ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

Dense matrix problems

- we'll take a look at
  - inner and outer products of two vector
  - matrix-vector and matrix-matrix multiplication
  - *LU* factorization and Cholesky decomposition
  - QR factorization
  - computing eigenvalues and eigenvectors
  - fast Fourier transforms
- when solving a problem of size n on p processors, we will assume
  - that p, and occasionally $\sqrt{p}$ divides n
  - that p is a perfect square, when forming two-dimensional process grids
  - matrices are $n \times n$ – square, not rectangular
  - we are memory constrained and data replication must be minimized
- these problems have been studied extensively and form the core of scientific computing on parallel machines
  - excellent implementations available
  - interest has been revived due to the expected disruption by multi-core architectures

Vector inner product

- the inner product of two n-vectors x, y is given by

$$x^T y = \sum_{i=1}^{n} x_i y_i \tag{1}$$

  which requires n multiplications and $n-1$ additions
- parallelization strategy:
  - n fine grain tasks, numbered $i = 1, \ldots, n$, that store x_i and y_i, and compute $x_i y_i$
  - communication is a sum reduction over n fine grain tasks
  - coarsening is achieved by coalescing n/p tasks together, assuming that each process can accommodate the data storage requirements
  - and mapping each coarse grain task to a process

Vector outer product

- the outer product of two n-vectors x and y is the $n \times n$ matrix A given by
$$A_{ij} = x_i y_j \tag{2}$$
which requires n^2 multiplications
- parallelization strategies are determined by the storage requirements
  - build a two-dimensional grid of n^2 fine grain tasks numbered (i, j), with $i, j = 1, \ldots, n$; each one computes $x_i y_j$
  - assuming no data replication is allowed
    - let task $(i, 1)$ store x_i and task $(1, i)$ store y_i
    - or, let task (i, i) store both x_i and y_i
  - either way, the task that owns each element must broadcast it to the other tasks: x_i along the i^{th} task row, y_j along the j^{th} task column
  - coarsening to p tasks can be accomplished by
    - combining n/p rows or columns
    - forming $(n/\sqrt{p}) \times (n/\sqrt{p})$ grid of fine grain tasks
  - and each coarse grain task can be assigned to a process
- either way, naïve broadcasting of the components of x and y would require as much total memory as replication
  - storage can be reduced by circulating portions of x and y through the tasks, with each task using the available portion and passing it on

The product of a matrix with a vector

- given an $n \times n$ matrix A and an n-vector x, the matrix vector product yields an n-vector y whose components are given by

$$y_i = \sum_{j=1}^{n} A_{ij} x_j \tag{3}$$

  requiring a total of n^2 multiply-add operations
- once again, the parallelization strategy is determined by how the data is distributed among fine grain tasks
  - build a two-dimensional grid of n^2 fine grain tasks numbered (i, j), with $i, j = 1, \ldots, n$; each one computes $A_{ij} x_j$
  - task (i, j) has $A_{i,j}$, but if no data replication is allowed
    - let task $(i, 1)$ store x_i and task $(1, i)$ store y_i
    - or, let task (i, i) store both x_i and y_i
  - the task that owns x_j must broadcast it along the j^{th} task row, and y_i is formed by sum reduction along the i^{th} task column
  - coarsening into p tasks can be accomplished by combining n/p rows/columns, or by forming $(n/\sqrt{p}) \times (n/\sqrt{p})$ blocks
  - and each coarse grain task can be assigned to a process

Coarsening along rows or columns

- for one-dimensional coarsening into n/p task rows
  - if x is stored in one task, it must be broadcast to all others
  - if x is distributed among tasks, with n/p components per task, then multiple broadcasts are required
  - each task computes the inner product of its n/p rows of A with the entire x to produce n/p components of y
- for one-dimensional coarsening into n/p task columns
  - n/p components of x are distributed among the tasks
  - each task computes the linear combination of its n/p columns with coefficients from its copy of x
  - since the right parts of x are already available, no communication is required
  - y is generated by a sum reduction across tasks
- these two are *duals* of each other
  - row coarsening begins with broadcast, followed by communication-free inner products
  - column coarsening begins with communication-free linear combinations, follows by a reduction

Two dimensional coarsening

- for two dimensional coarsening, we form $(n/\sqrt{p}) \times (n/\sqrt{p})$ blocks of fine grain task
  - each one holding a $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of A
  - with components of x distributed either across one task row, or along the diagonal, n/p components per task
- the algorithm combines the features of row/column coarsening
  - components of x are broadcast along task columns
  - each task performs n^2/p multiplications locally and sums $n/\sqrt{p}$ sets of products
  - sum reductions along task rows produce the components of y by combining the component products

Matrix multiplication

- the product of two $n \times n$ matrices A and B is an $n \times n$ matrix C given by

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \tag{4}$$

  where each one of n^2 entries requires n multiply-adds for a total of n^3 operations
- matrix multiplication can be viewed as
  - n^2 inner products
  - the sum of n outer products
  - n matrix vector products
- each one produces a parallel algorithm for matrix multiplication
- but we'll explore a direct solution instead

Partitioning and communication patterns

- we build a three dimensional array of n^3 fine grain tasks
  - with $i, j, k = 1, \ldots, n$, let task (i, j, k) be responsible for computing the product $A_{ij}B_{jk}$
  - assuming no data replication, we have to distribute the data for A and B among $2n^2$ tasks
  - suppose that task (i, j, j) holds $A_{i,j}$ and task (i, j, i) holds $B_{i,j}$
  - we will refer to tasks along i and j as task rows and columns
  - and tasks along k as *layers*
- the communication requirements among tasks are satisfied if we
  - broadcast the entries of the k^{th} column of A from task (i, j, j) to each task row in the k^{th} layer
  - broadcast the entries of the k^{th} row of B from task (i, j, i) to each task column of the k^{th} layer
  - form the result C_{ij} by the sum reduction of the values held by all the tasks layers k

