pgeos

Numerical Algorithms

Jingke Li

Portland State University

Simple for sequential implementation, but used heavily in scientific world.

Basic Numerical Algorithms

All numerical algorithms are matrix based (2D arrays)

Two groups algorithms

- *Dense-Matrix Algorithms:* (most of the entries in a matrix are not zero)
  - Matrix-vector multiplication
  - Matrix-matrix multiplication
  - Gaussian elimination
- *Sparse-Matrix Algorithms:* (most of the entries are 0 > 90%)
  - 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.

These are almost all algorithms used in numerical computation problems

They should be very efficient therefore parallelization comes in handy.

Most of them are EP (embarrassingly parallel), very easy to do in shared memory systems.

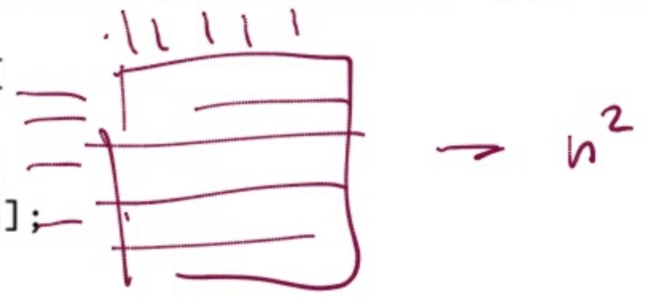
MP is harder (how to minimize message passing?)

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: $T = O(n^2)$

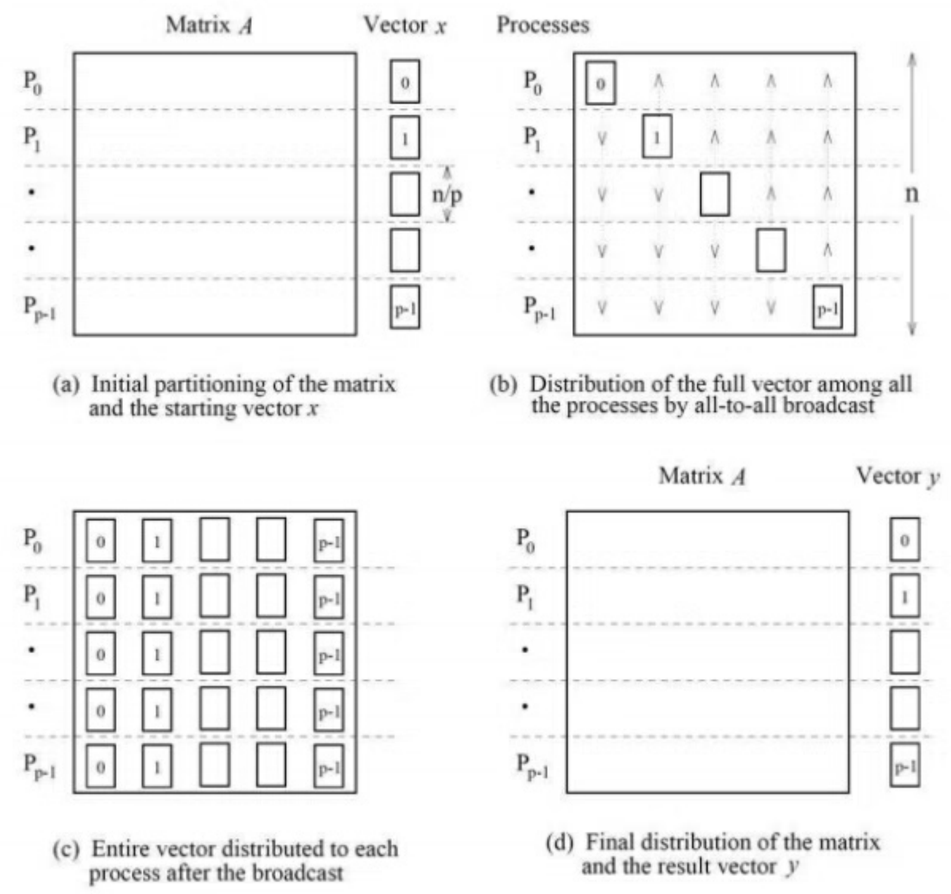
Parallelization (for Distributed-Memory Systems): (Message passing systems)

- Need to decide partition and communication.

There is no interference (all computation can be done concurrently)

