Parallel Linear Algebra

Our goals: Fast and efficient parallel algorithms for

- the matrix-vector product,
- the matrix-matrix product,
- solving systems of linear equations,
- applying finite difference systems,
- and computing the fast Fourier Transform.

The matrix-vector product is the basis of most of our algorithms.

Decomposing a matrix

How to distribute an $m \times n$ matrix A to p processes?

- Rowwise decomposition:
  each process is responsible for m/p contiguous rows.
- Columnwise decomposition:
  each process is responsible for n/p contiguous columns.
- Checkerboard decomposition:
  Assume that k divides m and that l divides n.
  - Assume moreover that $k \cdot l = p$.
  - Imagine that the processes form a $k \times l$ mesh.
  - Process (i, j) obtains the submatrix of A consisting of the ith row interval of length m/k and the jth column interval of length n/l.

The Matrix-Vector Product

Our goal: Compute $y = A \cdot x$ for a $m \times n$ matrix A and a vector x with n components.

- Assumptions:
  - We do assume that matrix A has been distributed to the various processes.
  - Process 1 knows the vector x and has to determine the vector y.
- The conventional sequential algorithm determines y by setting

$$y_i = \sum_{j=1}^{n} A[i,j] \cdot x_j.$$

  - To compute y_i we perform n multiplications and $n-1$ additions.
  - Overall $m \cdot n$ multiplications and $m \cdot (n-1)$ additions suffice.

The Rowwise Decomposition

- Replicate x: broadcast x to all processes in time $O(n \cdot \log_2 p)$.
- Each process determines its $\frac{m}{p}$ vector-vector products in time $O(\frac{m \cdot n}{p})$.
- Process 1 performs a Gather operation in time $O(m)$: $p - 1$ messages of length m/p are involved.
- Performance analysis:
  - Communication time is proportional to $n \cdot \log_2 p + m$ and overall time $\Theta(m \cdot n/p + n \cdot \log_2 p + m)$ is sufficient.
  - Efficiency is $\Theta(m \cdot n/(m \cdot n + p \cdot (n \cdot \log_2 p + m)))$.
  - Constant efficiency follows, if $m \cdot n = \Omega(p \cdot (n \cdot \log_2 p + m)) = \Omega(p \cdot \log_2 p \cdot n + m \cdot p)$
  - Hence we get constant efficiency for

$$m = \Omega(p \cdot \log_2 p) \text{ and } n = \Omega(p).$$

The Columnwise Decomposition

- Apply MPI_Scatter to distribute the blocks of x to "their" processes. Since this involves $p-1$ messages of length n/p, time $O(n)$ is sufficient.
- Each process i computes the matrix-vector product $y^i = A^i \cdot x^i$ for its block A^i of columns.
  Time $O(m \cdot n/p)$ is sufficient.
- Process 1 applies a Reduce operation to sum up $y^1, y^2, \ldots, y^p$ in time $O(m \cdot \log_2 p)$.
- Performance analysis:
  - Run time is bounded by $O(m \cdot n/p + n + m \cdot \log_2 p)$.
  - Here we have constant efficiency, if computing time dominates communication time. Require

$$m = \Omega(p) \text{ and } n = \Omega(p \cdot \log_2 p).$$

Checkerboard Decomposition

- Process 1 applies a Scatter operation addressed to the l processes of row 1 of the process mesh. Time $O(l \cdot \frac{n}{l}) = O(n)$.
- Then each process of row 1 broadcasts its block of x to the k processes in its column: time $O(\frac{n}{l} \cdot \log_2 k)$ suffices. All processes compute their matrix-vector products in time $O(m \cdot n/p)$.
- The processes in column 1 of the process mesh apply a Reduce operation for their row to sum up the l vectors of length $\frac{m}{k}$: time $O(m/k \cdot \log_2 l)$ is sufficient.
- Process 1 gathers the $k-1$ vectors of length $\frac{m}{k}$ in time $O(m)$.
- Performance analysis:
  - The total computation time is bounded by $O(m \cdot n/p + n + \frac{n}{l} \cdot \log_2 k + \frac{m}{k} \cdot \log_2 l + m)$.
  - The total communication time is bounded by $O(n+m)$, provided $\log_2 k \leq l$ and $\log_2 l \leq k$.
  - We obtain constant efficiency, if $m = \Omega(p)$ and $n = \Omega(p)$.

