Parallel Numerical Algorithms: Iterative Linear Systems, Differential Equations and Finite Difference Methods

Parallel and Distributed Computing

Department of Computer Science and Engineering (DEI)
Instituto Superior Técnico

December 6, 2012

Outline

- Iterative Methods for Linear Systems
  - Relaxation Methods
- Linear Second-order Partial Differential Equations (PDEs)
  - Finite difference methods
  - Example: steady-state heat distribution
  - Ghost points

Solving Linear Systems: recall

- Direct Methods: solution is sought directly, at once
  - Gaussian Elimination
  - LU Factorization
- Iterative Methods: solution is sought iteratively, by improvement
  - Relaxation Methods
  - Krylov Methods
  - Preconditioning

Iterative Methods for Linear Systems

- Iterative methods for solving linear system $Ax = b$ begin with initial guess for solution and successively improve it until solution is as accurate as desired
- In theory, infinite number of iterations might be required to converge to exact solution
- In practice, iteration terminates when residual $\|r\| = \|b - Ax\|$, or some other measure of error, is as small as desired
- Iterative methods are especially useful when matrix A is sparse because, unlike direct methods, no fill is incurred

Jacobi Method

- Beginning with initial guess $x^{(0)}$, Jacobi method computes next iterate by solving for each component of x in terms of others

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1} a_{ij} x_j^{(k)} \right) / a_{ii}, \ \ i = 1, \cdots, n$$

- If D, L, and U are diagonal, strict lower triangular, and strict upper triangular portions of A, then Jacobi method can be written as

$$x^{(k+1)} = D^{-1} \left(b - (L + U) x^{(k)} \right)$$

Jacobi Method

- Jacobi method requires nonzero diagonal entries, which can usually be accomplished by permuting rows and columns if not already true
- Jacobi method requires duplicate storage for x, since no component can be overwritten until all new values have been computed
- Components of new iteration do not depend on each other, so they can be computed simultaneously
- Jacobi method does not always converge, but it is guaranteed to converge under conditions that are often satisfied (e.g., if matrix is strictly diagonally dominant), though convergence rate may be very slow

Gauss-Seidel Method

- Faster convergence can be achieved by using each new component value as soon as it has been computed rather than waiting until next iteration
- This gives Gauss-Seidel method

$$x_i^{(k+1)} = \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right) / a_{ii}$$

- Using same notation as for Jacobi, Gauss-Seidel method can be written

$$x^{(k+1)} = (D + L)^{-1} \left(b - U x^{(k)} \right)$$

Gauss-Seidel Method

- Gauss-Seidel requires nonzero diagonal entries. Gauss-Seidel does not require duplicate storage for x, since component values can be overwritten as they are computed
- But each component depends on previous ones, so they must be computed successively
- Gauss-Seidel does not always converge, but it is guaranteed to converge under conditions that are somewhat weaker than those for Jacobi method (e.g., if matrix is symmetric and positive definite)
- Gauss-Seidel converges about twice as fast as Jacobi, but may still be very slow

Parallel Implementation

- Iterative methods for linear systems are composed of basic operations such as
  - vector updates
  - inner products
  - matrix-vector multiplication
  - solution of triangular systems
- In parallel implementation, both data and operations are partitioned across multiple tasks
- In addition to communication required for these basic operations, necessary convergence test may require additional communication (e.g., sum or max reduction)