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

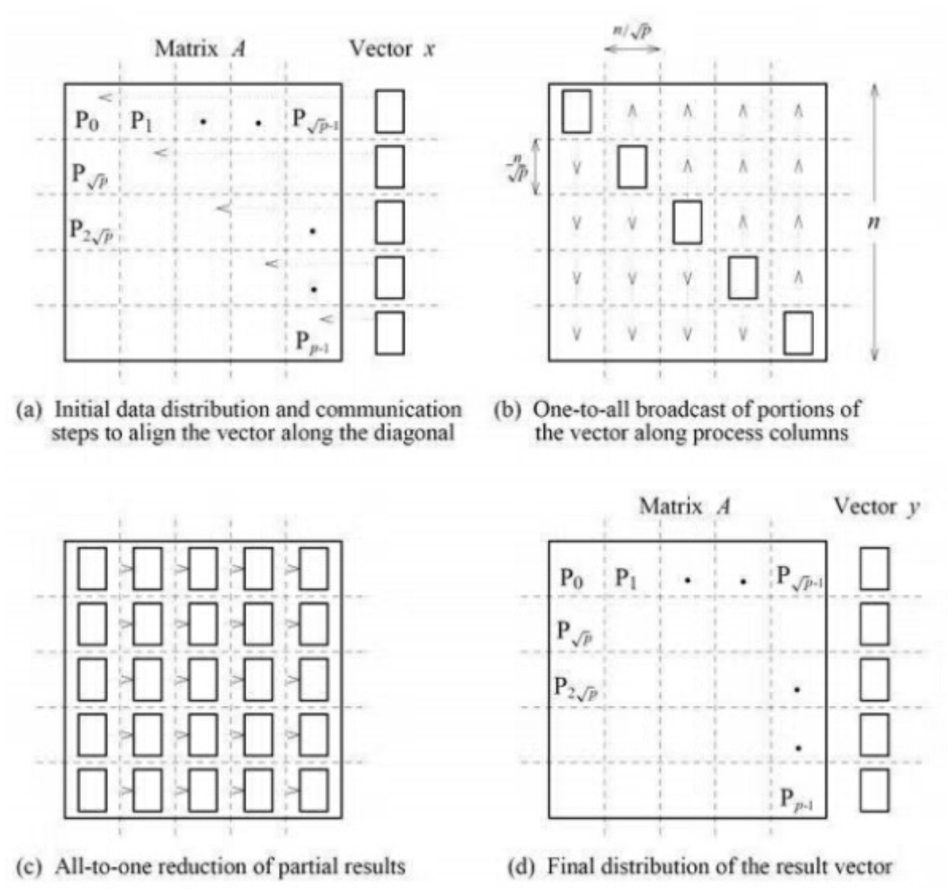
(d) Final distribution of the matrix and the result vector y

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

there is no silver bullet, multiple implementations required to find the best solution.

