Parallel Numerical Algorithms

Chapter 3 – Dense Linear Systems
Section 3.2 – LU Factorization

Michael T. Heath and Edgar Solomonik

Department of Computer Science
University of Illinois at Urbana-Champaign

CS 554 / CSE 512

Outline

LU Factorization

- System of linear algebraic equations has form

$$\boxed{\boldsymbol{Ax} = \boldsymbol{b}}$$

where $\boldsymbol{A}$ is given $n \times n$ matrix, $\boldsymbol{b}$ is given n-vector, and $\boldsymbol{x}$ is unknown solution n-vector to be computed

- Direct method for solving general linear system is by computing *LU factorization*

$$\boxed{\boldsymbol{A} = \boldsymbol{LU}}$$

where $\boldsymbol{L}$ is unit lower triangular and $\boldsymbol{U}$ is upper triangular

LU Factorization

- System $\boldsymbol{Ax} = \boldsymbol{b}$ then becomes

$$\boldsymbol{LUx} = \boldsymbol{b}$$

- Solve lower triangular system

$$\boldsymbol{Ly} = \boldsymbol{b}$$

  by forward-substitution to obtain vector $\boldsymbol{y}$

- Finally, solve upper triangular system

$$\boldsymbol{Ux} = \boldsymbol{y}$$

  by back-substitution to obtain solution $\boldsymbol{x}$ to original system

Gaussian Elimination Algorithm

LU factorization can be computed by Gaussian elimination as follows, where $\boldsymbol{U}$ overwrites $\boldsymbol{A}$

for $k = 1$ **to** $n - 1$ { loop over columns }
    for $i = k + 1$ **to** n { compute multipliers for current column }
        $\ell_{ik} = a_{ik}/a_{kk}$
    end
    for $j = k + 1$ **to** n
        for $i = k + 1$ **to** n { apply transformation to remaining submatrix }
            $a_{ij} = a_{ij} - \ell_{ik}a_{kj}$
        end
    end
end

Gaussian Elimination Algorithm

- In general, row interchanges (pivoting) may be required to ensure existence of LU factorization and numerical stability of Gaussian elimination algorithm, but for simplicity we temporarily ignore this issue
- Gaussian elimination requires about $n^3/3$ paired additions and multiplications, so model serial time as

$$T_1 = \gamma\, n^3/3$$

  where γ is time required for multiply-add operation
- About $n^2/2$ divisions also required, but we ignore this lower-order term

Loop Orderings for Gaussian Elimination

- Gaussian elimination has general form of triple-nested loop in which entries of $\boldsymbol{L}$ and $\boldsymbol{U}$ overwrite those of $\boldsymbol{A}$

for ____________
    for ____________
        for ____________
            $a_{ij} = a_{ij} - (a_{ik}/a_{kk})\, a_{kj}$
        end
    end
end

- Indices i, j, and k of **for** loops can be taken in any order, for total of $3! = 6$ different ways of arranging loops

Loop Orderings for Gaussian Elimination

- Different loop orders have different memory access patterns, which may cause their performance to vary widely
- *Right-looking* orderings (loop over k is outermost) perform updates to the trailing matrix (update all a_{ij} for $i, j \geq k$) eagerly
- *Left-looking* orderings (loop over k is innermost) update the trailing matrix lazily (updates to a_{ij} done only when all entries $a_{i'j'}$ with $\min(i', j') < \min(i, j)$ have been updated)
- Right-looking ordering achieve better read-locality (the same divisor and outer-product vectors are reused)
- Left-looking ordering achieve better write-locality (entries of A may be changed in memory only once)

Gaussian Elimination Algorithm

- Right-looking form of Gaussian elimination

for $k = 1$ **to** $n - 1$
    for $i = k + 1$ **to** n
        $\ell_{ik} = a_{ik}/a_{kk}$
    end
    for $j = k + 1$ **to** n
        for $i = k + 1$ **to** n
            $a_{ij} = a_{ij} - \ell_{ik}\, a_{kj}$
        end
    end
end

- Multipliers ℓ_{ik} computed outside inner loop for greater efficiency

Parallel Algorithm

Partition

- For $i, j = 1, \ldots, n$, fine-grain task (i, j) stores a_{ij} and computes and stores

