **to** $\min(i,j) - 1$ **do**
2 **recv** A_{kj}
3 **recv** L_{ik}
4 $A_{ij} = A_{ij} - L_{ik}A_{kj}$
5 **if** $i \leq j$ **then**
6 **broadcast** A_{ij} **to** (k,j), $k = i+1, \ldots, n$
7 **else**
8 **recv** A_{jj}
9 $L_{ij} = A_{ij}/A_{jj}$
10 **broadcast** L_{ij} **to** (i,k), $k = i+1, \ldots, n$

Row coarsening

- with one dimensional row coarsening
  - we forgo parallelism in updating rows
  - there is no need to broadcast the multipliers L_{ij} since each row is contained entirely within a task
  - we still need the vertical broadcasts of matrix rows to the tasks below

Algorithm 3: LU(A, task=(i,j)) by rows

```
1  for k = 1 to n − 1 do
2      if k ∈ myrows then
3          broadcast {A_kj : k ≤ j ≤ n}
4      else
5          recv {A_kj : k ≤ j ≤ n}
6      for i ∈ myrows, i > k do
7          L_ik = A_ik / A_kk
8      for j = k + 1 to n do
9          for i ∈ myrows, i > k do
10             A_ij = A_ij − L_ik A_kj
```

Observations on row coarsening

- each task becomes idle as soon as it last row is completed
  - if rows are contiguous, a task may finish long before the overall computation is done
  - even worse, updating rows requires progressively less work with increasing row number
- we may improve concurrency and load balance
  - by assigning rows to tasks in a cyclic manner where row i is updated by task $i \mod p$
  - other mappings may be useful
- other improvements involve overlapping computation with communication
  - at step k, each task completes updating its portion of the remaining unreduced matrix before moving on to step $k+1$
  - however, the task that owns the $k+1$ row could broadcast it as soon as it becomes available, before moving on to the step k update
  - this *send ahead* strategy may grant other tasks earlier access to the data necessary to start working on the next step

Column coarsening

Algorithm 4: LU(A, task=(i,j)) by columns

```
1  for k = 1 to n − 1 do
2      if k ∈ mycolumns then
3          for i = k + 1 to n do
4              L_ik = A_ik / A_kk
5          broadcast {L_ik : k < i ≤ n}
6      else
7          recv {L_ik : k < i ≤ n}
8      for i ∈ mycolumns, j > k do
9          for i = k + 1 to n do
10             A_ij = A_ij − L_ik A_kj
```

- observations similar to row coarsening apply

Block coarsening

Algorithm 5: LU(A, task=(i,j)) by blocks

1 **for** $k = 1$ **to** $n-1$ **do**
2 **if** $k \in myrows$ **then**
3 **broadcast** $\{A_{kj} : j \in mycolumns, j > k\}$ **to** all tasks in my task column
4 **else**
5 **recv** $\{A_{kj} : j \in mycolumns, j > k\}$
6 **if** $k \in mycolumns$ **then**
7 **for** $i \in myrows,\ i > k$ **do**
8 $L_{ik} = A_{ik}/A_{kk}$
9 **broadcast** $\{L_{ik} : i \in myrows, i > k\}$ **to** all tasks in my task row
10 **else**
11 **recv** $\{L_{ik} : i \in myrows, i > k\}$
12 **for** $j \in mycolumns,\ j > k$ **do**
13 **for** $i \in myrows,\ i > k$ **do**
14 $Aij = A_{ij} - L_{ik}A_{kj}$

Observations on block coarsening

- each task becomes idle as soon as it last row and column are completed
  - if rows and columns are in contiguous blocks, a task may finish long before the overall computation is done
  - even worse, computing multipliers and updating blocks requires progressively less work with increasing row and column numbers
- we may improve concurrency and load balance
  - by assigning rows and columns to tasks in a cyclic manner where A_{ij} is assigned to task $(i \mod \sqrt{p}, j \mod \sqrt{p}$
  - other mappings may be useful
- other improvements involve overlapping computation with communication
  - at step k, each task completes updating its portion of the remaining unreduced submatrix before moving on to step $k+1$
  - the broadcast of each segment of row $k+1$, and the computation and broadcast of each segment of multipliers for step $k+1$, can be initiated as soon as the relevant segments of row $k+1$ and column $k+1$ have been updated by their owners, before moving to competing the update for step k
  - this *send ahead* strategy may grant other tasks earlier access to the data necessary to start working on the next step

Pivoting

- the order of rows of A does not affect the solution to the system of equations
  - *partial pivoting* sorts the rows by the largest absolute value of the leading column of the remaining unreduced matrix
  - this choice ensures that the magnitude of the multipliers do not exceed 1, which
    - reduces amplification of round-off errors
    - ensures existence
    - improves numerical stability
- partial pivoting introduces a permutation matrix P, which leads to the factorization

$$PA = LU \tag{6}$$

which implies that the solution x is obtained through

$$Ly = Pb \tag{7}$$

$$Ux = y \tag{8}$$

with forward substitution in the lower triangular system, followed by back substitution in the upper triangular system