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];
    }
  }
}
```

three nested loops n^3

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

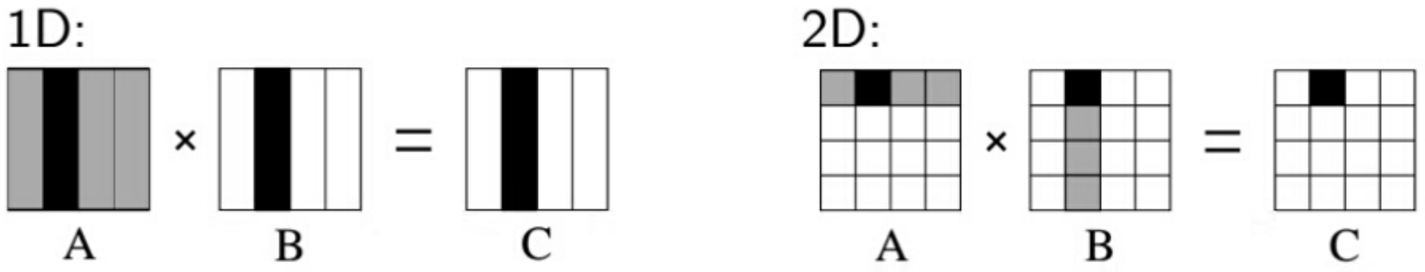
Best sequential implementation

The idea is to cache intermediate result (cache) to avoid redundant multiplication.

In each of the partition choice some communication is required unless the data is replicated across all nodes.

Distributed-Memory Matrix-Matrix Multiplication

- *Partition Choices:*



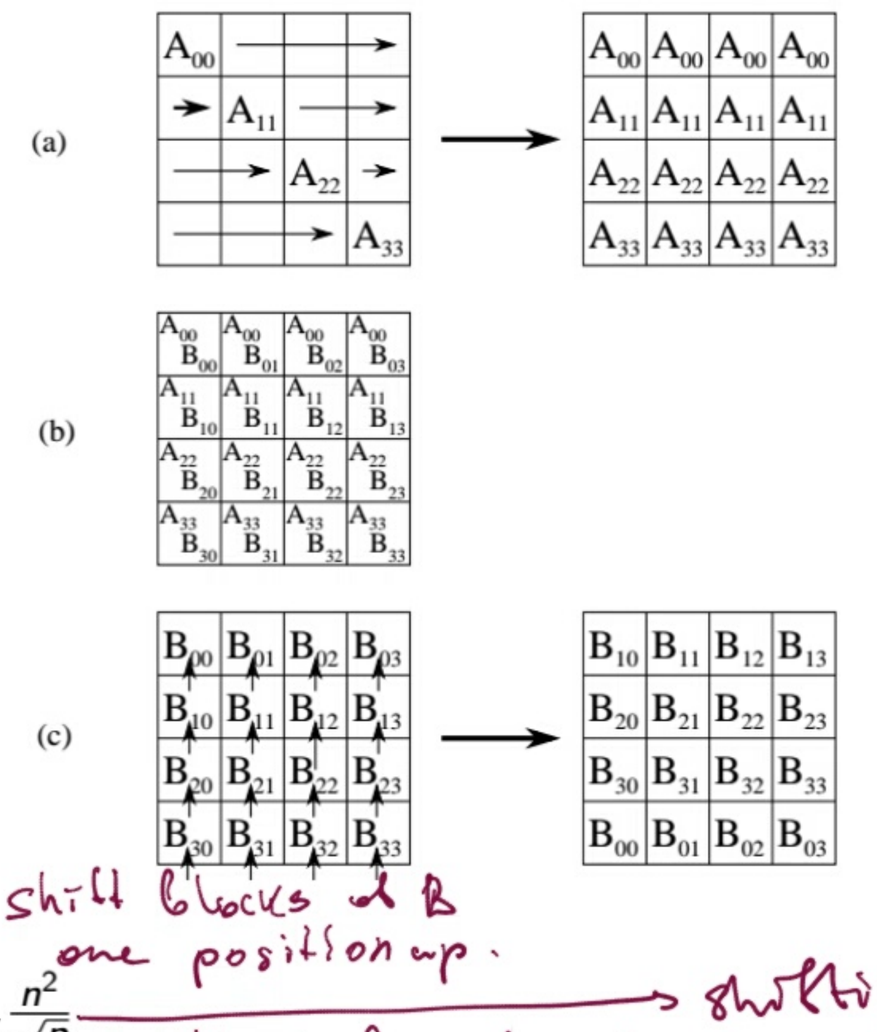
- *Communication Choices:* synchronous, and hardware optimized.
  - A single broadcast — The needed data is broadcast to all destinations once for all. (but what if everyone is source?)
    - A single step, but needs a lot of buffer storage.
  - Multiple shifts — The needed data is shifted towards its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

Again many solutions: try, and error approach.

One possible implementation of MPI algorithm.

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: $T_p \approx \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}$ (initial broadcast; shifting)

Every Processes has submatrix of A and B

Tradeoff: how much data each process will buffer?

(Entire matrix is not acceptable)

Can we get rid of broadcast?

Improved reading? yes → use Cannon's Matrix-Mult.
no → Don't bother.

Cannon's Matrix-Mult Algorithm (LOOK IT UP)

Memory efficient variant of the simple algorithm.