Summary

The checkerboard decomposition has the best performance, if $m \approx n$. Why?

- All three decompositions have the same computation time.
- Assuming $m = n$,
  - the communication time of the rowwise decomposition is dominated by boadcasting the vector x: time $O(n \log_2 p)$,
  - whereas the final Reduce dominates for the columnwise decomposition: time $O(m \log_2 p)$.
  - The checkerboard decomposition cuts down on the message length!

Matrix-Matrix Product

Our goal is to compute the $n \times n$ product matrix $C = A \cdot B$ for $n \times n$ matrices A and B.

- To compute $C[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j]$ sequentially, n multiplications and $n-1$ additions are required. Since C has n^2 entries, we obtain running time $\Theta(n^3)$.
- We discuss four approaches:
  - the first algorithm uses the rowwise decomposition.
  - The algorithm of Fox and its improvement, the algorithm of Cannon, use the checkerboard decomposition.
  - The DNS algorithm assumes a variant of the checkerboard decomposition.

The Rowwise Decomposition

- Process i receives the submatrices A^i of A and B^i of B, corresponding to the ith row interval of length $\frac{n}{p}$.
- Further subdivide A^i, B^i into the $\frac{n}{p}$ square submatrices $A^{i,j}, B^{i,j}$.
- Define $C^{i,j}$ analogously and observe that $C^{i,j} = \sum_{k=1}^{p} A^{i,k} \cdot B^{k,j}$ holds. The computation:
  - In phase 1 process i computes all products $A^{i,i} \cdot B^{i,j}$ for $j = 1, \ldots, p$ in time $O(p \cdot \frac{n}{p} \cdot \frac{n}{p} \cdot \frac{n}{p}) = O(\frac{n^3}{p^2})$, then sends B^i to process $i+1$ and receives B^{i-1} from process $i-1$ in time $O(n^2/p)$.
  - In phase 2 process i computes all products $A^{i,i-1} \cdot B^{i-1,j}$, sends B^{i-1} to process $i+1$ and receives B^{i-2} from $i-1 \ldots$.
- Performance analysis:
  - All in all p phases. Hence computing time is bounded by $O(n^3/p)$ and communication time is bounded by $O(n^2)$.
  - The compute/communicate ratio $\frac{n^3}{p}/n^2 = \frac{n}{p}$ is small!

The Algorithm of Fox

- We again determine the product matrix according to $C^{i,j} = \sum_{k=1}^{p} A^{i,k} \cdot B^{k,j}$, but now
  - processes are arranged in a $\sqrt{p} \times \sqrt{p}$ mesh of processes.
  - Process i knows the $n/\sqrt{p} \times n/\sqrt{p}$ submatrices $A^{i,j}$ and $B^{i,j}$.
- We have $\sqrt{p}$ phases.
  In phase k we want process (i, j) to compute $A^{i,i+k-1} \cdot B^{i+k-1,j}$:
  - process $(i, i + k - 1)$ broadcasts $A^{i,i+k-1}$ to all processes in row i,
  - process (i, j) computes $A^{i,i+k-1} \cdot B^{i+k-1,j}$,
  - receives $B^{i+k,j}$ from $(i + 1, j)$ and sends $B^{i+k-1,j}$ to $(i - 1, j)$.
- Performance Analysis:
  - Per phase: computing time $O((\frac{n}{\sqrt{p}})^3)$ and communication time $O(\frac{n^2}{p} \cdot \log p)$.
  - We have $\sqrt{p}$ phases: computation time $O(\frac{n^3}{p})$, communication time $O(\frac{n^2}{\sqrt{p}} \cdot \log p)$. The compute/communicate ratio $\frac{n}{\sqrt{p} \log_2 p}$ increases.

