Numerical Algorithms

Jingke Li

Portland State University

Basic Numerical Algorithms

- *Dense-Matrix Algorithms:*
  - Matrix-vector multiplication
  - Matrix-matrix multiplication
  - Gaussian elimination
- *Sparse-Matrix Algorithms:*
  - Jacobi relaxation
  - Gauss-Seidel
  - Multi-grid method

For all these algorithms, parallelizing them to run on a shared-memory system is trivial. They all have easily-parallelizable loops.

Matrix-Vector Multiplication

Compute $y = Ax$ (A is an $n \times n$ matrix, x, y are $n \times 1$ vectors)

Sequential Algorithm:

```
void matrix_vector(int n, double a[n][n], double x[n], double y[n]) {
  int i, j;
  for (i = 0; i < n; i++) {
    y[i] = 0;
    for (j = 0; j < n; j++)
      y[i] += a[i][j] * x[j];
  }
}
```

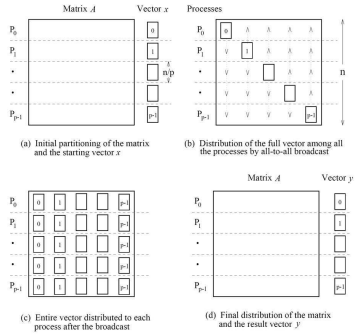
Complexity: $T = O(n^2)$

Parallelization (for Distributed-Memory Systems):

- Need to decide partition and communication.

Distributed-Memory Matrix-Vector Multiplication

- 1D Partition (row-wise):

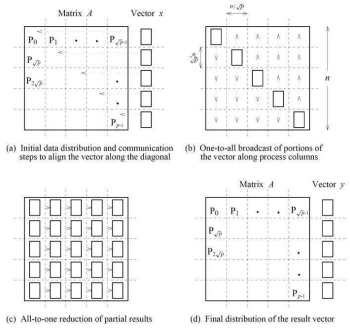


(a) Initial partitioning of the matrix and the starting vector x
(b) Distribution of the full vector among all the processes by all-to-all broadcast
(c) Entire vector distributed to each process after the broadcast
(d) Final distribution of the matrix and the result vector y

Complexity: $T_p \approx \frac{n^2}{p} + t_s \log p + t_w n$

Distributed-Memory Matrix-Vector Multiplication (cont.)

- 2D Partition:



(a) Initial data distribution and communication steps to align the vector along the diagonal
(b) One-to-all broadcast of portions of the vector along process columns
(c) All-to-one reduction of partial results
(d) Final distribution of the result vector

Complexity: $T_p \approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p$

Matrix-Matrix Multiplication

Compute $C = AB$ (A, B, C are $n \times n$ dense matrices)

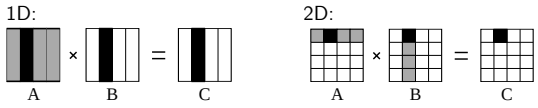
Sequential Algorithm:

```
void matrix_mult(int n, double a[n][n], double b[n][n], double c[n][n]) {
  int i, j, k;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      c[i][j] = 0;
      for (k = 0; k < n; k++)
        c[i][j] +=  a[i][k] * b[k][j];
    }
  }
}
```

Complexity: $T = O(n^3)$ (Best is $O(n^{2.8})$ — Strassen's Algorithm)

Distributed-Memory Matrix-Matrix Multiplication

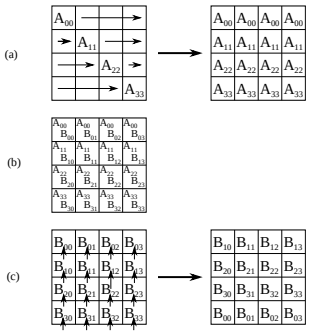
- *Partition Choices:*



- *Communication Choices:*
  - A single broadcast — The needed data is broadcast to all destinations once for all.
    - A single step, but needs a lot of buffer storage.
  - Multiple shifts — The needed data is shifted towards its destination one step at a time.
    - Only local communication and very little buffer space, but takes multiple steps for data to reach destination.