Key Idea: Replace the standard loop

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

→ triple loop nest

with a skewed loop (in summation order is not important (i can be anything))

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

skewing.

Communication: Multiple shifts for both arrays A and B.

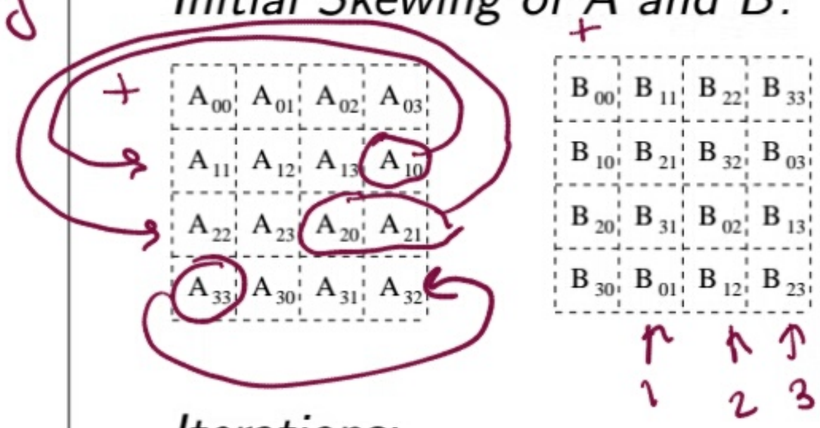
Complexity: $T_p \approx \frac{n^3}{p} + 2t_s\sqrt{p} + 2t_w\frac{n^2}{\sqrt{p}}$

if skewing is possible → small performance increase

Cannon's Matrix-Mult Algorithm (cont.)

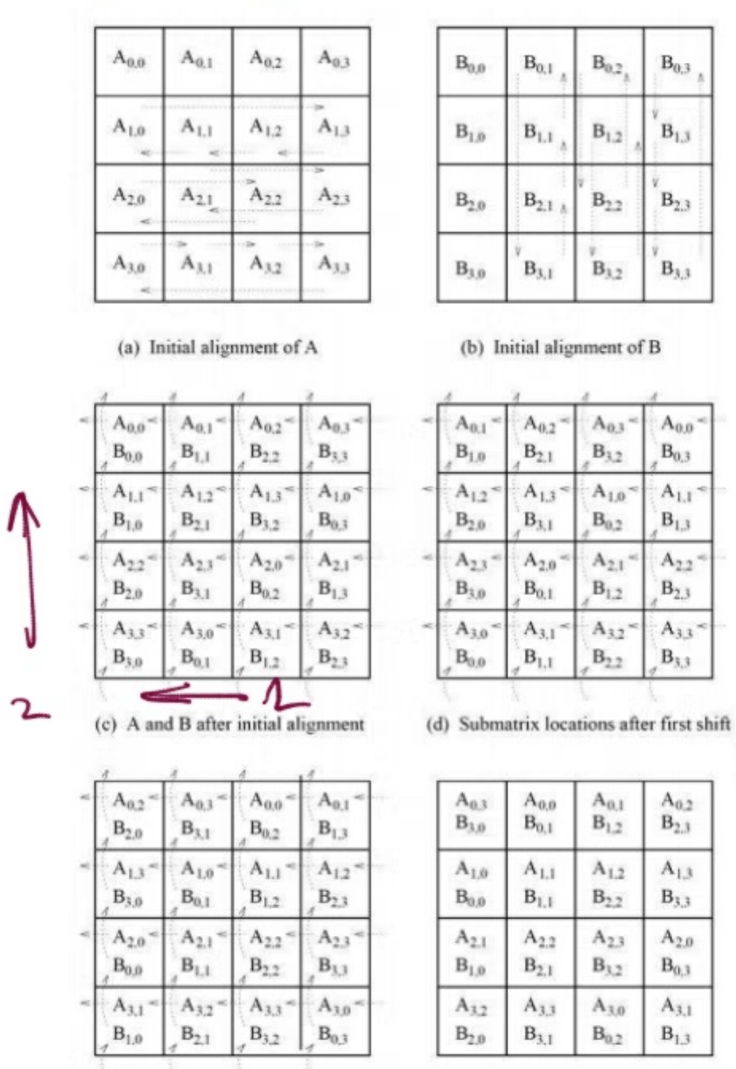
Skewing

Initial Skewing of A and B:



Iterations:

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



Throw away old blocks

Efficient when skewing can be done efficiently → otherwise don't bother.

After skewing all processes can start work.
Shifts is inexpensive

Gaussian Elimination (engine that drives many computations)

Many problems can be reduced to it.

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

Idea:

T-triangular

- First reduce $\mathbf{Ax} = \mathbf{b}$ to an upper triangular system $\mathbf{Tx} = \mathbf{c}$.
- Then use back substitution to solve for $\mathbf{x}$. ——++ (more expensive)

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.

} preserve solution steps.

Complexity directly proportional to the size of the matrix $= n^3$

Gaussian Elimination Example

- The initial linear system: High school 4 unknowns → 4 equations (at least)