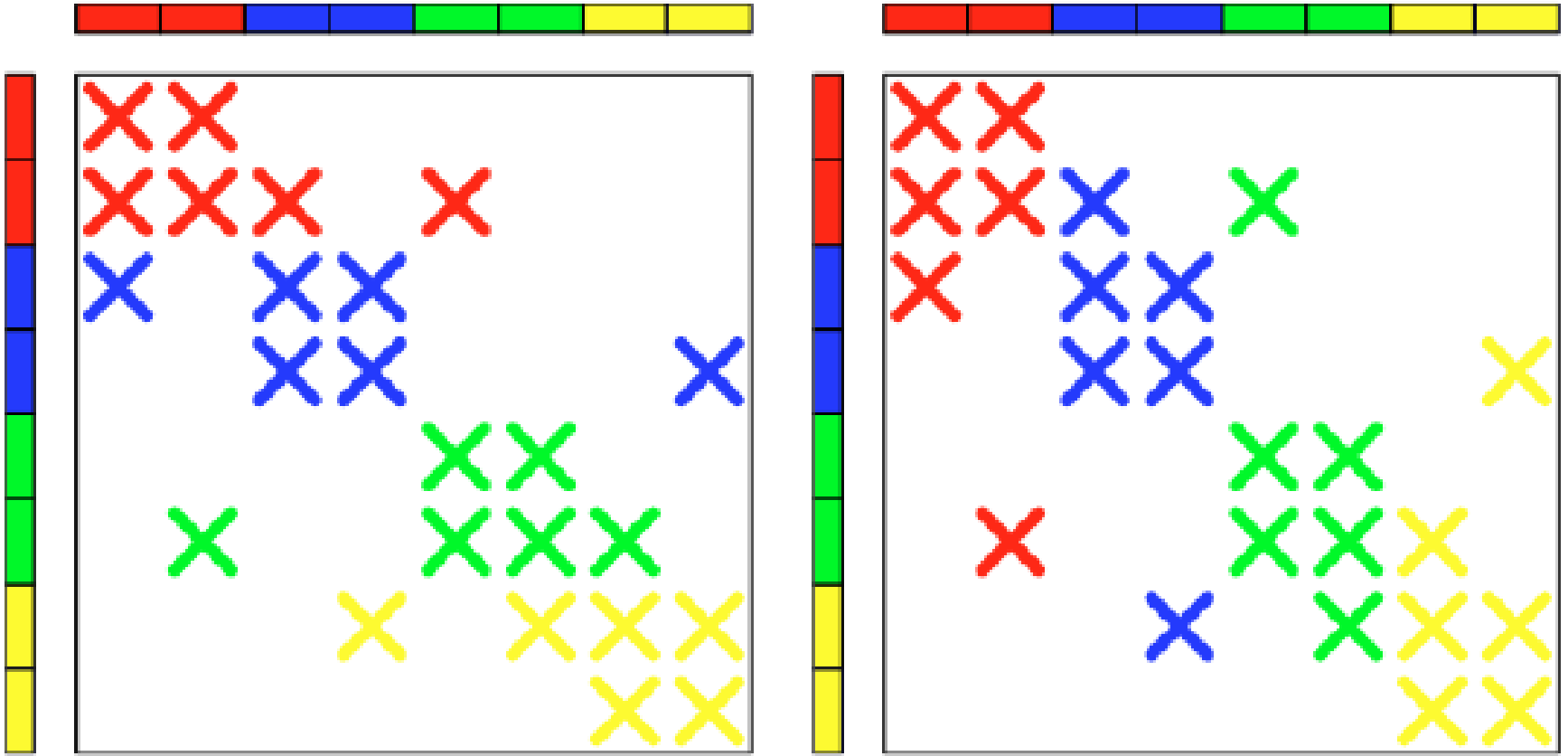
Partitioning of Vectors

- Iterative methods typically require several vectors, including solution x, right-hand side b, residual $r = b - Ax$, and possibly others
- Even when matrix A is sparse, these vectors are usually dense
- These dense vectors are typically uniformly partitioned among p tasks, with given task holding same set of component indices of each vector
- Thus, vector updates require no communication, whereas inner products of vectors require reductions across tasks

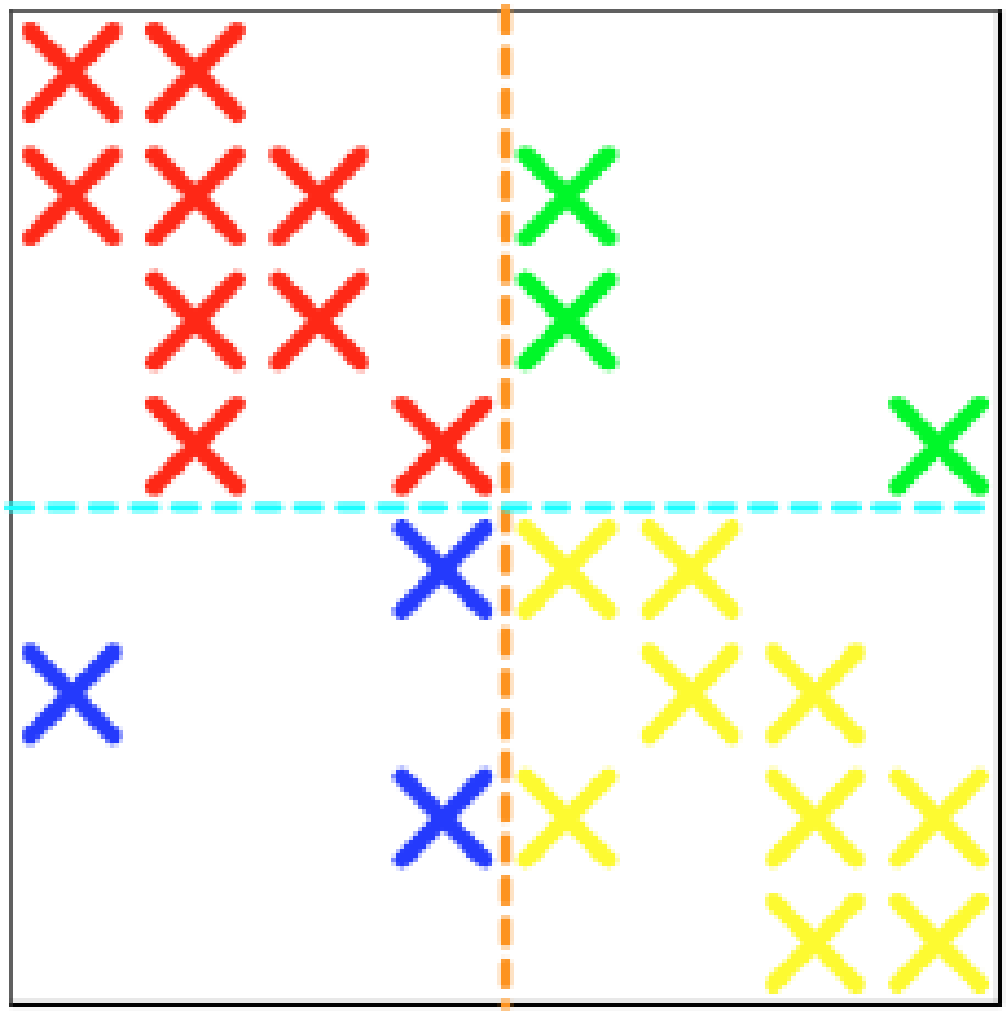
Partitioning of Sparse Matrix

- Sparse matrix A can be partitioned among tasks by rows, by columns, or by submatrices
- Partitioning by submatrices may give uneven distribution of nonzeros among tasks; indeed, some submatrices may contain no nonzeros at all
- Partitioning by rows or by columns tends to yield more uniform distribution because sparse matrices typically have about same number of nonzeros in each row or column

Sparse MatVec with 1-D Partitioning



Sparse MatVec with 2-D Partitioning

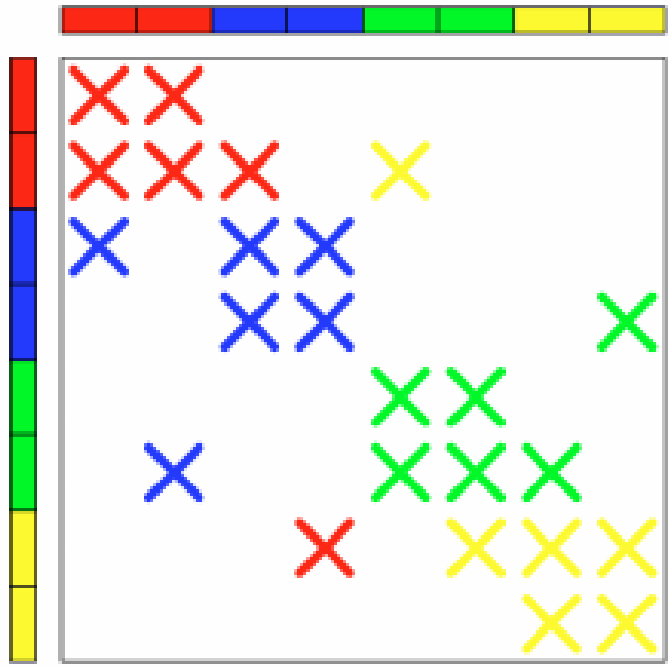


Row Partitioning of Sparse Matrix

- Suppose that each task is assigned n/p rows, yielding p tasks, where for simplicity we assume that p divides n
- In dense matrix-vector multiplication, since each task owns only n/p components of vector operand, communication is required to obtain remaining components
- If matrix is sparse, however, few components may actually be needed, and these should preferably be stored in neighboring tasks
- Assignment of rows to tasks by contiguous blocks or cyclically would not, in general, result in desired proximity of vector components