A Simple 2D Matrix-Mult Algorithm

- Processors are arranged in a logical $\sqrt{p} \times \sqrt{p}$ 2D topology.
- Each processor gets a block of $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ block of $A, B, \& C$.
- Perform all-to-all broadcasts within processor rows for A's blocks.
- Take $\sqrt{p}$ iterations of the following steps:
  1. Multiply A's block with B's block to form C's block.
  2. Shift B's blocks within processor columns.



Complexity: $T_p \approx \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}$

Cannon's Matrix-Mult Algorithm

Memory efficient variant of the simple algorithm.

Key Idea: Replace the standard loop

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} \times B_{k,j}$$

with a skewed loop

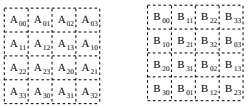
$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k) \bmod \sqrt{p}} \times B_{(i+j+k) \bmod \sqrt{p},j}$$

Communication: Multiple shifts for both arrays A and B.

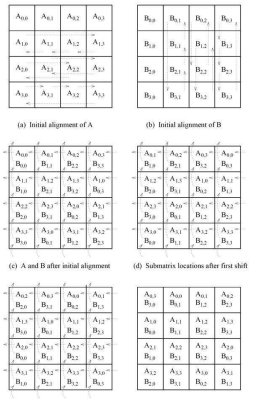
Complexity: $T_p \approx \frac{n^3}{p} + 2t_s\sqrt{p} + 2t_w \frac{n^2}{\sqrt{p}}$

Cannon's Matrix-Mult Algorithm (cont.)

Initial Skewing of A and B:



Iterations:
- Multiply A and B blocks.
- Shift A blocks towards left.
- Shift B blocks upwards.



Gaussian Elimination

A well-known algorithm for solving a linear system $\mathbf{Ax} = \mathbf{b}$.

Idea:
- First reduce $\mathbf{Ax} = \mathbf{b}$ to an upper triangular system $\mathbf{Tx} = \mathbf{c}$.
- Then use back substitution to solve for $\mathbf{x}$.

The two needed operations are:
- Interchange any two rows (this simply reorders the n equations).
- Replace any row by a linear combination of itself and another row.

Gaussian Elimination Example

- The initial linear system:

```
2.0 x1 + 1.0 x2 + 3.0 x3 + 4.0 x4 =  29.0
5.0 x1 + 2.0 x2 + 0.0 x3 + 1.0 x4 =  13.0
1.0 x1 + 0.0 x2 + 0.0 x3 + 1.0 x4 =   5.0
3.0 x1 + 1.0 x2 + 1.0 x3 + 0.0 x4 =   8.0
```

- Matrix form and triangulating:

```
2.0  1.0  3.0  4.0  29.0          2.0  1.0  3.0  4.0  29.0
0.0 -0.5 -7.5 -9.0 -59.5   ⇒     0.0 -0.5 -7.5 -9.0 -59.5
0.0 -0.5 -1.5 -1.0  -9.5          0.0  0.0  6.0  8.0  50.0
0.0 -0.5 -3.5 -6.0 -35.5          0.0  0.0  0.0 -2.3  -9.3
```

- Back to linear system form:

```
2.0 x1 + 1.0 x2 + 3.0 x3 + 4.0 x4 =  29.0
       - 0.5 x2 - 7.5 x3 - 9.0 x4 = -59.5
                  6.0 x3 + 8.0 x4 =  50.0
                          -2.3 x4 =  -9.3
```

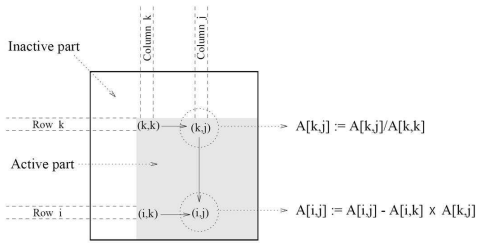
- Use back-substitution to solve: `x1=1, x2=2, x3=3, x4=4`