```
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:

Already in transformed form

```
 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:

Switch back

```
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
```

Solved →

Old system and new systems are completely equivalent

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

If matrix can be transformed in this form it can be solved.

Yes: any system can be solved in this way

No: the only two operations we have sometimes are not enough. (Shuffling is required sometimes)

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;
    }
  }
}
```

not so bad for partitioning

divide transforming elements by the diagonal (this is one).

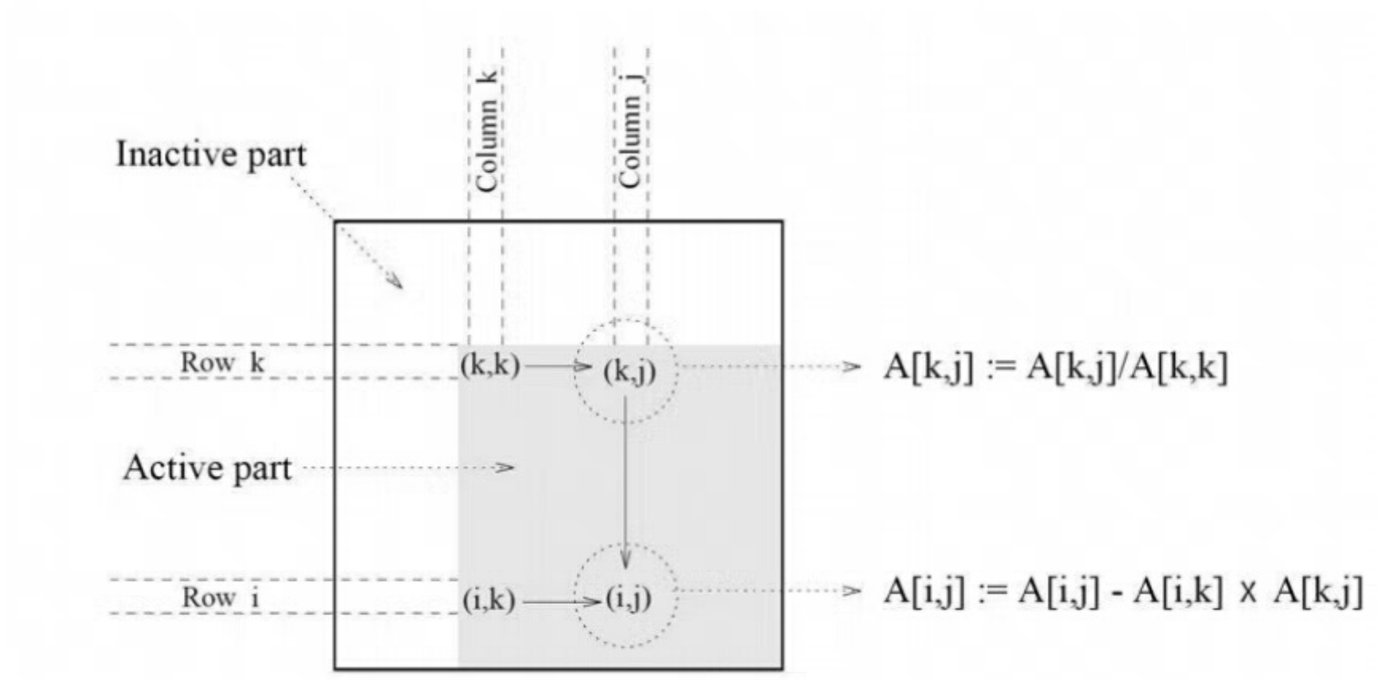
Solution preserving operation

Cancel elements to reset them to 0 (below diagonal)

worst case partitioning doesn't matter

Complexity: $T = O(n^3)$ → Obvious (Not so bad $N^2 + N^3$) $= O(n^3)$

Naive Gaussian Elimination (cont.)



Any following transformation won't affect previous results.

Naive is not stable i.e. integer can be overflown or divided by a very small value as ϵ

Notice: row order of the initial system is irrelevant - hence partial pivoting - move all rows with highest coef to higher order (row).

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:

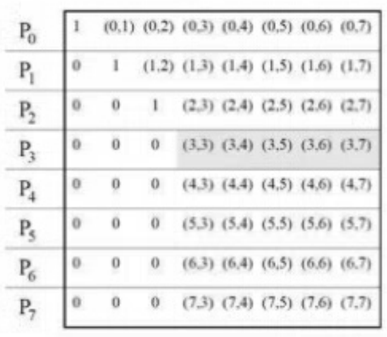
```
Initial:                          => Pivoting:
  2.0  1.0  3.0  4.0  29.0            5.0  2.0  0.0  1.0  13.0
  1.0  0.0  0.0  1.0   5.0            1.0  0.0  0.0  1.0   5.0
  3.0  1.0  1.0  0.0   8.0            3.0  1.0  1.0  0.0   8.0
  5.0  2.0  0.0  1.0  13.0            2.0  1.0  3.0  4.0  29.0

=> 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

How to partition array?

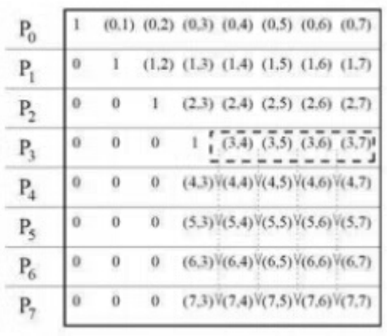
Each row is mapped to a process.



(a) Computation:

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

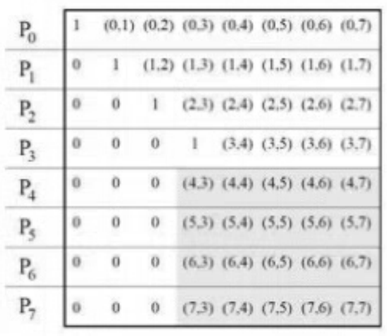
(ii) A[k,k] := 1



(b) Communication:

One−to−all broadcast of row A[k,*]

from pivot to other rows.



(c) Computation:

(i) A[i,j] := A[i,j] − A[i,k]x A[k,j] for k < i < n and k < j < n

(ii) A[i,k] := 0 for k < i < n

Backward substitution requires message passing to above process

How to partition this? any operation in a row depends on operation in a column.

In conclusion: with this partition only one communication is needed.

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:

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

Much better

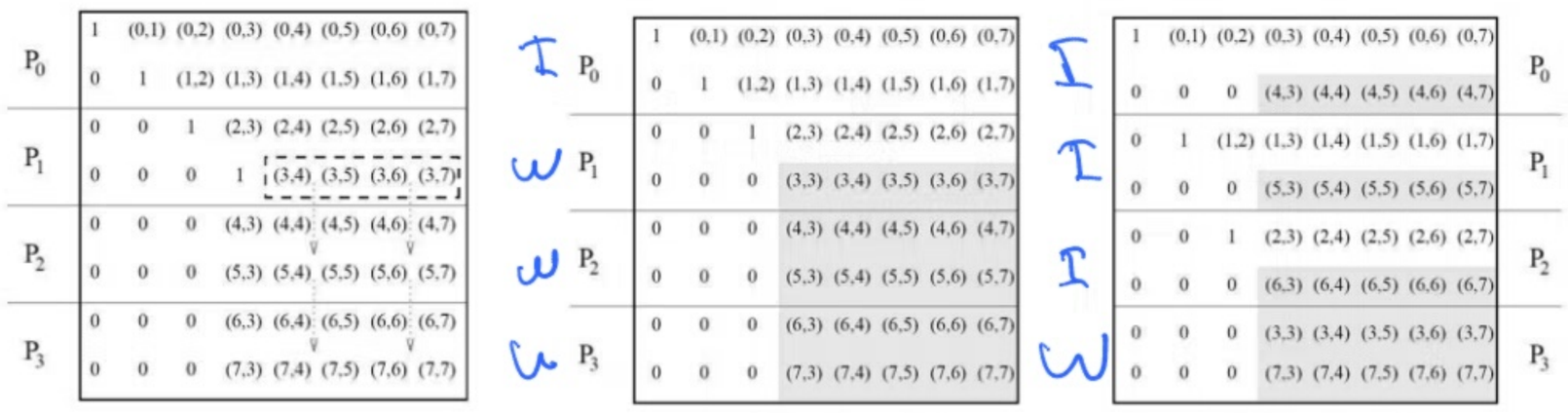
Highest is n^2

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.

I - idling W - working



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

load balancing is required since computation tends to shift from one memory region to another.

Solution: Cyclic Partition dynamically reassign partitions to idling processes. (Much more Complicated!)

Full pivot: row + column shuffling.

Parallel GE with Partial-Pivoting

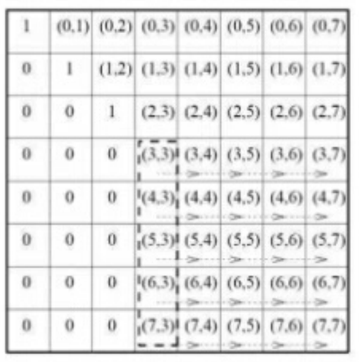
- Distribute the matrix as complete rows; each process stores approximately n/p rows of the matrix.
- Processes collectively decide on which two rows need to be swapped to do the partial pivoting.
- Two processes do a `send/recv` to each other to do the swap. (Alternatively, we can keep track of which 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