Two-Dimensional Partitioning

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} 1 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 1 & 9 & 0 & 5 & 0 & 0 & 0 \\ 8 & 0 & 1 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 & 1 & 8 & 0 & 0 \\ 0 & 4 & 0 & 0 & 3 & 1 & 3 & 0 \\ 0 & 0 & 0 & 6 & 0 & 9 & 1 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix}$$



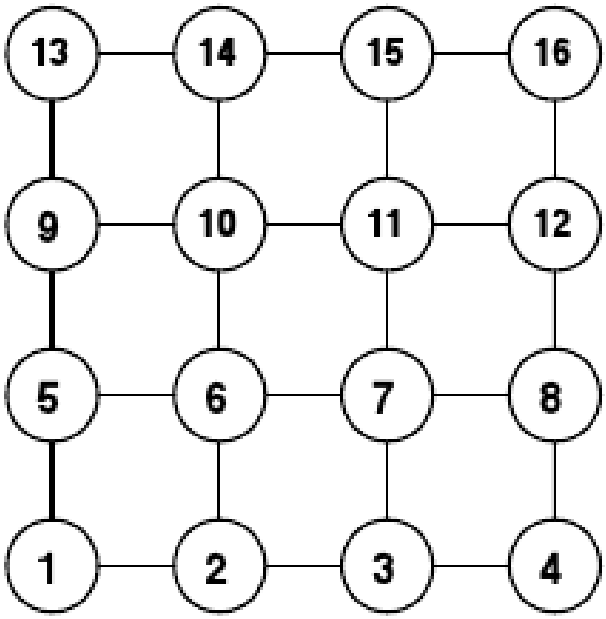
Parallel Jacobi and Gauss-Seidel

- Contiguous groups of variables are assigned to each task, so most communication is internal, and external communication is limited to nearest neighbors in 1-D mesh
- More generally, Jacobi method usually parallelizes well if underlying grid is partitioned in this manner, since all components of x can be updated simultaneously
- Unfortunately, Gauss-Seidel methods require successive updating of solution components in given order (in effect, solving triangular system), rather than permitting simultaneous updating as in Jacobi method

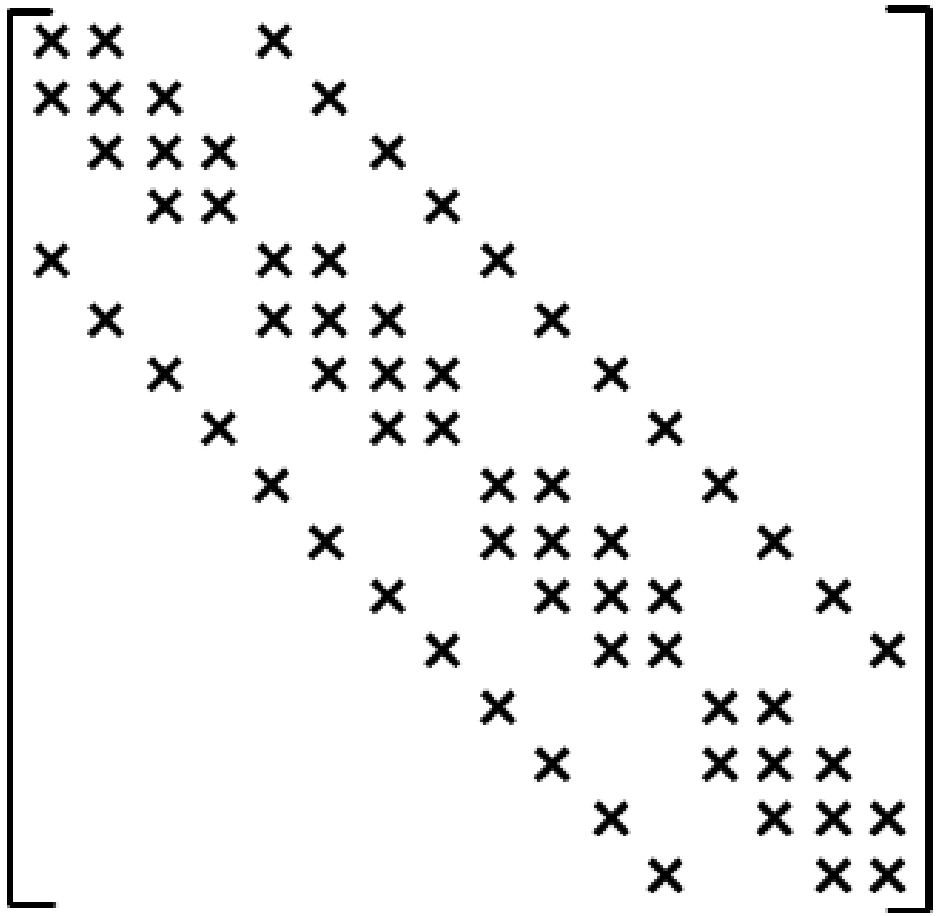
Red-Black Ordering

- Apparent sequential order can be broken, however, if components are reordered according to coloring of underlying graph
- For 5-point discretization on square grid, for example, color alternate nodes in each dimension red and others black, giving color pattern of chess or checker board
- Then all red nodes can be updated simultaneously, as can all black nodes, so algorithm proceeds in alternating phases, first updating all nodes of one color, then those of other color, repeating until convergence

Row-Wise Ordering for 2-D Grid

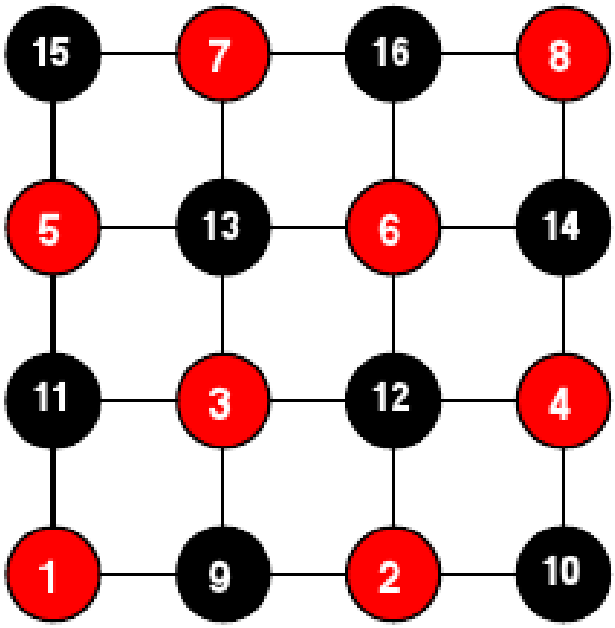


G (A)