Naive Gaussian Elimination

```
void gaussian_naive(int n, double a[n][n], double b[n], double x[n]) {
  int i, j, k;
  for (k = 0; k < n; k++) {
    for (j = k+1; j < n; j++) {
      a[k][j] = a[k][j] / a[k][k];
    }
    x[k] = b[k] / a[k][k];
    for (i = k+1; i < n; i++) {
      for (j = k+1; j < n; j++) {
        a[i][j] -= a[i][k] * a[k][j];
      }
      b[i] = a[i][k] * x[k];
      a[i][k] = 0;
    }
  }
}
```

Complexity: $T = O(n^3)$

Naive Gaussian Elimination (cont.)



Gaussian Elimination with Partial-Pivoting

The naive GE algorithm runs into trouble when a pivot element is zero. (In fact, it runs into trouble whenever a pivot element is "small.")

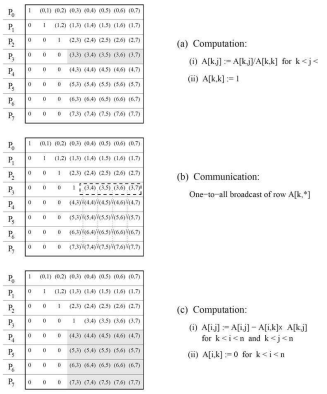
Partial-Pivoting — Search the column below the diagonal and find the largest element in absolute magnitude; perform row interchange to bring this element to the diagonal.

Example:

```
Initial:                    => Pivoting:
  2.0  1.0  3.0  4.0  29.0      5.0  2.0  0.0  1.0  13.0
  1.0  0.0  0.0  1.0   5.0      1.0  0.0  0.0  1.0   5.0
  3.0  1.0  1.0  0.0   8.0      3.0  1.0  1.0  0.0   8.0
  5.0  2.0  0.0  1.0  13.0      2.0  1.0  3.0  4.0  29.0

=> Elimination:             =>
  5.0  2.0  0.0  1.0  13.0
  0.0 -0.4  0.0  0.8   2.4     Continue for other columns ...
  0.0 -0.2  1.0 -0.6   0.2
  0.0  0.2  3.0  3.6  23.8
```


Parallel GE with 1-D Partitioning



Parallel GE with 1-D Partitioning

- Assign one row to each process.
- Execute the outer loop sequentially.
- At iteration k+1, process P_k either broadcast or shift its row to processes $P_{k+1}, \ldots, P_{n-1}$.
- Each process performs local computation.

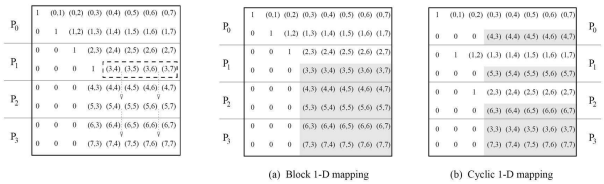
Complexity:

$$T_p = \frac{3}{2}n(n-1) + t_s n \log n + \frac{1}{2}t_w n(n-1)\log n$$

Improvements

Problems with the Simple Algorithm — The partition is fine-grained, and towards the end, the active region of the matrix is shrinking towards lower right corner; implies that processes fall idle.

Solutions: block and cyclic partitions.



(a) Block 1-D mapping (b) Cyclic 1-D mapping

Parallel GE with Partial-Pivoting

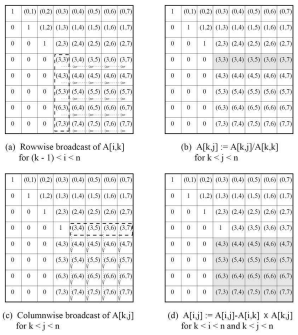
- Distribute the matrix as complete rows; each process stores approximately n/p rows of the matrix.
- Processes collectively decide on which two rows need to be swapped to do the partial pivoting.
- Two processes do a `send/recv` to each other to do the swap. (Alternatively, we can keep track of which row is which through an indirection array.)
- The pivot row is broadcast to all processes.
- In each process, we loop over those rows below the pivot row and apply the elimination step (an `saxby()` operation).
- Repeat until we hit bottom.