The Algorithm of Cannon

- The setup is as for the algorithm of Fox.
  In particular, process (i, j) has to determine $C^{i,j} = \sum_{k=1}^{p} A^{i,k} \cdot B^{k,j}$.
- At the very beginning, redistribute matrices, such that process (i, j) holds $A^{i,i+j}$ and $B^{i+j,j}$.
- We again have $\sqrt{p}$ phases. In phase k we want process (i, j) to compute $A^{i,i+j+k-1} \cdot B^{i+j+k-1,j}$:
  - process (i, j) computes $A^{i,i+j+k-1} \cdot B^{i+j+k-1,j}$,
  - sends $A^{i,i+j+k-1}$ to $(i, j-1)$ and $B^{i+j+k-1,j}$ to $(i-1, j)$ and
  - receives $A^{i,i+j+k}$ from $(i, j+1)$ and $B^{i+j+k,j}$ from $(i+1, j)$.
- Performance Analysis:
  - Per phase: computation time $O((\frac{n}{\sqrt{p}})^3)$, communication time $O((\frac{n}{\sqrt{p}})^2)$.
  - Overall, computation time $O(\frac{n^3}{p})$, communication time $O(\frac{n^2}{\sqrt{p}})$ and the compute/communicate ratio $\frac{n}{\sqrt{p}}$ increases again.

How did we save Communication?

- Rowwise decomposition: in each of the p phases row blocks are exchanged.
  All in all $O(p \cdot n^2/p)$ communication.
- The algorithm of Fox: a broadcast in each of the $\sqrt{p}$ with communication time $O(n^2/p \cdot \log p)$.
  All in all communication time $O(n^2/\sqrt{p} \cdot \log p)$: merging point-to-point messages into broadcasts is profitable!
- The algorithm of Cannon: after initially rearranging submatrices, the broadcasts in the algorithm of Fox are replaced by point to point messages.
  All in all communication time $O(\sqrt{p} \cdot n^2/p)$.

The DNS Algorithm

$p = n^3$ processes are arranged in an $n \times n \times n$ mesh of processes. Process $(i, j, 1)$ stores $A[i, j], B[i, j]$ and has to determine $C[i, j]$.

- We move $A[i, k]$ to process $(i, *, k)$: $(i, k, 1)$ sends $A[i, k]$ to (i, k, k), which broadcasts $A[i, k]$ to all processes $(i, *, k)$.
- Next we move $B[k, j]$ to process $(*, j, k)$: $(k, j, 1)$ sends $B[k, j]$ to (k, j, k), which broadcasts $B[k, j]$ to all processes $(*, j, k)$.
- Process (i, j, k) computes the product $A[i, k] \cdot B[k, j]$.
- Process $(i, j, 1)$ computes $\sum_{k=1}^{n} A[i, k] \cdot B[k, j]$ with MPI_Reduce.
- Performance analysis:
  - The replication step takes time $O(\log_2 n)$, since the broadcast dominates. The multiplication step runs in constant time and the Reduce operation runs in logarithmic time.
  - Time $O(\log_2 n)$ suffices. Its efficiency $\Theta(1/\log_2 n)$ is too small.
  - We scale down.

Scaling down the number of processors

- We work with p processes. Let $q = p^{1/3}$ and imagine that the p processes are arranged in a $q \times q \times q$ mesh.
- Input distribution: process $(i, j, 1)$ receives the $\frac{n}{q} \times \frac{n}{q}$ submatrices $A^{i,j}$ and $B^{i,j}$: the matrices $A^{i,j}$ and $B^{i,j}$ play the role of the entries $A[i, j]$ and $B[i, j]$.
- Mimic the algorithm for n^3 processes.
- Performance analysis:
  - The total computing time is $O(\frac{n^3}{q^3}) = O(\frac{n^3}{p})$, since $\frac{n}{q} \times \frac{n}{q}$ matrices have to be multiplied.
  - During replication and summing, $\frac{n}{q} \times \frac{n}{q}$ matrices are involved and hence the communication time is bounded by $O(\frac{n^2}{q^2} \cdot \log p)$.
  - The compute/communicate ratio is $\frac{n}{q \cdot \log_2 p}$.
- Best performance so far. p should be sufficiently large.

Summary

- The checkerboard decomposition is again better than the rowwise decomposition.
- Cannon's algorithm replaces a broadcast by a point-to-point message and is therefore faster than the algorithm of Fox.
- The DNS algorithm partitions the matrices A and B among q^2 of the q^3 processes.
  - Thus each "input process" gets a relatively large chunk.
  - However there are only two (instead of $\sqrt{p}$) communication steps: namely when replicating and summing.
  - Observe that DNS is better than Cannon only if p is sufficiently large.