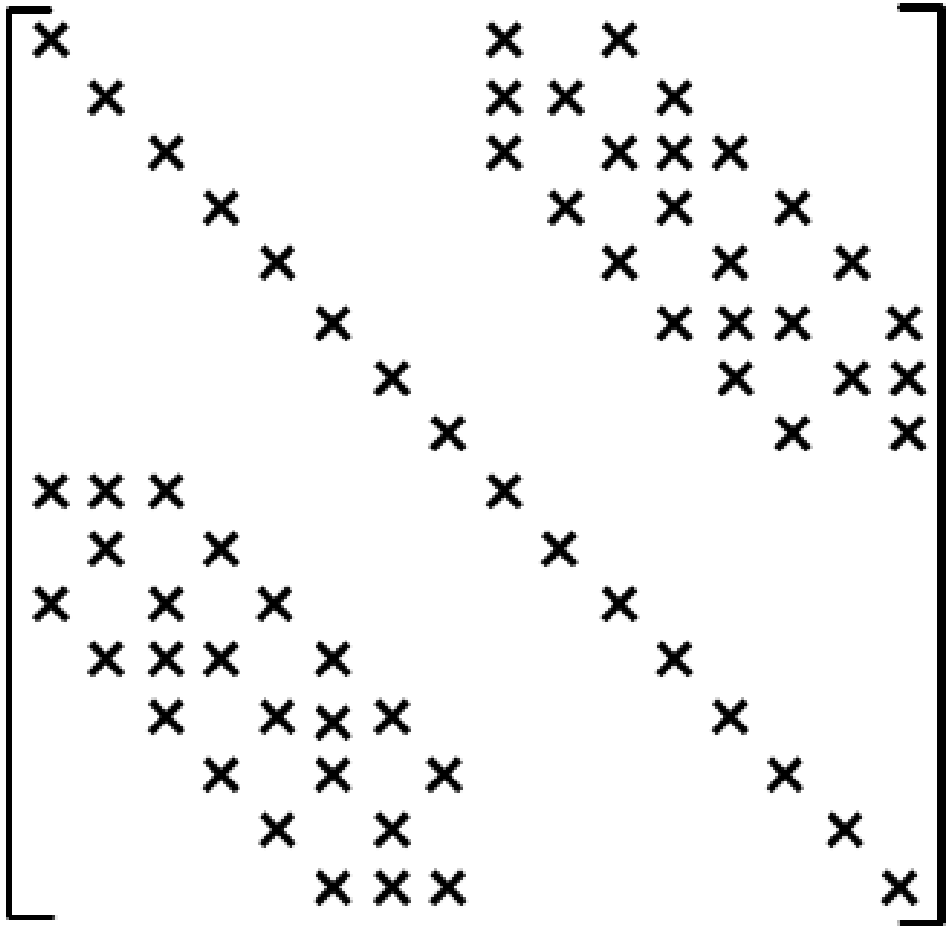


A

Red-Black Ordering for 2-D Grid



$G(A)$



A

Partial Differential Equations

Partial Differential Equations, PDE

Equation containing derivatives of a function of two or more variables (Ordinary if single variable).

$$u = f(x, y) \qquad u_y = \frac{\partial f}{\partial y} \qquad u_{xy} = \frac{\partial^2 f}{\partial x \partial y}$$

Partial Differential Equations

Partial Differential Equations, PDE

Equation containing derivatives of a function of two or more variables (Ordinary if single variable).

$$u = f(x, y) \qquad u_y = \frac{\partial f}{\partial y} \qquad u_{xy} = \frac{\partial^2 f}{\partial x \partial y}$$

Second-Order PDE: contains no partial derivatives of order more than two.

Partial Differential Equations

Partial Differential Equations, PDE

Equation containing derivatives of a function of two or more variables (Ordinary if single variable).

$$u = f(x, y) \qquad u_y = \frac{\partial f}{\partial y} \qquad u_{xy} = \frac{\partial^2 f}{\partial x \partial y}$$

Second-Order PDE: contains no partial derivatives of order more than two.

Linear second-order PDEs are of the form:

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Eu_x + Fu_y + Gu = H$$

A, B, C, D, E, F, G and H are functions of x and y only.

Linear second-order PDEs

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Eu_x + Fu_y + Gu = H$$

1. $\pi u_{xy} + x^2 u_{yy} = \sin(xy)$
2. $u_{xx}^2 + u_{yy} = 0$
3. $uu_{xy} + 4xu_{yy} = x + y$
4. $\sin(xy)u_x + 6xyu_{xy} = 0$

Linear second-order PDEs

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Eu_x + Fu_y + Gu = H$$

1. $\pi u_{xy} + x^2 u_{yy} = \sin(xy)$
2. $u_{xx}^2 + u_{yy} = 0$
3. $uu_{xy} + 4xu_{yy} = x + y$
4. $\sin(xy)u_x + 6xyu_{xy} = 0$

Linear second-order PDEs

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Eu_x + Fu_y + Gu = H$$

1. $\pi u_{xy} + x^2 u_{yy} = \sin(xy)$
2. $u_{xx}^2 + u_{yy} = 0$
3. $uu_{xy} + 4xu_{yy} = x + y$
4. $\sin(xy)u_x + 6xyu_{xy} = 0$

Linear second-order PDEs

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Eu_x + Fu_y + Gu = H$$

1. $\pi u_{xy} + x^2 u_{yy} = \sin(xy)$
2. $u_{xx}^2 + u_{yy} = 0$
3. $uu_{xy} + 4xu_{yy} = x + y$
4. $\sin(xy)u_x + 6xyu_{xy} = 0$

Linear second-order PDEs

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Eu_x + Fu_y + Gu = H$$

1. $\pi u_{xy} + x^2 u_{yy} = \sin(xy)$
2. $u_{xx}^2 + u_{yy} = 0$
3. $uu_{xy} + 4xu_{yy} = x + y$
4. $\sin(xy)u_x + 6xyu_{xy} = 0$