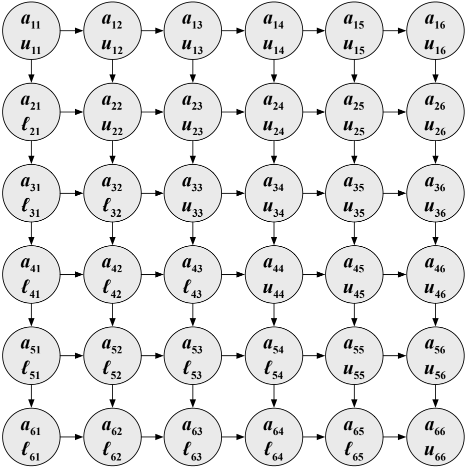
$$\begin{cases} u_{ij}, & \text{if } i \leq j \\ \ell_{ij}, & \text{if } i > j \end{cases}$$

yielding 2-D array of n^2 fine-grain tasks

Communicate

- Broadcast entries of A vertically to tasks below
- Broadcast entries of L horizontally to tasks to right

Fine-Grain Tasks and Communication



Fine-Grain Parallel Algorithm

for $k = 1$ **to** $\min(i, j) - 1$
  recv broadcast of a_{kj} from task (k, j) { vert bcast }
  recv broadcast of ℓ_{ik} from task (i, k) { horiz bcast }
  $a_{ij} = a_{ij} - \ell_{ik}\, a_{kj}$ { update entry }
end
if $i \le j$ **then**
  broadcast a_{ij} to tasks (k, j), $k = i + 1, \ldots, n$ { vert bcast }
else
  recv broadcast of a_{jj} from task (j, j) { vert bcast }
  $\ell_{ij} = a_{ij}/a_{jj}$ { multiplier }
  broadcast ℓ_{ij} to tasks (i, k), $k = j + 1, \ldots, n$ { horiz bcast }
end

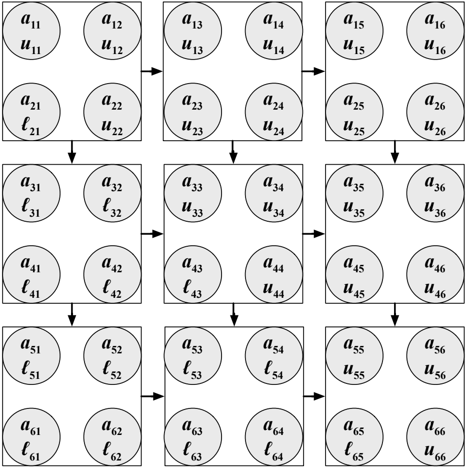
Agglomeration

Agglomerate

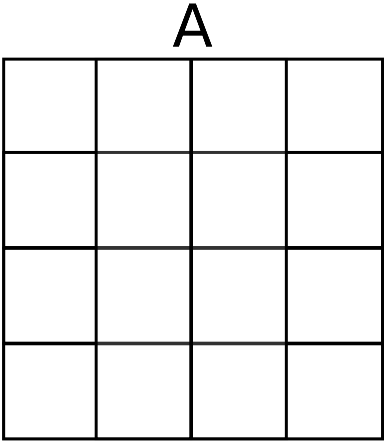
With $n \times n$ array of fine-grain tasks, natural strategies are

- 2-D: combine $k \times k$ subarray of fine-grain tasks to form each coarse-grain task, yielding $(n/k)^2$ coarse-grain tasks
- 1-D column: combine n fine-grain tasks in each column into coarse-grain task, yielding n coarse-grain tasks
- 1-D row: combine n fine-grain tasks in each row into coarse-grain task, yielding n coarse-grain tasks