Solving Linear Systems

We are given a matrix A and a right-hand side b and would like to solve the linear system $A \cdot x = b$.

- We begin with the easy case of lower triangular matrices A and describe back substitution.
- Then we discuss efficient parallelizations of Gaussian elimination
- and continue with iterative methods: Jacobi relaxation, the Gauss-Seidel algorithm, the conjugate gradient approach and the Newton method.
- Finally we consider parallelization of the finite difference method.

Backsubstitution

We have to solve the system

$$A[i,1] \cdot x_1 + \cdots + A[i,i] \cdot x_i = b_i$$

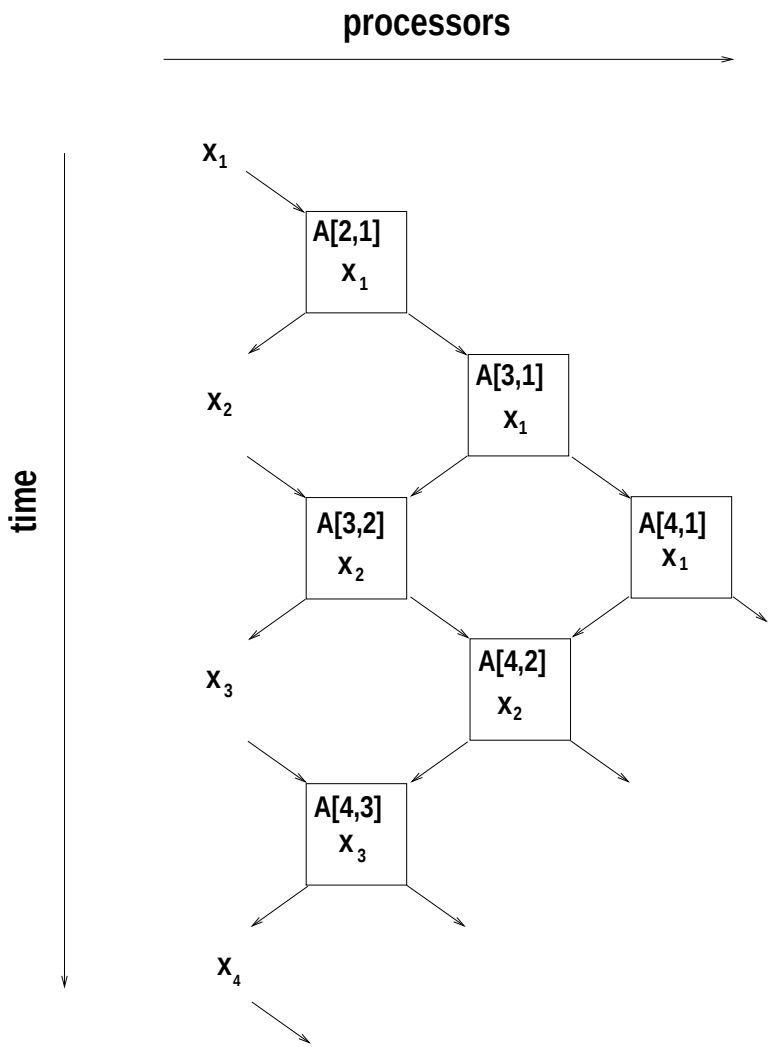
for $i = 1, \ldots, n$.

- A sequential solution:
  - first determine x_1 from the first equation $A[i,1] \cdot x_1 = b_1$.
  - If we already know $x_1, \ldots, x_{i-1}$, then determine x_i from the ith equation.
  - Since an evaluation of the ith equation requires time $O(i)$, the sequential solution runs in time $O(n^2)$.
- We consider two input distributions:
  - The off-diagonal decomposition of matrix A:
    process 1 knows the main diagonal and process i ($i \geq 2$) knows the $i-1$st offdiagonal $A[i,1], A[i+1,2], \ldots, A[n,n-i+1]$.
  - And the rowwise decomposition.