Elliptic PDEs

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Eu_x + Fu_y + Gu = H$$

Elliptic PDEs: $B^2 - AC < 0$

Poisson Equation: $u_{xx} + u_{yy} = f(x, y)$ $\qquad (A = C = 1,\ B = 0)$

(if $f(x, y) = 0$, called Laplace Equation)

- potential problems: electric, magnetic, gravitic
- distribution of heat or charge in conductors
- certain fluid and torsion problems

Parabolic PDEs

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Eu_x + Fu_y + Gu = H$$

Parabolic PDEs: $B^2 - AC = 0$

Heat Equation or Diffusion Equation: $ku_{xx} = u_t$ $\qquad (A = k,\ B = C = 0)$

- heat conduction in solids
- diffusion of liquids and gases

Hyperbolic PDEs

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Eu_x + Fu_y + Gu = H$$

Hyperbolic PDEs: $B^2 - AC > 0$

Wave Equation: $c^2 u_{xx} = u_{tt}$ $\qquad (A = c^2,\ B = 0,\ C = -1)$

- wave propagation model
- vibration of strings and membranes

Finite Difference Methods

In general, not possible to obtain an analytical solution to a PDE.

Finite Difference Methods

Numerical methods that obtain an approximate result of PDEs by dividing the variables (often time and space) into discrete intervals.

Finite Difference Methods

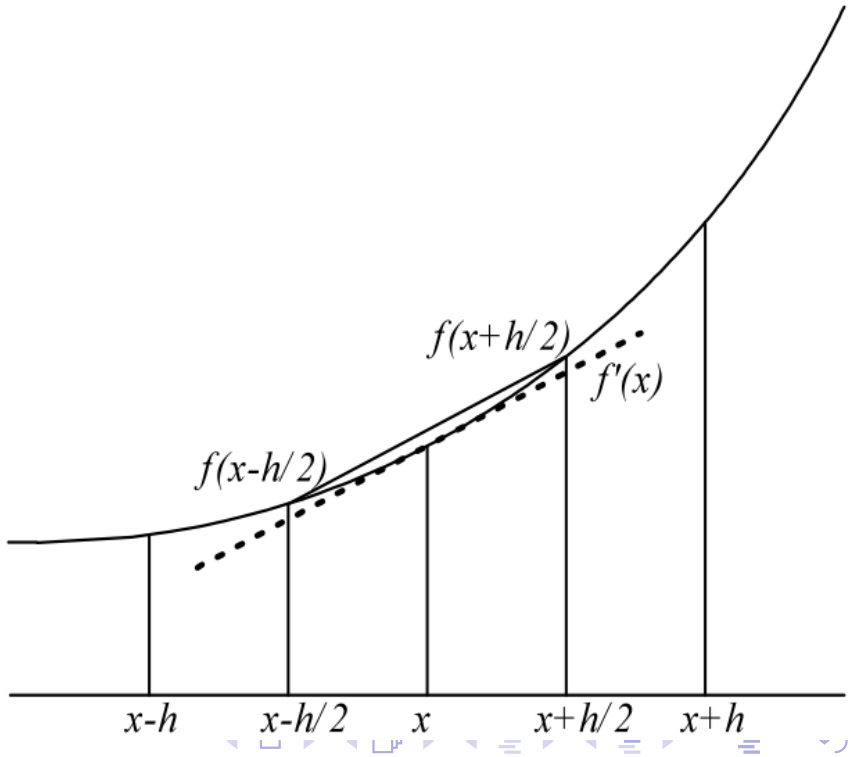
In general, not possible to obtain an analytical solution to a PDE.

Finite Difference Methods

Numerical methods that obtain an approximate result of PDEs by dividing the variables (often time and space) into discrete intervals.

Approximation to the first derivative of $f(x)$, $f'(x)$:

$$f'(x) \approx \frac{f(x + h/2) - f(x - h/2)}{h}$$



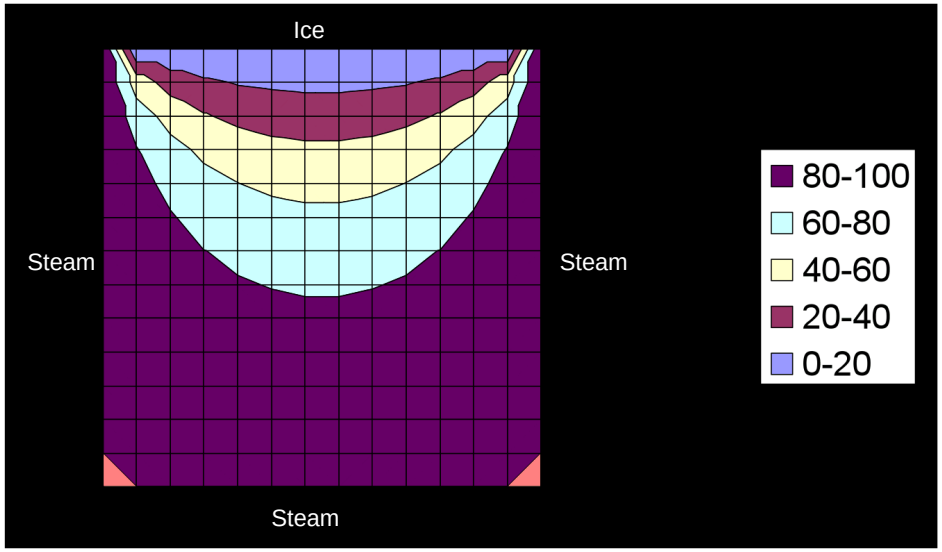
Finite Difference Methods

Approximation to the second derivative of $f(x)$, $f''(x)$:

$$
\begin{aligned}
f''(x) &\approx \frac{\frac{f(x+h/2+h/2)-f(x+h/2-h/2)}{h} - \frac{f(x-h/2+h/2)-f(x-h/2-h/2)}{h}}{h} \\
&= \frac{f(x+h) - f(x) - (f(x) - f(x-h))}{h^2} \\
&= \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}
\end{aligned}
$$

Steady-State Heat Distribution

Problem: find the steady-state heat distribution over a thin square plate.