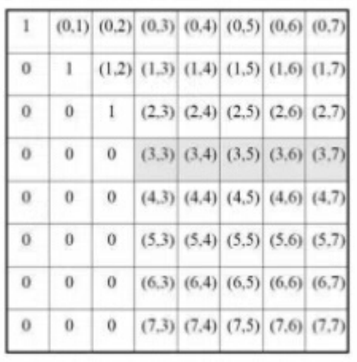
Each processor gets a 2D block of the matrix.

Steps:

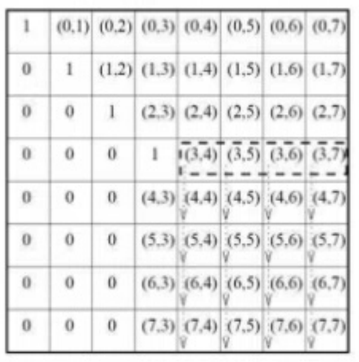
- Broadcast the "active" column along the rows.
- Prepare the pivot row concurrently.
- Broadcast the pivot row along the columns.
- Perform the elimination step concurrently.



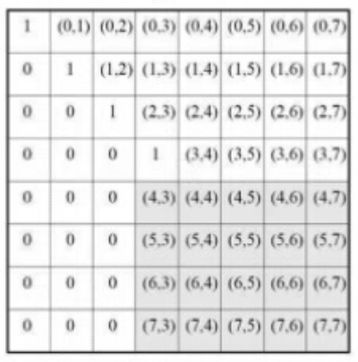
(a) Rowwise broadcast of A[i,k] for (k - 1) < i < n



(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] x A[k,j] for k < i < n and k < j < n

1. 2D partitioning matching with problem domain better.

2. Hardware natively is 2D partitioned. (2D is easier to link than 1D)

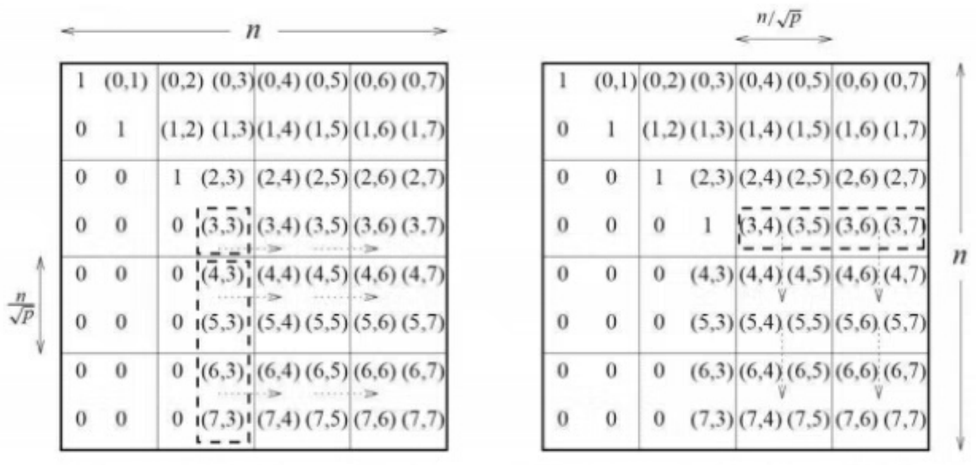
Each node is subblock of the original grid.

- smaller message size.
-

Again Experiments are always required to find better Performance

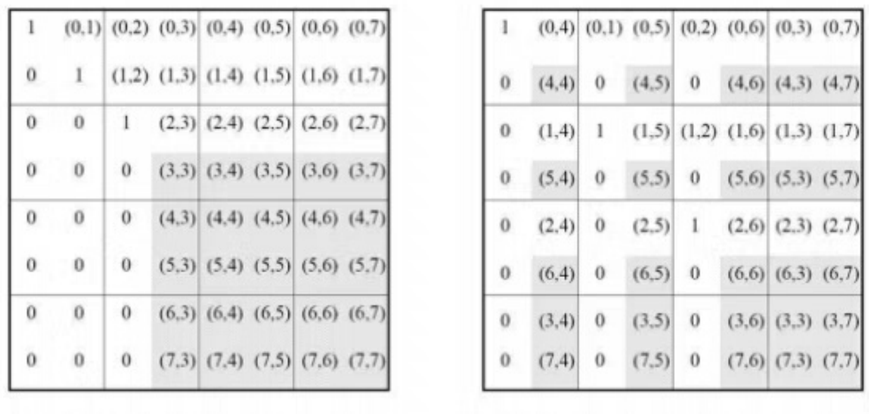
Sparse: value depends only on neighbours value.

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:

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

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.

* in the dense case there is n×n unknowns, but not in this case

ϕ is a physical plane $\phi_{i,j}$ → coordinate

ϕ_{ij} depends on the value of it's neighbours.

n×n plane → n^2 points (solving equation has n unknowns $O(n^3)$

$n^2 = O(n^6)$ ← very expensive for gaussian elimination approach

Can be solved with gaussian dimination, however a lot of empty coorse will be done.

Solving the Laplace Equation

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

Doesn't give the exact final solution, however gives close enough answer.

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.

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

(new)

- 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.

The only problem we are trying to achieve Consider temperature fluctuation for small enough delta to be solved.

Direct implementation of the given form.
Convergence – number of iterations to reach final point.

Jacobi Relaxation Algorithm (cont.) (Naive approach)