The Off-Diagonal Decomposition I

- We use the linear array as communication pattern.
- Process 1 successively determines $x_1, \ldots, x_n$.
  Once computed, x_i is forwarded thru the linear array.
- How to solve the ith equation $A[i,1] \cdot x_1 + \cdots + A[i,i] \cdot x_i = b_i$?
  - Process i computes $A[i,1] \cdot x_1$ immediately after receiving x_1 from process $i-1$. Then i sends $A[i,1] \cdot x_1$ to process $i-1$ and x_1 to process $i+1$.
  - If process $i-1$ receives x_2 from process $i-2$, it computes the product $A[i,2] \cdot x_2$, sends the sum $A[i,1] \cdot x_1 + A[i,2] \cdot x_2$ to process $i-2$ and forwards x_2 to process i.
  - We communicate according to the principle of "just in time production".

The Off-Diagonal Decomposition II



The Off-Diagonal Decomposition III

Backsubstitution with p processes.

- Assign the off-diagonals $(A[j,1],\ldots,A[n,n-j+1])$ for $j \in \{(i-1)\cdot n/p+1,\ldots,i\cdot n/p\}$ to process i.
- The computing time: we have p phases with compute time $O((n/p)^2)$ per phase.
  All in all compute time is bounded by $O(\frac{n^2}{p})$.
- Communication $O(n/p)$ per phase and hence all in all $O(n)$ communication.

The running time is bounded by $O(\frac{n^2}{p}+n)$.
We achieve constant efficiency, whenever $n=\Omega(p)$.

The Rowwise Decomposition

- This time process i determines x_i.
- Once x_i is determined, process i broadcasts x_i to processes $i+1, \ldots, n$.

For p processes:

- Each process is responsible for $\frac{n}{p}$ variables. Time $O(n)$ per variable is sufficient. $\Rightarrow$ The compute time is bounded by $O(\frac{n^2}{p})$.
- There is one broadcast per unknown and communication time is bounded by $O(n \cdot \log_2 p)$.
- We achieve constant efficiency, whenever $n = \Omega(p \cdot \log_2 p)$.

Gaussian Elimination with Partial Pivoting

Include right hand side b as last column of matrix A.

If we have already eliminated nonzeroes below the diagonals in columns $1, \ldots, i-1$, then

- use largest entry $A[i,j]$ for $j = i, \ldots, n$ as pivot,
- swap rows i and j and set $\text{row}_k = \text{row}_k - \frac{A[k,i]}{A[i,i]} \cdot \text{row}_i$ for $k > i$.

Performance analysis for the sequential algorithm:

- When dealing with row i:
  - Determine the largest entry $A[j,i]$ in column i in time $O(n)$.
  - The elimination step for row i requires $O(n-i+1)$ arithmetic operations.
  - All in all $O(n + (n-i+1)^2) = O(n^2)$ operations suffice.
- The total number of arithmetic operations is bounded by $O(n^3)$.

A parallelization of Gaussian Elimination I

- We work with p processes and the rowwise decomposition: each process receives an "interval" of n/p rows.
- We maintain the sequential structure of pivoting, but parallelize each pivoting step instead.
- Assume that we have reached row i.
  - To utilize the rowwise decomposition we look for the largest entry in row i (and not in column i).
  - We have to eliminate all non-zeroes in row i: the process holding row i has to
    - determine the largest entry $A[i,k]$ in row i,
    - compute the vector m_i of multiples for the elimination step
    - and send m_i to the remaining processes.

A parallelization of Gaussian Elimination II

- Avoid broadcasting (m_i, k). When dealing with row $i-1$:
  - After computing m_{i-1}, the process j holding row i interrupts its elimination work for row $i-1$,
  - immediately recomputes row i and determines (m_i, k) instead,
  - sends (m_i, k) to process $j+1$ and
  - then resumes its elimination work for row $i-1$.
- We cover communication by computation:
  - the expensive broadcast of (m_i, k) is replaced by sending (m_i, k) thru the linear array of processes.
  - Whenever a process receives (m_i, k), it immediately forwards (m_i, k) to its neighbor process.
- Performance analysis:
  - No delay when eliminating row i, if the compute time $\Theta(\frac{n}{p} \cdot n)$ for pivoting dominates the maximal communication delay $p \cdot n$.
- The overall compute time is bounded by $O(n \cdot \frac{n}{p} \cdot n) = O(\frac{n^3}{p})$.

  There is no delay due to communication, provided $n = \Omega(p^2)$.