Underlying PDE is the Poisson Equation:

$$u_{xx} + u_{yy} = f(x, y) \qquad 0 \leq x \leq a, 0 \leq y \leq b$$

Boundary conditions:

$$u(x, 0) = G_1(x) \quad u(x, a) = G_2(x) \quad 0 \leq x \leq a$$
$$u(0, y) = G_3(y) \quad u(a, y) = G_4(y) \quad 0 \leq y \leq b$$

Steady-State Heat Distribution

Sequential program: create a 2-dimensional grid, with dimensions x and y divided into n and m pieces, respectively. Iteratively compute $u(x_i, y_j)$ until convergence.

Let $h = x/n$ and $k = y/m$.

$$\begin{aligned} u_{xx}(x_i, y_j) &\approx \frac{u(x_i + h, y_j) - 2u(x_i, y_j) + u(x_i - h, y_j)}{h^2} \\ &= \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \end{aligned}$$

$$u_{yy}(x_i, y_j) \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2}$$

Inserting in the original equation:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} = f_{i,j}$$

Steady-State Heat Distribution

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} = f_{i,j}$$

In our case,

- no internal sources, $f_{i,j} = 0$
- laterals and bottom at 100°, $u_{i,1} = u_{1,j} = u_{a,j} = 100$
- top at 0°, $u_{i,b} = 0$
- also, if we use $h = k$

simplifies to:

$$u_{i+1,j} - 2u_{i,j} + u_{i-1,j} + u_{i,j+1} - 2u_{i,j} + u_{i,j-1} = 0$$

Solving for $u_{i,j}$:

$$u_{i,j} = \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{4}$$

Steady-State Heat Distribution

$$u_{i,j} = \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{4}$$

Jacobi method:

- start with an initial guess of $u_{i,j}$
- compute $u_{i,j}$ at iteration q from values of $q - 1$
- stop when values between iterations within a given error

Parallel Heat Distribution

Primitive task:

Parallel Heat Distribution

Primitive task: $u_{i,j}$ in each iteration

Communication:

Parallel Heat Distribution

Primitive task: $u_{i,j}$ in each iteration

Communication: adjacent cells, $u_{i-1,j}$, $u_{i+1,j}$, $u_{i,j-1}$, $u_{i,j+1}$

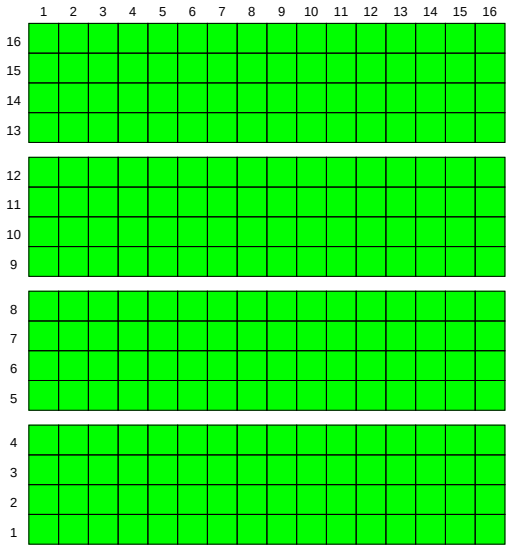
Agglomeration:

Parallel Heat Distribution

Primitive task: $u_{i,j}$ in each iteration

Communication: adjacent cells, $u_{i-1,j}$, $u_{i+1,j}$, $u_{i,j-1}$, $u_{i,j+1}$

Agglomeration: row-wise block stripped



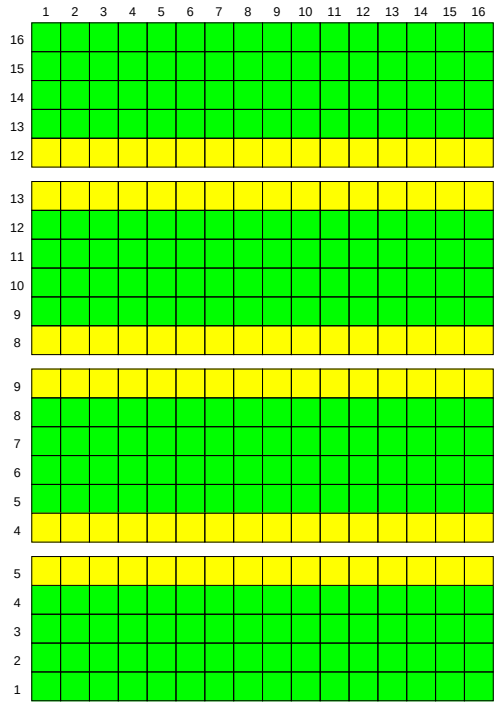
Sideways within process, what about top/bottom?

Parallel Heat Distribution

Primitive task: $u_{i,j}$ in each iteration

Communication: adjacent cells, $u_{i-1,j}$, $u_{i+1,j}$, $u_{i,j-1}$, $u_{i,j+1}$

Agglomeration: row-wise block stripped



At the end of each iteration, each process sends the top and bottom lines.

Ghost Points

Ghost Points

Memory locations used to store redundant copies of data held by neighboring processes

Allocating ghost points as extra rows / columns simplifies parallel algorithms by allowing same loop to update all cells.

Ghost Points

Ghost Points

Memory locations used to store redundant copies of data held by neighboring processes

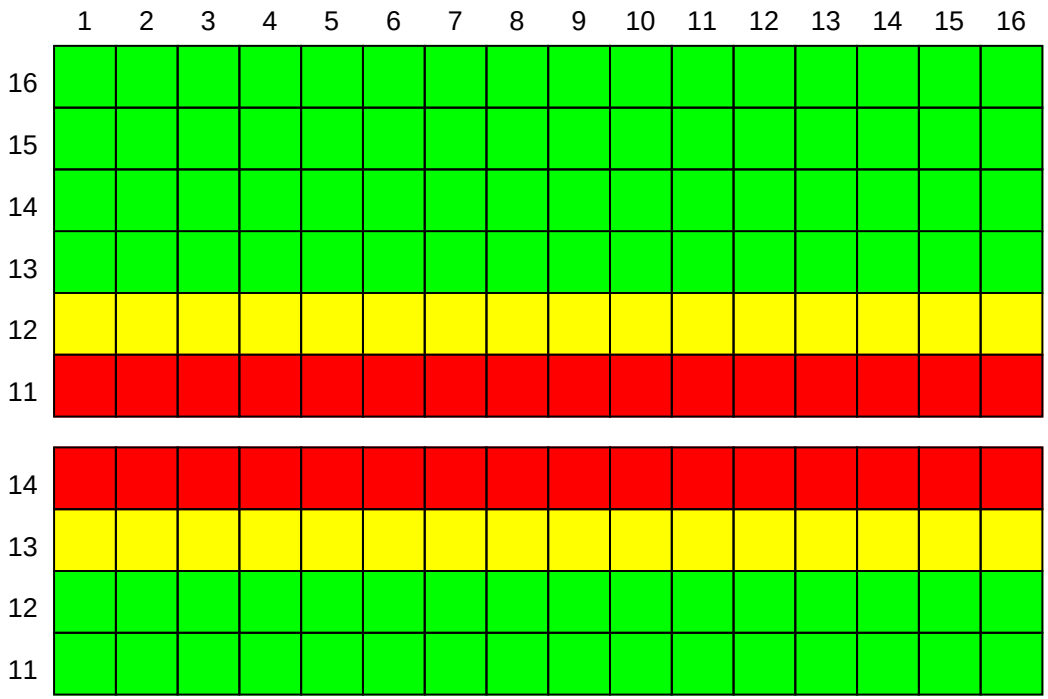
Allocating ghost points as extra rows / columns simplifies parallel algorithms by allowing same loop to update all cells.