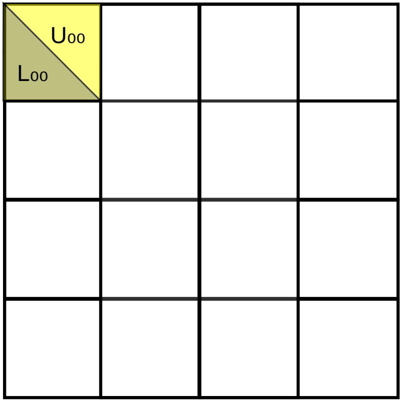
2-D Agglomeration



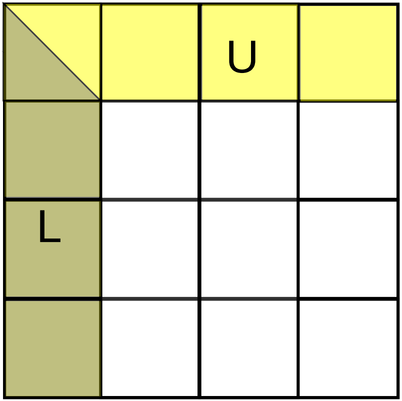
Blocked LU factorization



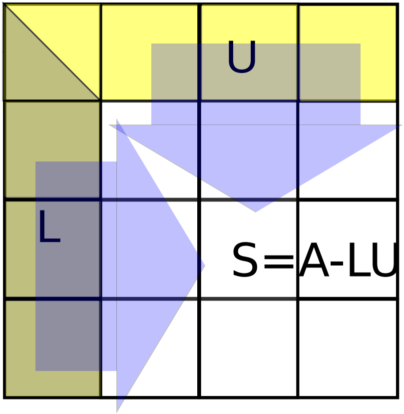
Blocked LU factorization



Blocked LU factorization



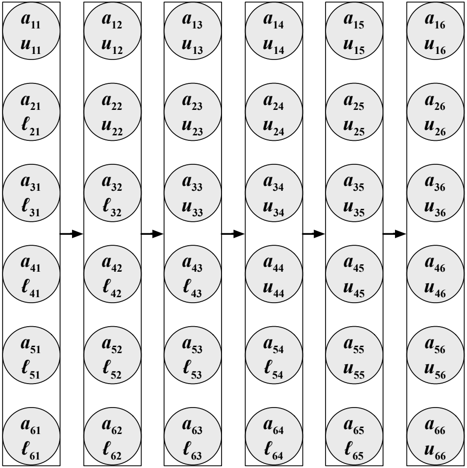
Blocked LU factorization



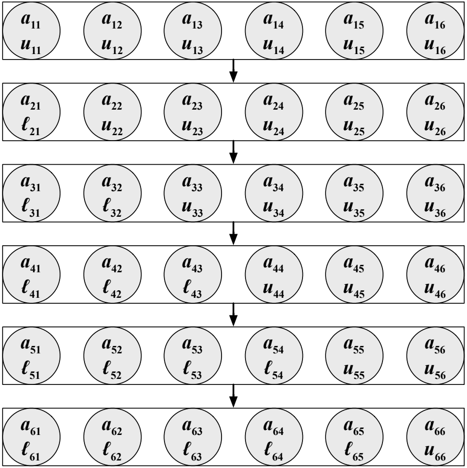
Coarse-Grain 2-D Parallel Algorithm

for $k = 1$ **to** $n - 1$
  broadcast $\{a_{kj} : j \in$ *mycols*, $j \geq k\}$ in processor column
  if $k \in$ *mycols* **then**
    for $i \in$ *myrows*, $i > k$
      $\ell_{ik} = a_{ik}/a_{kk}$ { multipliers }
    end
  end
  broadcast $\{\ell_{ik} : i \in$ *myrows*, $i > k\}$ in processor row
  for $j \in$ *mycols*, $j > k$
    for $i \in$ *myrows*, $i > k$,
      $a_{ij} = a_{ij} - \ell_{ik}\, a_{kj}$ { update }
    end
  end
end