```
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;
}
```

Trivial implementation

fmax → not necessary.

Jacobi

Gauss-Seidel Algorithm (Do not use old values).

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:

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

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

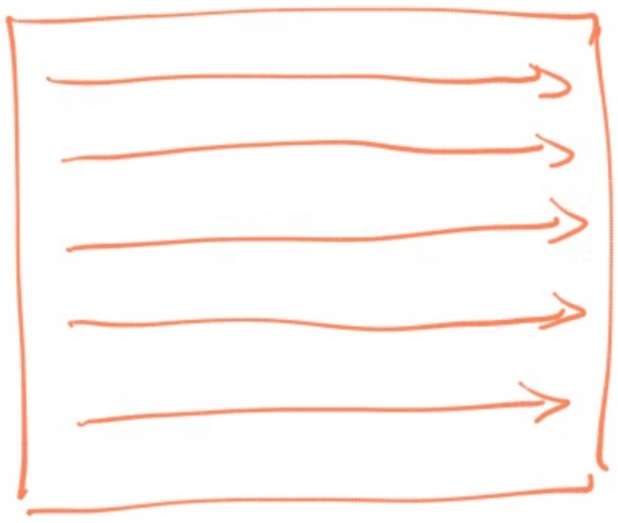
natural

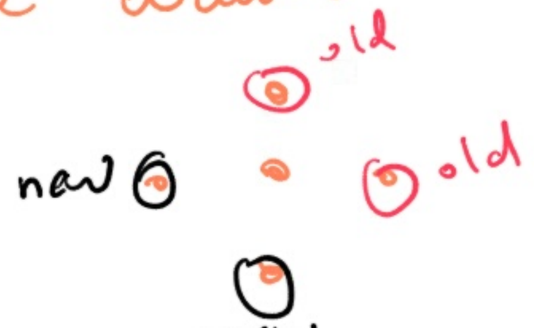
- Gauss-Seidel algorithm does not need buffers and has a better convergence speed (since newer values are used in all updates).

The whole thing gives you approximation of the result.
No need to buffer.
New value

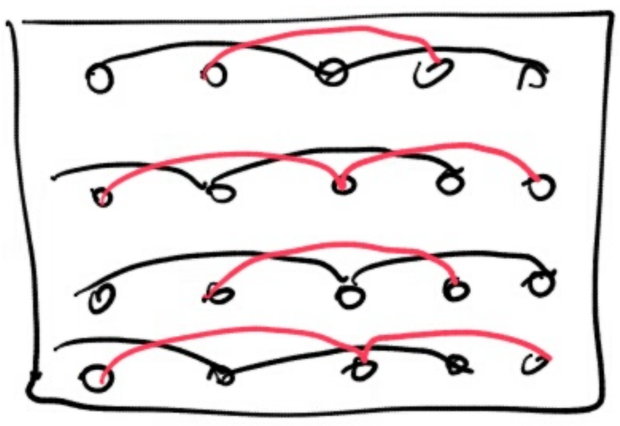
Two particular al-

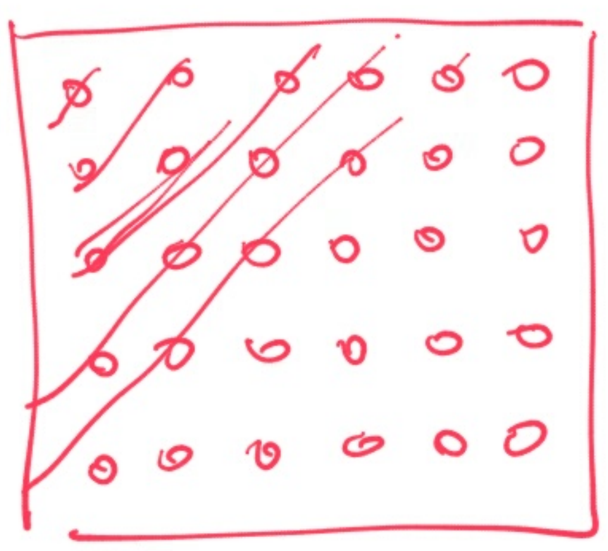
No constrains : start on any node
iterate in any order you want.

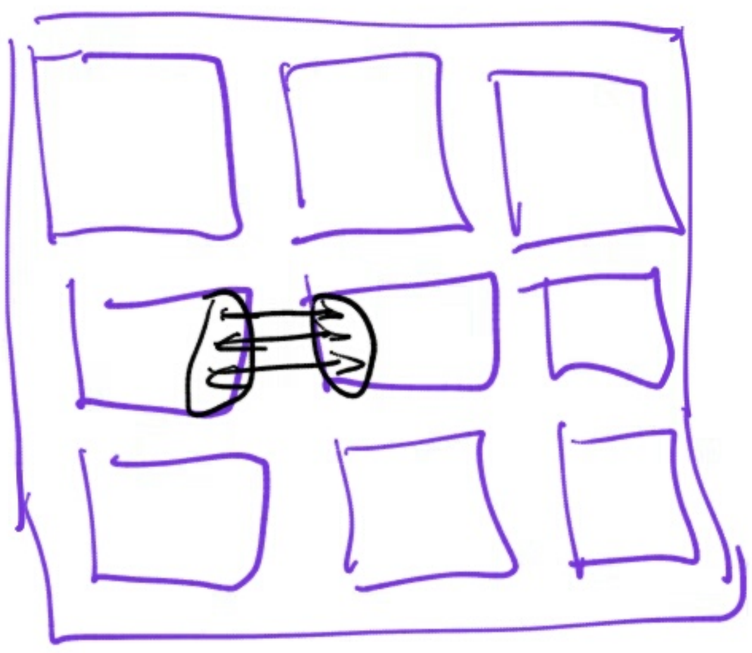




Old then





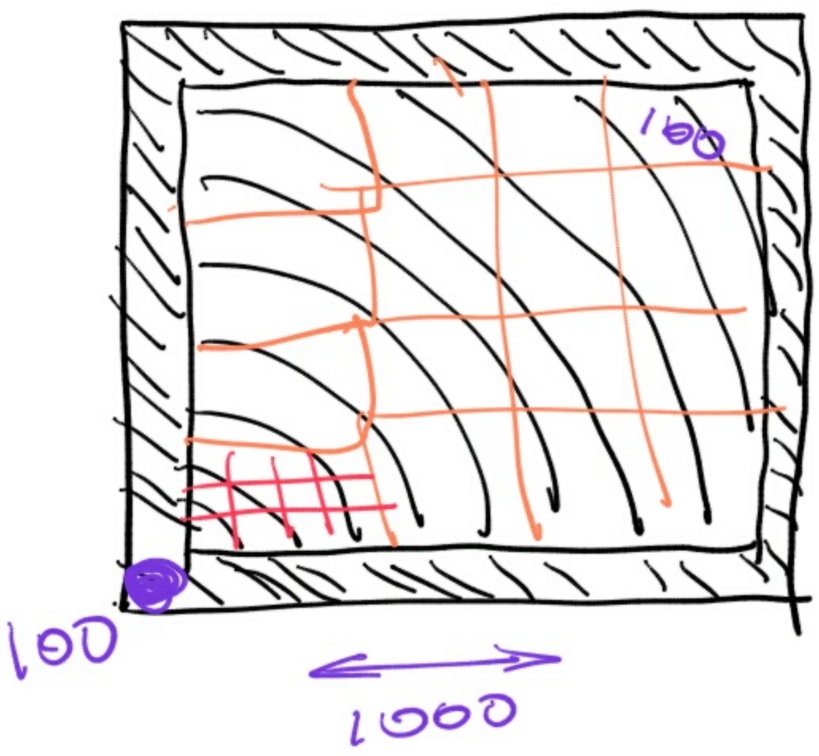


Chaotic 1.2.3 → communicate every iteration

Postpone → Collect update and communicate every 5 iterations (or something else)

↙ Slower convergion, no costly communication.

Better algorithm : Multi Grid.



Border values do not change

for laplace it will take 1000 (n) iterations to propagate to top right corner

Construct smaller domain 100 x 100 or 10x10

Each 100x100 is broken into 10 x 10 recursively.

I can be broken in to smaller pieces if needed.

Use smaller dynamic domains to propagate values faster.

Read about it?

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

```
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
```

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


Multi-Grid Methods much better.

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.