How many rows for ghost points?

Ghost Points

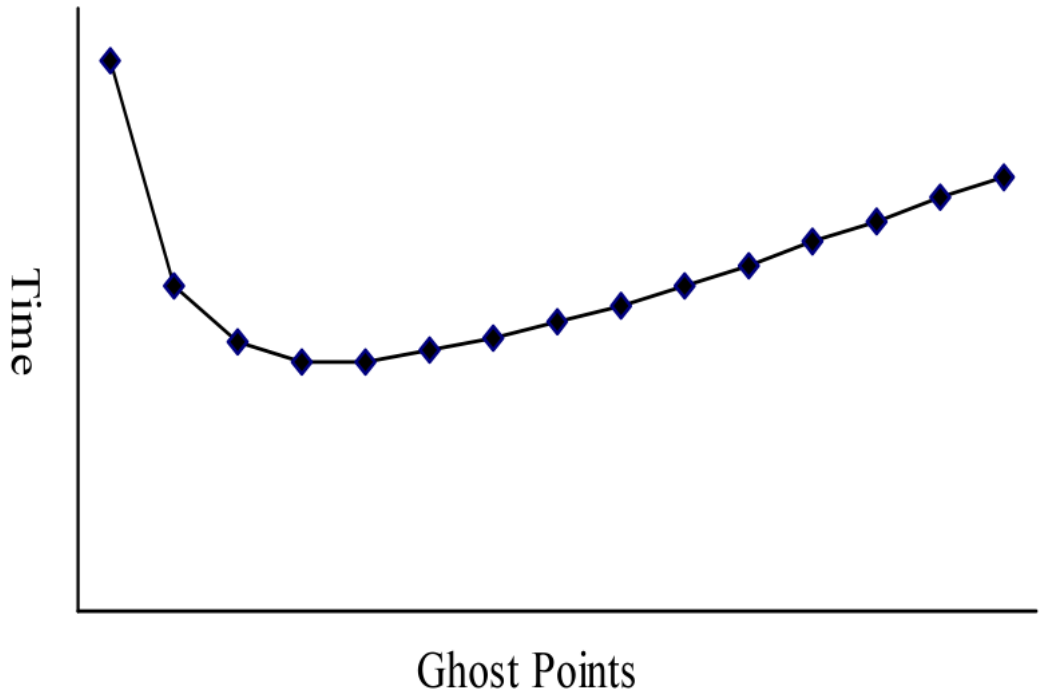
- if only one row/column transmitted, communication time may be dominated by message latency
- if we send two, we can advance simulation two time steps before another communication

  $\Rightarrow$ however we need to replicate computation



Ghost Points

- if only one row/column transmitted, communication time may be dominated by message latency
- if we send two, we can advance simulation two time steps before another communication
  $\Rightarrow$ however we need to replicate computation



Complexity Analysis of Row-wise Parallel Algorithm

(analysis per iteration, and a single ghost row)

Sequential time complexity:

Complexity Analysis of Row-wise Parallel Algorithm

(analysis per iteration, and a single ghost row)

Sequential time complexity: $\Theta(n^2)$

Parallel computational complexity:

Complexity Analysis of Row-wise Parallel Algorithm

(analysis per iteration, and a single ghost row)

Sequential time complexity: $\Theta(n^2)$

Parallel computational complexity: $\Theta(\frac{n^2}{p})$

Parallel communication complexity:

Complexity Analysis of Row-wise Parallel Algorithm

(analysis per iteration, and a single ghost row)

Sequential time complexity: $\Theta(n^2)$

Parallel computational complexity: $\Theta(\frac{n^2}{p})$

Parallel communication complexity: $\Theta(n)$
(two sends and two receives of n elements)

Isoefficiency Analysis of Row-wise Parallel Algorithm

Isoefficiency analysis: $T(n,1) \geq CT_0(n,p)$

($T(n,1)$ sequential time; $T_0(n,p)$ parallel overhead)

Sequential time complexity: $T(n,1) = \Theta(n^2)$

Parallel overhead dominated by communication: $\Theta(n)$

$$T_0(n,p) = p \times \Theta(n) = \Theta(pn)$$

Isoefficiency Analysis of Row-wise Parallel Algorithm

Isoefficiency analysis: $T(n,1) \geq CT_0(n,p)$

($T(n,1)$ sequential time; $T_0(n,p)$ parallel overhead)

Sequential time complexity: $T(n,1) = \Theta(n^2)$

Parallel overhead dominated by communication: $\Theta(n)$

$$T_0(n,p) = p \times \Theta(n) = \Theta(pn)$$

$$n^2 \geq Cpn \quad \Rightarrow \quad n \geq Cp$$

Isoefficiency Analysis of Row-wise Parallel Algorithm

Isoefficiency analysis: $T(n,1) \geq CT_0(n,p)$

($T(n,1)$ sequential time; $T_0(n,p)$ parallel overhead)