Parallel GE with 2-D Partitioning

Each processor gets a 2D block of the matrix.

Steps:

- Broadcast the "active" column along the rows.
- Prepare the pivot row concurrently.
- Broadcast the pivot row along the columns.
- Perform the elimination step concurrently.



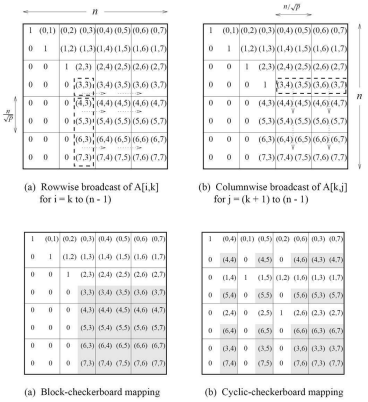
(a) Rowwise broadcast of A[i,k] for (k - 1) < i < n

(b) A[k,j] := A[k,j]/A[k,k] for k < j < n

(c) Columnwise broadcast of A[k,j] for k < j < n

(d) A[i,j] := A[i,j]-A[i,k] × A[k,j] for k < i < n and k < j < n

Block and Cyclic 2-D Partitions



(a) Rowwise broadcast of A[i,k] for i = k to (n - 1)

(b) Columnwise broadcast of A[k,j] for j = (k + 1) to (n - 1)

(a) Block-checkerboard mapping

(b) Cyclic-checkerboard mapping

Solving Large, Sparse Linear Systems

Large sparse matrices often appear in scientific or engineering applications when solving partial differential equations. As an example, consider the Laplace Equation:

$$\boxed{\phi_{i,j} = \frac{1}{4}(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}), \quad 0 < i,j < n}$$

It describes a computation over an $n \times n$ mesh. $\phi_{i,j}$ deotes the value at the mesh point $[i,j]$.

```
# # # ... # # # ..#
# .  . ... .  .  . ..#
.. .  . ... ... ... ...
# .  . ... .  n  . ..#
# .  . ... w  x  e ..#
# .  . ... .  s  . ..#
.. .  . ... ... ... ...
# .  . ... .  .  . ..#
# # # ... # # # ..#
```

- The value at point $[i,j]$ is related to the values of its four neighbors.
- The values at the four boarder lines are typically fixed.

Solving the Laplace Equation

The Laplace Equation is a linear system with n^2 unknowns. It can be turned into the standard $\mathbf{Ax = b}$ form:

Let $I = i \cdot n + j$, then the system becomes:

$$\phi_{I+n} + \phi_{I-n} + \phi_{I+1} + \phi_{I-1} - 4\phi_I = 0, \quad 0 < I < n^2$$

which can also be expressed in matrix form:

```
-4  1  0  0 ... 1  0 ...
 1 -4  1  0 ... 0  1 ...
 0  1 -4  1 ... 0  0 ...
 0  0  1 -4 ... .. .. ..
... ... ... ... ... .. ..
 1  0  0 .. ... .. .. ..
 0  1  0 .. ... .. .. ..
... ... ... ... .. .. ..
... ... ... ... .. .. ..
... ... ... ... .. .. ..
```

- The matrix is very large, yet sparse.
- It can be solved by using Gaussian Elimination, but the cost would be very high: $O(n^6)$.

A better approach is to directly solve the system over the original $n \times n$ mesh domain.

Jacobi Relaxation Algorithm

It is an iterative algorithm:

- Cycle through the mesh points, compute a new value for each point using the average of the four-neighboring "old" values.

$$\boxed{\phi^{(t+1)}_{i,j} = \frac{1}{4}(\phi^{(t)}_{i+1,j} + \phi^{(t)}_{i-1,j} + \phi^{(t)}_{i,j+1} + \phi^{(t)}_{i,j-1}), \quad 0 < i,j < N}$$

- Once all the new values are found, replace old values with new ones.
- Repeat until the difference between old and new is small enough.