Iterative Methods

- In an *iterative method* an approximate solution of a linear system $A \cdot x = b$ is successively improved.
- One starts with an initial "guess" $x(0)$ and
- replaces $x(t)$ by a presumably better solution $x(t+1)$.

Assume that the computation of $x(t+1)$ is based on the

matrix-vector product.

- We obtain a fast parallel algorithm and can exploit sparse linear systems.
- We describe:
  - the Jacobi relaxation and its variants,
  - the Newton method to approximately compute the inverse A^{-1}.

Jacobi Relaxation

- Assume that $A \cdot x^* = b$. If A has a nonzero diagonal, then

$$x_i^* = \frac{1}{A[i,i]} \cdot \left(b_i - \sum_{j \neq i} A[i,j] \cdot x_j^* \right).$$

- The Jacobi iteration: if $x_i(t)$ is an approximate solution, set

$$x_i(t+1) = \frac{1}{A[i,i]} \cdot \left(b_i - \sum_{j \neq i} A[i,j] \cdot x_j(t) \right).$$

- Each Jacobi iteration corresponds to a matrix-vector product. Hence one iteration runs in time $O(\frac{n^2}{p})$ and we obtain a fast approximation, whenever few iterations suffice.
- When does the Jacobi iteration converge against the unique solution?

Jacobi Relaxation: Convergence

- Let D be the diagonal matrix with $D[i,i] = A[i,i]$.
  Set $M = D^{-1} \cdot (D - A)$.
  - Another view of the Jacobi iteration:
    if A is invertible and if x^* is the unique solution of $A \cdot x = b$, then

$$x(t+1) - x^* = M \cdot (x(t) - x^*).$$

  - Consequently $x(t) - x^* = M^t \cdot (x(0) - x^*)$ follows for all t.
  - If $\lim_{t\to\infty} M^t = 0$, then $x(t)$ converges against x^*.
- The Jacobi Relaxation converges for row diagonally dominant matrices A: i.e., if

$$|A[i,i]| > \sum_{j \neq i} |A[i,j]|$$

holds for all i.

Two Extensions

- In many practical applications the Jacobi overrelaxation converges faster.
  - for a suitable coefficient γ:

$$x_i(t+1) = (1-\gamma) \cdot x_i(t) + \frac{\gamma}{A[i,i]} \cdot \left(b_i - \sum_{j \neq i} A[i,j] \cdot x_j(t) \right).$$

  - The Jacobi relaxation is a special case: set $\gamma = 1$.
- The Gauss-Seidel algorithm incorporates already recomputed values of x_j (i.e., it replaces $x_j(t)$ by $x_j(t+1)$).
  - An example is

$$x_i(t+1) = \frac{1}{A[i,i]} \cdot \left(b_i - \sum_{j<i} A[i,j] \cdot x_j(t+1) - \sum_{j>i} A[i,j] \cdot x_j(t) \right).$$

  - The Gauss-Seidel method does not look parallelizable!?

Inversion by Newton Iteration

Assume that A is invertible and that X_t is an approximate inverse of A. The Newton iteration is

$$X_{t+1} = 2 \cdot X_t - X_t \cdot A \cdot X_t.$$

- What is the intuition behind this approach?
  - Consider the residual matrix $R_t = I - A \cdot X_t$ which measures the "distance" between A^{-1} and X_t.

$$\begin{aligned} R_{t+1} &= I - A \cdot X_{t+1} \\ &= I - A \cdot (2 \cdot X_t - X_t \cdot A \cdot X_t) \\ &= (I - A \cdot X_t)^2 = R_t^2. \end{aligned}$$

  - R_t converges rapidly towards the 0-matrix, whenever X_0 is a good approximation of A^{-1}.
- Since the Newton iteration is based on matrix-matrix products, each iteration is easily parallelized.

The Finite Difference Method

Finite differences can be used to approximate derivatives of a function f, since

$$\begin{aligned} f'(x) &= \lim_{h\to 0} \frac{f(x+h)-f(x)}{h} = \lim_{h\to 0} \frac{f(x)-f(x-h)}{h} \\ f''(x) &= \lim_{h\to 0} \frac{f'(x+h)-f'(x)}{h} = \lim_{h\to 0} \frac{f(x+h)-2f(x)+f(x-h)}{h^2}. \end{aligned}$$

- The finite difference method is used to solve differential equations:
  - Derivatives are approximated by finite differences
  - and differential equations are modeled by linear systems of equations.
- The usually sparse systems are mostly solved with iterative methods.