Sequential time complexity: $T(n,1) = \Theta(n^2)$

Parallel overhead dominated by communication: $\Theta(n)$

$$T_0(n,p) = p \times \Theta(n) = \Theta(pn)$$

$$n^2 \geq Cpn \quad \Rightarrow \quad n \geq Cp$$

Scalability function: $M(f(p))/p$

$$M(n) = n^2 \quad \Rightarrow \quad \frac{M(Cp)}{p} = \frac{C^2p^2}{p} = C^2p$$

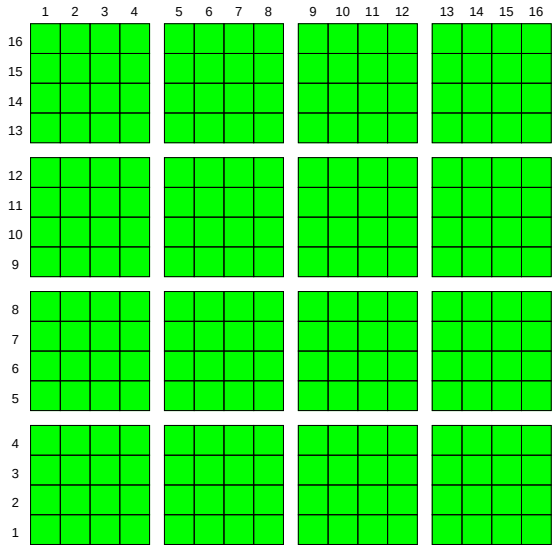
$\Rightarrow$ System is not highly scalable.

Parallel Heat Distribution

Primitive task: $u_{i,j}$ in each iteration

Communication: adjacent cells, $u_{i-1,j}$, $u_{i+1,j}$, $u_{i,j-1}$, $u_{i,j+1}$

Agglomeration: checkerboard block decomposition



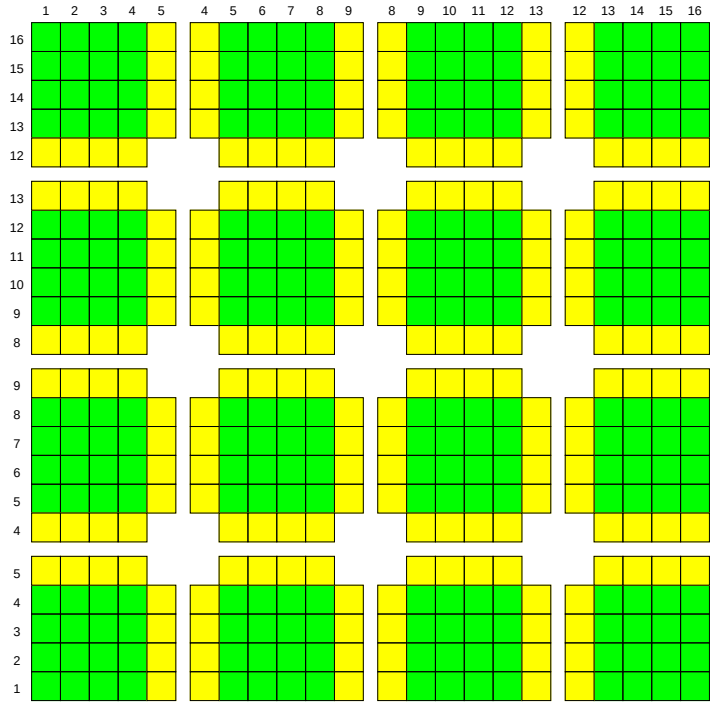
Ghost points?

Parallel Heat Distribution

Primitive task: $u_{i,j}$ in each iteration

Communication: adjacent cells, $u_{i-1,j}$, $u_{i+1,j}$, $u_{i,j-1}$, $u_{i,j+1}$

Agglomeration: checkerboard block decomposition



At the end of each iteration, each process sends the top, bottom and sides.

Complexity Analysis of Checkerboard Decomposition

(analysis per iteration, and a single ghost row/column)

Sequential time complexity: $\Theta(n^2)$

Parallel computational complexity: $\Theta(\frac{n^2}{p})$

Parallel communication complexity:

Complexity Analysis of Checkerboard Decomposition

(analysis per iteration, and a single ghost row/column)

Sequential time complexity: $\Theta(n^2)$

Parallel computational complexity: $\Theta(\frac{n^2}{p})$

Parallel communication complexity: $\Theta(n/\sqrt{p})$
(four sends and four receives of $\frac{n}{\sqrt{p}}$ elements)

Isoefficiency Analysis of Checkerboard Parallel Algorithm

Isoefficiency analysis: $T(n,1) \geq CT_0(n,p)$

($T(n,1)$ sequential time; $T_0(n,p)$ parallel overhead)

Sequential time complexity: $T(n,1) = \Theta(n^2)$

Parallel overhead dominated by communication: $\Theta(n/\sqrt{p})$

$$T_0(n,p) = p \times \Theta(n/\sqrt{p}) = \Theta(n\sqrt{p})$$

Isoefficiency Analysis of Checkerboard Parallel Algorithm

Isoefficiency analysis: $T(n,1) \geq CT_0(n,p)$

($T(n,1)$ sequential time; $T_0(n,p)$ parallel overhead)

Sequential time complexity: $T(n,1) = \Theta(n^2)$

Parallel overhead dominated by communication: $\Theta(n/\sqrt{p})$

$$T_0(n,p) = p \times \Theta(n/\sqrt{p}) = \Theta(n\sqrt{p})$$

$$n^2 \geq Cn\sqrt{p} \quad \Rightarrow \quad n \geq C\sqrt{p}$$

Isoefficiency Analysis of Checkerboard Parallel Algorithm

Isoefficiency analysis: $T(n,1) \geq CT_0(n,p)$

($T(n,1)$ sequential time; $T_0(n,p)$ parallel overhead)

Sequential time complexity: $T(n,1) = \Theta(n^2)$

Parallel overhead dominated by communication: $\Theta(n/\sqrt{p})$

$$T_0(n,p) = p \times \Theta(n/\sqrt{p}) = \Theta(n\sqrt{p})$$

$$n^2 \geq Cn\sqrt{p} \quad \Rightarrow \quad n \geq C\sqrt{p}$$

Scalability function: $M(f(p))/p$

$$M(n) = n^2 \quad \Rightarrow \quad \frac{M(C\sqrt{p})}{p} = \frac{C^2 p}{p} = C^2$$

$\Rightarrow$ System is perfectably Scalable!

Review

- Iterative Methods for Linear Systems
  - Relaxation Methods
- Linear Second-order Partial Differential Equations (PDEs)
  - Finite difference methods
  - Example: steady-state heat distribution
  - Ghost points

Next Class

- Map-Reduce
- ccNUMA