Parallelization:

- For the distributed-memory case, simply partition the mesh in both dimensions; each processor will handle a $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ sub-mesh. Communications are only required on the "peripheral" of the mesh region in each processor.

Jacobi Relaxation Algorithm (cont.)

```
int jacobi(int n, double x[n][n], double epsilon) {
  double xnew[n][n], delta;
  int i, j, cnt = 0;
  do {
    cnt++;
    delta = 0.0;
    for (i = 1; i < n-1; i++) {
      for (j = 1; j < n-1; j++) {
        xnew[i][j] = (x[i-1][j] + x[i][j-1]
                    + x[i+1][j] + x[i][j+1]) / 4.0;
        delta = fmax(delta, fabs(xnew[i][j] - x[i][j]));
      }
    }
    for (i = 1; i < n-1; i++) {
      for (j = 1; j < n-1; j++) {
        x[i][j] = xnew[i][j];
      }
    }
  } while (delta > epsilon);
  return cnt;
}
```


Gauss-Seidel Algorithm

While simple, the Jacobi relaxation algorithm has two drawbacks, (1) It needs buffers to store old mesh point values; and (2) Its convergence speed can be slow.

Gauss-Seidel algorithm is an improvement over the Jacobi algorithm. It's like Jacobi, except that mesh points are updated "in-place," overwriting the old value.

The order of mesh point updates does not really matter. Here are two possible orders:

$$\phi_{i,j}^{(t+1)} = \frac{1}{4}(\phi_{i+1,j}^{(t)} + \phi_{i-1,j}^{(t+1)} + \phi_{i,j+1}^{(t)} + \phi_{i,j-1}^{(t+1)}), \qquad 0 < i,j < N$$

$$\phi_{i,j}^{(t+1)} = \frac{1}{4}(\phi_{i+1,j}^{(t+1)} + \phi_{i-1,j}^{(t)} + \phi_{i,j+1}^{(t+1)} + \phi_{i,j-1}^{(t)}), \qquad 0 < i,j < N$$

- Gauss-Seidel algorithm does not need buffers and has a better convergence speed (since newer values are used in all updates).

Parallelizing Gauss-Seidel

Due to the "in-place" updates, race condition may happen among multiple reads and a single write to the same memory location. But this is not really a problem for an iterative algorithm — using an old value vs. using a new value affects only the convergency rate.

The following are the common parallel approaches:

- *Natural index ordering* (allow race condition)
- *Red-black ordering* (avoid race condition)
  - Denote alternating points as "red" and "black".
  - Repeat following two steps until convergence:
    . update all red points (in any order)
    . update all black points (in any order)

```
r  b  r  b  r  b
b  r  b  r  b  r
r  b  r  b  r  b
b  r  b  r  b  r
r  b  r  b  r  b
b  r  b  r  b  r
```

- *Wavefront ordering* (avoid race condition)
  - Proceed along diagonal line.
  - Repeat until success.

```
1  2  3  4  ..  ..
2  3  4  5  ..  ..
3  4  5  ..  ..  ..
4  5  ..  ..  ..  ..
5  ..  ..  ..  ..  ..
.  ..  ..  ..  ..  ..
```


Multi-Grid Methods

Observations — Iterative algorithms converge to solutions faster on coarser grids than on finer grids. Iterative algorithms converge quicker if the initial approx. of the values of the variables are good.
Algorithm:

- Begin with the original problem, where the solution is defined (and desired) on an $n \times n$ mesh.
- (*Projection*) Coarsen the problem by several powers of 2: $n \to n/2^m$. Boundary values need to be averaged down to the coarser mesh.
- (*Relaxation*) Solve the coarse version problem using any iterative solver. It should go fast since the mesh is small.
- (*Interpolation*) Boost up the problem by a factor of 2, interpolating field points. What was one mesh point turns into 4 mesh points.
- Re-run relaxation on this problem.
- Repeat *Relaxation/Interpolation* steps until back to original problem.