An Example: The Poisson Equation

Find a function $u : [0,1]^2 \to \mathbb{R}$ which

- satisfies the Poisson equation $\mathbf{u_{xx} + u_{yy} = H}$
- and which has prescribed values on the boundary of the unit square $[0,1]^2$.

- If u is sufficiently smooth and if h is sufficiently small, then

$$u_{xx}(x,y) \approx \frac{u(x+h,y) - 2u(x,y) + u(x-h,y)}{h^2}.$$

- Approximate $u_{y,y}$ analogously and we get $H(x,y) \approx$

$$\frac{u(x+h,y) + u(x-h,y) + u(x,y+h) + u(x,y-h) - 4u(x,y)}{h^2}.$$

- For N sufficiently large, set

$$h = \frac{1}{N}, \quad u_{i,j} = u(\frac{i}{N}, \frac{j}{N}) \text{ and } H_{i,j} = H(\frac{i}{N}, \frac{j}{N}).$$

The Linear System I

- Choose (x, y) as one of the grid points $\{(\frac{i}{N}, \frac{j}{N}) \mid 0 < i, j < N\}$ and we get the linear system

$$-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} = \frac{H_{i,j}}{N^2}$$

- The system is huge: $(N-1)^2$ equations in $(N-1)^2$ unknowns. (The values of u at the boundary are prescribed.)
- The matrix of the system has $(N-1)^4$ entries, but is sparse, since any equation has at most five nonzero coefficients.
- To utilize sparsity, we apply iterative methods.

The Linear System II

- We process the system beginning with the lower boundary and working upwards:

$$u_{i+1,j} = 4u_{i,j} - (u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) + \frac{H_{i,j}}{N^2}.$$

- We apply the Jacobi Relaxation and get

$$u_{i+1,j}(t+1) = 4u_{i,j}(t) - (u_{i-1,j}(t) + u_{i,j+1}(t) + u_{i,j-1}(t)) + \frac{H_{i,j}}{N^2}.$$

- How to obtain an efficient parallelization of one iteration?
  - We use the checkerboard decomposition of the $N-1 \times N-1$ grid. p processes are arranged in a $\sqrt{p} \times \sqrt{p}$ mesh.
  - The process responsible for determining $u_{i+1,j}(t+1)$ has to know $u_{i-1,j}(t)$, $u_{i,j+1}(t)$, $u_{i,j-1}(t)$ and $u_{i,j}(t)$.
  - Any missing value belongs to the "near-boundary" of a neighbor.

The Linear System: Performance Analysis

- The processes communicate their $O(\frac{N}{\sqrt{p}})$ near-boundary values.
- Afterwards they perform an iteration without further communication: computation time is bounded by $O(\frac{N^2}{p})$, since the update time for any grid point is constant and each process has to update $O(\frac{N^2}{p})$ grid points.
- We have constant efficiency, whenever $N = \Omega(\sqrt{p})$.
- One can show that the matrix of the linear system is symmetric and positive definite: the Jacobi Relaxation converges.

Gauss-Seidel Revisited

- So far we have used the Jacobi Relaxation

$$u_{i+1,j}(t+1) = 4u_{i,j}(t) - (u_{i-1,j}(t) + u_{i,j+1}(t) + u_{i,j-1}(t)) + \frac{H_{i,j}}{N^2}.$$

- Can we use the generally better Gauss-Seidel method instead?
  - Use the new update

$$u_{i,j}(t+1) = \frac{u_{i+1,j}(t) + u_{i-1,j}(t) + u_{i,j+1}(t) + u_{i,j-1}(t)}{4} - \frac{H_{i,j}}{4N^2}.$$

  - Label grid point (i, j) white iff $i + j$ is even and black otherwise: white grid points are updated with black grid points only.
  - Execute one iteration of the Gauss-Seidel algorithm by
    - ★ first updating black grid points conventionally with the Jacobi relaxation.
    - ★ Then apply the Gauss-Seidel algorithm to update white grid points by using the already updated black grid points.