Coarsening

- there are four natural ways to coarsen our $n \times n \times n$ fine grain tasks into p coarse grain tasks
  - by task rows: combine the $(n/p) \times n \times n$ tasks along a given task row
  - by task columns: combine the $n \times (n/p) \times n$ tasks along a given task column
  - partition the layers in a two dimensional grid by combining $(n/\sqrt{p}) \times (n/\sqrt{p}) \times n$ fine grain tasks
  - using three dimensional blocks by combining $(n/\sqrt[3]{p}) \times (n/\sqrt[3]{p}) \times (n/\sqrt[3]{p})$ tasks
- the two one dimensional coarsening strategies are similar
  - for row coarsening
    - each task needs only the part of A it already has, but needs all B entries
    - so global communication is required to broadcast the n^2/p entries of B held by each task
  - conversely, for column coarsening
    - each task needs only the parts of B that it already has, but it needs all of A
    - so global communication is required to broadcast the n^2/p entries of A held by each task
  - if accumulating A or B on each processor is not feasible, tasks can circulate portions of the array in a ring

Coarsening using a two dimensional grid

- block matrix multiplication has the same overall form as actual product, with scalar operations replaced by the matrix product of blocks!
- you should verify that
$$C_{ij} = \sum_{k=1}^{\sqrt{p}} A_{ik} B_{kj} \tag{5}$$
for $i, j = 1, \ldots, \sqrt{p}$
- assume that task (i, j) has local access to block A_{ij} and B_{ij} and computes block C_{ij} of the result
- this requires all blocks A_{ik} and B_{kj} for $k = 1, \ldots, \sqrt{p}$ to be communicated
  - first, a global broadcast of A blocks across each task row
  - followed by a global broadcast of B blocks across each task column
- memory requirements can be addressed by either of the following:
  - broadcast blocks of A across rows while circulating blocks of B across columns in lock step, so that they arrive at a given task at the same time
  - circulate blocks of A horizontally and blocks of B vertically, after an initial circular shift, so that blocks meet at a given task at the right time

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
$$Ax = b \tag{6}$$
- 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
$$A = LU \tag{7}$$
known as LU factorization
- Eq. 6 becomes
$$LUx = b \tag{8}$$
which we can now solve in two simpler steps
$$\begin{aligned} Ly &= b & (9) \\ Ux &= y & (10) \end{aligned}$$
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 $i, j = 1, \dots, n$; each task
    - stores A_{ij}
    - computes and stores U_{ij}, if $i \le j$
    - 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 $k \in myrows$ then
3          broadcast $\{A_{kj} : k \le j \le n\}$
4      else
5          recv $\{A_{kj} : k \le j \le n\}$
6      for $i \in myrows, i > k$ do
7          $L_{ik} = A_{ik}/A_{kk}$
8      for $j = k + 1$ to $n$ do
9          for $i \in myrows, i > k$ 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 $k \in mycolumns$ then
3          for $i = k + 1$ to $n$ do
4              $L_{ik} = A_{ik}/A_{kk}$
5          broadcast $\{L_{ik} : k < i \leq n\}$
6      else
7          recv $\{L_{ik} : k < i \leq n\}$
8      for $i \in mycolumns, j > k$ 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             $A + ij = A_{ij} - L_{ik}A_{kj}$
```

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

$$PA = LU \tag{11}$$

which implies that the solution x is obtained through

$$\begin{aligned} Ly &= Pb \qquad (12) \\ Ux &= y \qquad (13) \end{aligned}$$

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
$$A = LL^T \tag{14}$$
with L a lower triangular matrix with positive entries along the diagonal
- so the linear system $Ax = b$ can be solved through
$$\begin{aligned} Ly &= b & (15) \\ L^T x &= y & (16) \end{aligned}$$
- 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
$$\begin{bmatrix} A_{11} & A_{21} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11} & L_{21} \\ 0 & L_{22} \end{bmatrix} \tag{17}$$
yields
$$L_{11} = \sqrt{A_{11}} \quad L_{21} = A_{21}/L_{11} \quad L_{22} = \sqrt{A_{22} - L_{21}^2} \tag{18}$$

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))

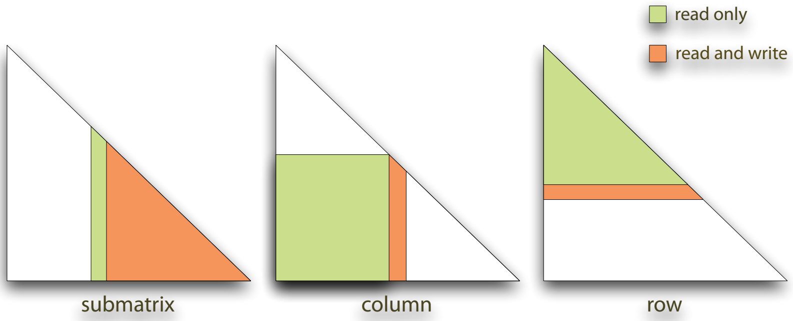
```
1  for $k = 1$ to $\min(i,j) - 1$ do
2      recv $A_{kj}$
3      recv $A_{ik}$
4      $A_{ij} = A_{ij} - A_{ik}A_{kj}$
5  if $i = j$ then
6      $A_{ii} = \sqrt{A_{ii}}$
7      broadcast $A_{il}$ to tasks $(k,i)$ and $(i,k)$, $k = i+1, \ldots, n$
8  if $i < j$ then
9      recv $A_{ii}$
10     $A_{ij}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


🟩 read only
🟧 read and write
submatrix
column
row