1-D Column Agglomeration



1-D Row Agglomeration

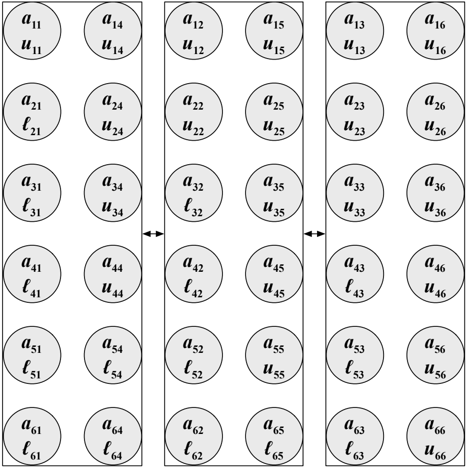


Mapping

Map

- 2-D: assign $(n/k)^2/p$ coarse-grain tasks to each of p processors treating target network as 2-D mesh, using
  - blocked mapping (aggregating into larger blocks)
  - cyclic mapping of blocks, yielding *block-cyclic* layout
- 1-D: assign n/p coarse-grain tasks to each of p processors treating target network as 1-D mesh, using
  - blocked mapping (aggregating into panels)
  - cyclic mapping of rows/cols, yielding row-cyclic or column-cyclic layout

1-D Column Agglomeration with Cyclic Mapping



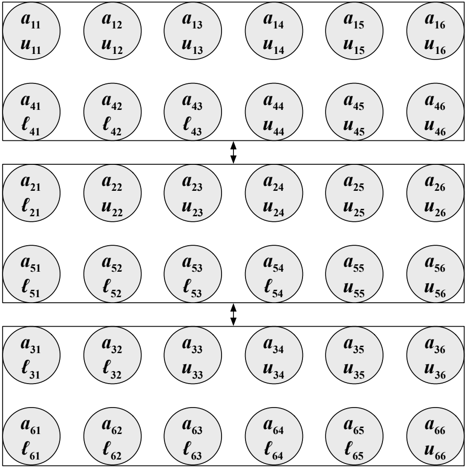
1-D Column Agglomeration

- Matrix rows need not be broadcast vertically, since any given column is contained entirely in only one process
- But there is no parallelism in computing multipliers or updating any given column
- Horizontal broadcasts still required to communicate multipliers for updating

Coarse-Grain 1-D Column Parallel Algorithm

for $k = 1$ **to** $n - 1$
  if $k \in$ *mycols* **then**
    for $i = k + 1$ **to** n
      $\ell_{ik} = a_{ik}/a_{kk}$ { multipliers }
    end
  end
  broadcast $\{\ell_{ik} : \; k < i \leq n\}$ { broadcast }
  for $j \in$ *mycols*, $j > k$
    for $i = k + 1$ **to** n
      $a_{ij} = a_{ij} - \ell_{ik}\, a_{kj}$ { update }
    end
  end
end

1-D Row Agglomeration with Cyclic Mapping



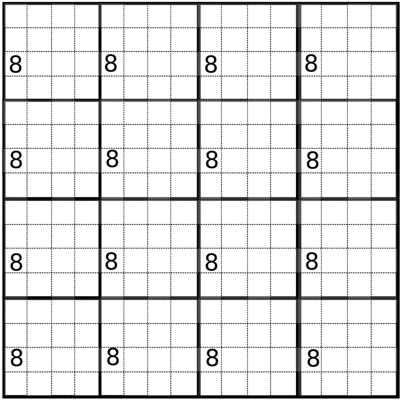
1-D Row Agglomeration

- Multipliers need not be broadcast horizontally, since any given matrix row is contained entirely in only one process
- But there is no parallelism in updating any given row
- Vertical broadcasts still required to communicate each row of matrix to processors below it for updating

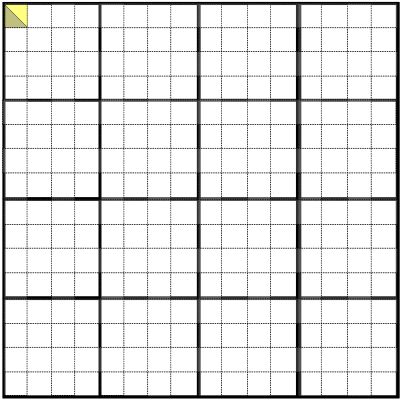
Coarse-Grain 1-D Row Parallel Algorithm

for $k = 1$ **to** $n - 1$
    broadcast $\{a_{kj} :\ k \le j \le n\}$ { broadcast }
    for $i \in$ *myrows*, $i > k$,
        $\ell_{ik} = a_{ik}/a_{kk}$ { multipliers }
    end
    for $j = k + 1$ **to** n
        for $i \in$ *myrows*, $i > k$,
            $a_{ij} = a_{ij} - \ell_{ik}\, a_{kj}$ { update }
        end
    end
end