Pivoting in parallel

- increased numerical stability costs increased parallel complexity and significant performance implications
- for one dimensional coarsening by column, the search for the pivot element requires no extra communication, but it is purely serial
  - once the pivot is found, the index of the pivot row must be communicated to the other tasks, and rows must be explicitly or implicitly interchanged in each task
- for coarsening by rows, the search for the pivot is parallel, but it requires communication among tasks and inhibits the overlapping of successive steps
  - if rows are explicitly interchanged, then only two tasks are involved
  - if rows are implicitly interchanged, changes to the assignment of rows to tasks are required, which has effects on concurrency and load balance
- in the presence of partial pivoting, column and row coarsening trade off on the relative speeds of computation versus communication
- with two dimensional coarsening, pivot search is parallel but requires communication among tasks along columns and destroys the possibility of overlapping successive steps

Alternatives to pivoting

- various alternatives have been proposed
  - constraining pivoting to blocks of rows
  - pivoting when the multiplier exceeds a given threshold
  - pairwise pivoting
- these strategies are not foolproof, and trade off some stability and accuracy for speed

Cholesky factorization

- when A is a positive definite symmetric matrix is has a Cholesky factorization

$$A = LL^T \tag{9}$$

  with L a lower triangular matrix with positive entries along the diagonal
- so the linear system $Ax = b$ can be solved through

$$\begin{aligned} Ly &= b & (10)\\ L^T x &= y & (11) \end{aligned}$$

- the factorization is derived by equating corresponding entries of A with those of LL^T and generating them in the correct order
  - for example, in the 2×2 case

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

  yields

$$L_{11} = \sqrt{A_{11}} \quad L_{21} = A_{21}/L_{11} \quad L_{22} = \sqrt{A_{22} - L_{21}^2} \tag{13}$$

Computing the Cholesky factorization

Algorithm 6: CHOLESKY(A)

```
1 for k = 1 to n do
2     A_kk = sqrt(A_kk)
3     for i = k + 1 to n do
4         A_ik = A_ik / A_kk
5     for j = k + 1 to n do
6         for i = j to n do
7             A_ij = A_ij − A_ik A_jk
```

- note that
  - n square roots are required, all of positive numbers
  - only lower triangle of A is accessed, so the strict upper triangular part need not be stored
  - A becomes L in place
  - the algorithm is stable so no pivoting is required
- it takes roughly half the number of LU operations: approximately $n^3/6$ multiply-adds

Parallelizing Cholesky

- number fine grain tasks as (i,j) with $i,j = 1,\ldots,n$; each task
  - stores A_{ij}
  - computes and stores L_{ij}, if $i \geq j$
  - computes and stores L_{ji}, if $i < j$

  yielding a two dimensional array of n^2 tasks
- no need to compute and store the zero entries in the upper triangle

Communication patterns for parallel Cholesky

Algorithm 7: CHOLESKY(A, task=(i,j))

1 **for** $k = 1$ **to** $\min(i,j) - 1$ **do**
2 **recv** A_{kj}
3 **recv** A_{ik}
4 $A_{ij} = A_{ij} - A_{ik}A_{kj}$
5 **if** $i = j$ **then**
6 $A_{ii} = \sqrt{A_{ii}}$
7 **broadcast** A_{il} **to** tasks (k,i) and (i,k), $k = i+1,\ldots,n$
8 **if** $i < j$ **then**
9 **recv** A_{ii}
10 $A_{ij} = A_{ij}/A_{ii}$
11 **broadcast** A_{ij} **to** (k,j), $k = i+1,\ldots,n$
12 **if** $i > j$ **then**
13 **recv** A_{jj}
14 $A_{ij} = A_{ij}/A_{jj}$
15 **broadcast** A_{ij} **to** (i,k), $k = j+1,\ldots,n$

Coarsening

- strategies very similar to LU factorization
  - one dimensional by row or column
  - two dimensional blocks

  with column coarsening used most often in practice
- each choice of index in the outer loop yields different algorithm, named after the portion of the matrix that is updated by the basic operation in the inner loops
  - submatrix Cholesky: with k as the outer loop index, the inner loops perform a rank 1 update of the remaining unreduced submatrix, using the current column
  - column Cholesky: with j in the outer loop, inner loops compute the current column, using matrix-vector multiplies that accumulates the effects of previous columns
  - row Cholesky: with i in the outer loop, inner loops compute current row by solving a triangular system involving the previous rows

Cholesky memory access patterns