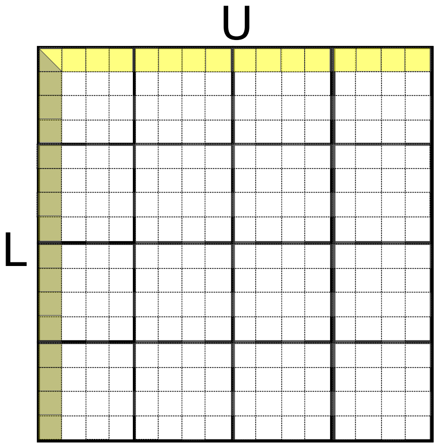
Block-Cyclic LU Factorization



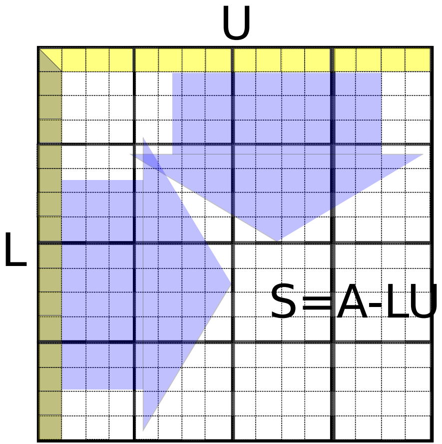
Block-Cyclic LU Factorization



Block-Cyclic LU Factorization



Block-Cyclic LU Factorization



Performance Enhancements

- Each processor becomes idle as soon as its last row and column are completed
- With block mapping, in which each processor holds contiguous block of rows and columns, some processors become idle long before overall computation is complete
- Block mapping also yields unbalanced load, as computing multipliers and updates requires successively less work with increasing row and column numbers
- Cyclic or reflection mapping improves both concurrency and load balance

Performance Enhancements

Performance can also be enhanced by overlapping communication and computation

- At step k, each processor completes updating its portion of remaining unreduced submatrix before moving on to step $k+1$
- Broadcast of each segment of row $k+1$, and computation and broadcast of each segment of multipliers for step $k+1$, could be initiated as soon as relevant segments of row $k+1$ and column $k+1$ have been updated by their owners, before completing remainder of their updating for step k
- This *look-ahead* strategy enables other processors to start working on next step earlier than they otherwise could

Execution Time for 1-D Agglomeration

- With 1-D column agglomeration, each processor factorizes panels of b columns, then broadcasts them to perform the trailing matrix update
- While work-efficient ($Q_p = \Theta(n^3)$), the concurrency in computational cost is constrained by panel factorization

$$F_p(n,b) = \Theta((n/b)nb^2 + n^3/p)$$

  so we need $b < n/p$ to maintain $F_p(n,b) = \Theta(n^3/p)$
- The overall execution time is given by

$$T_p(n,b) = \Theta\Big((n/b)T_p^{\text{bcast}}(nb) + \gamma F_p(n,b)\Big)$$

- It is generally minimized by picking $b = \Theta(n/p)$

$$T_p(n,b) = \Theta(\alpha p \log p + \beta n^2 + \gamma n^3/p)$$

Execution Time for 2-D Agglomeration

- With 2-D agglomeration and block-cyclic mapping, a processor factorizes a $b \times b$ diagonal block, broadcasts it to a column and row of processors, which update the panels and broadcast them to perform the trailing matrix updates
- The computational cost is constrained by lack of concurrency in the diagonal

$$F_p(n, b) = O(n^3/p + nb^2 + n^2b/\sqrt{p})$$

- The overall execution time is given by

$$T_p(n, b) = \Theta\Big((n/b)(T^{\mathrm{bcast}}_{\sqrt{p}}(b^2) + T^{\mathrm{bcast}}_{\sqrt{p}}(nb/\sqrt{p})) + \gamma F_p(n, b)\Big)$$

- It is generally minimized by picking $b = n/\sqrt{p}$

$$T_p(n) = T_p(n, n/\sqrt{p}) = \Theta(\alpha\sqrt{p}\log p + \beta n^2/\sqrt{p} + \gamma n^3/p)$$

Scalability for 2-D Agglomeration

- Cannon's algorithm for matrix multiplication (2-D agglomeration), could achieve strong scaling speed-up $p_s = O((\gamma/\alpha)n^2)$ and unconditional weak scaling
- The SUMMA algorithm, which was based on broadcasts, achieved slightly inferior scaling due to a $\Theta(\log(p))$ term on the latency cost
- The execution time of 2-D agglomeration for LU is the same as of SUMMA, so the efficiency and scaling characteristics are the same
- On the other hand, it is not possible to achieve strong scaling to $O((\gamma/\alpha)n^3/\log(n))$ processors as the depth of the usual LU algorithm is $D = n$, meaning the maximum speed-up is $p_s = \Theta(\max_p S_p) = O(Q_1/D) = O(n^2)$

Partial Pivoting

- Row ordering of A is irrelevant in system of linear equations
- Partial pivoting takes rows in order of largest entry in magnitude of leading column of remaining unreduced matrix
- This choice ensures that multipliers do not exceed 1 in magnitude, which reduces amplification of rounding errors
- In general, partial pivoting is required to ensure existence and numerical stability of LU factorization

Partial Pivoting

- Partial pivoting yields factorization of form

$$\boxed{PA = LU}$$

where P is permutation matrix

- If $PA = LU$, then system $Ax = b$ becomes

$$PAx = LUx = Pb$$

which can be solved by forward-substitution in lower triangular system $Ly = Pb$, followed by back-substitution in upper triangular system $Ux = y$

Parallel Partial Pivoting

- Partial pivoting complicates parallel implementation of Gaussian elimination and significantly affects potential performance
- With 2-D algorithm, pivot search is parallel but requires communication within processor column ($S = \Omega(n \log(p))$) and inhibits overlap
- With 1-D column algorithm, pivot search requires no communication but is purely serial
- Once pivot is found, index of pivot row must be communicated to other processors, and rows must be explicitly or implicitly interchanged in each process

Alternatives to Partial Pivoting

- Because of negative effects of partial pivoting on parallel performance, various alternatives have been proposed that limit pivot search
  - tournament pivoting (perform tree of partial pivoting on different subsets of matrix rows, selecting b at a time)
  - threshold pivoting (use local rows as pivots if the diagonal entries are within threshold of column norm)
  - pairwise pivoting (eliminate $n(n-1)/2$ entries by as many 2-by-2 transformations $\boldsymbol{L}_i\boldsymbol{P}_i$, where $\boldsymbol{L}_i$ is unit-lower triangular and $\boldsymbol{P}_i$ is a permutation matrix, applied to appropriate row pairs)
- Stability generally slightly worse in theory and for particularly hard test-cases
- Better stability without worrying about pivoting may be achieved via QR factorization

Communication vs. Memory Tradeoff

- If explicit replication of storage is allowed, then lower communication volume is possible
- As with matrix multiplication, algorithms that leverage all available memory to reduce communication cost to the maximum extent possible
- If sufficient memory is avaiable, then these algorithms can achieve provably optimal communication

References

- J. W. Demmel, M. T. Heath, and H. A. van der Vorst, Parallel numerical linear algebra, *Acta Numerica* 2:111-197, 1993
- G. A. Geist and C. H. Romine, LU factorization algorithms on distributed-memory multiprocessor architectures, *SIAM J. Sci. Stat. Comput.* 9:639-649, 1988
- L. Grigori, J. Demmel, and H. Xiang, CALU: A communication optimal LU factorization algorithm, *SIAM J. Matrix Anal. Appl.* 32:1317-1350, 2011
- B. A. Hendrickson and D. E. Womble, The torus-wrap mapping for dense matrix calculations on massively parallel computers, *SIAM J. Sci. Stat. Comput.* 15:1201-1226, 1994

References

- J. M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, 1988
- J. M. Ortega and C. H. Romine, The ijk forms of factorization methods II: parallel systems, *Parallel Comput.* 7:149-162, 1988
- Y. Robert, *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*, Wiley, 1990
- S. A. Vavasis, Gaussian elimination with pivoting is P-complete, *SIAM J. Disc. Math.* 2:413-423, 1989