

Parallel Laplace Solver with MPI

Shaohao Chen

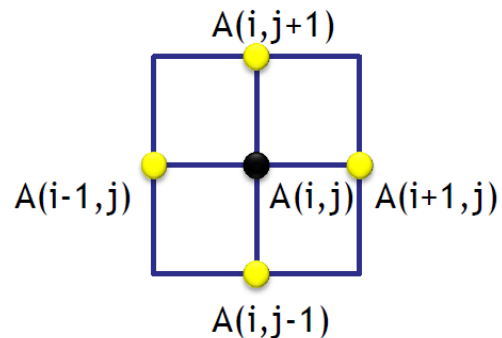
ORCD at MIT

Laplace solver (1)

- Two-dimensional Laplace equation: $\nabla^2 f(x, y) = 0$
- Discretize the Laplacian using first-order differential method and obtain the solution as following,

$$A_{k+1}(i, j) = \frac{A_k(i-1, j) + A_k(i+1, j) + A_k(i, j-1) + A_k(i, j+1)}{4}$$

- The solution on one grid only depends on the four neighbor grids:



Laplace solver (2)

- Use iterative algorithm to obtain a converged solution.
- Jacobi iterative algorithm:
 1. Give a trial solution A based on a provided initial condition.
 2. Calculate a new solution, that is $A_{\text{new}}(i,j)$, using the old values of the four neighbor points that are stored in A .
 3. Update the solution: $A=A_{\text{new}}$.
 4. Iterate steps 2 and 3 until converged, i.e. $\max(|A_{\text{new}}(i,j)-A(i,j)|)<\text{tolerance}$.
 5. Finally the converged solution is stored in A .

Serial code in C (kernel)

```
while ( dA > tolerance && iteration <= max_iterations ) {    // do until error is minimal or until max steps
    for(i = 1; i <= ROWS; i++)        // main calculation: average my four neighbors
        for(j = 1; j <= COLUMNS; j++) {
            A_new[i][j] = 0.25 * (A[i+1][j] + A[i-1][j] + A[i][j+1] + A[i][j-1]);
        }
    dA = 0.0;        // reset largest change
    for(i = 1; i <= ROWS; i++)
        for(j = 1; j <= COLUMNS; j++){
            dA = fmax( fabs(A_new[i][j]-A[i][j]), dA);    // find the latest change
            A[i][j] = A_new[i][j];        // copy grid to old grid for next iteration
        }
    iteration++;
}
```

Serial code in Fortran (kernel)

```
do while ( dA > tolerance .and. iteration <= max_iterations)    ! do until error is minimal or until max steps
  do j=1,columns          ! main calculation: average my four neighbors
    do i=1,rows
      A_new(i,j)=0.25*(A(i+1,j)+A(i-1,j)+A(i,j+1)+A(i,j-1) )
    enddo
  enddo
  dA=0.0                  ! reset largest change
  do j=1,columns
    do i=1,rows
      dA = max( abs(A_new(i,j) - A(i,j)), dA )    ! find the latest change
      A(i,j) = A_new(i,j)          ! copy grid to old grid for next iteration
    enddo
  enddo
  iteration = iteration+1
enddo
```

Parallelize with MPI

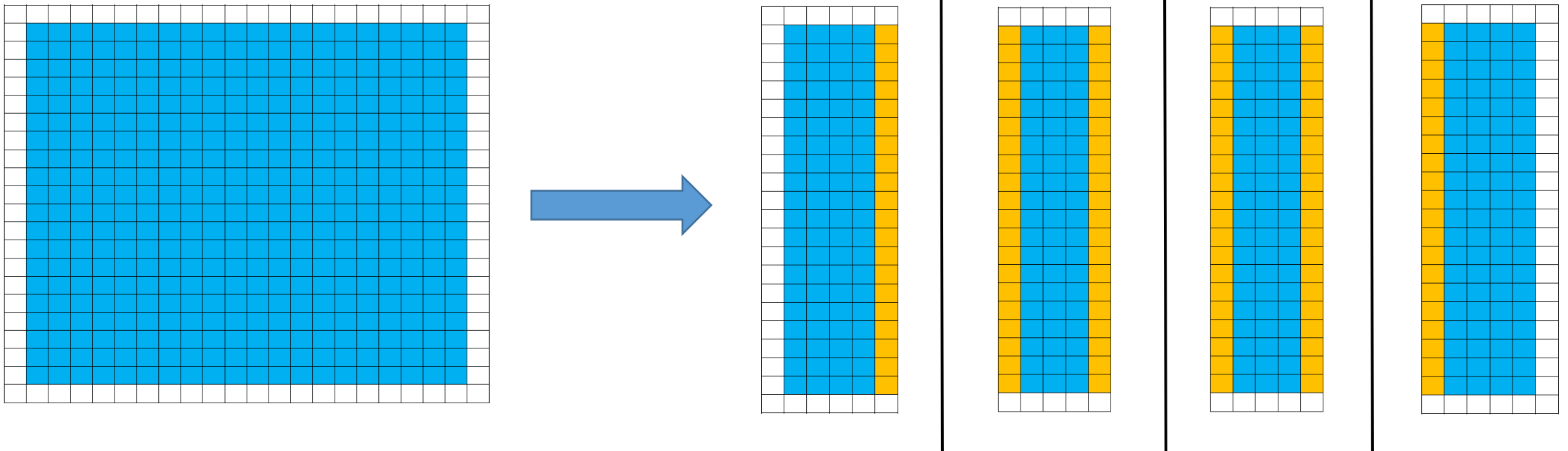
Analysis for parallelism:

1. Find the “hot spots”, the most time-consuming parts of the code.
2. Decompose the grids into sub-grids. Each process owns one sub-grid.
3. Pass necessary data between processes. (e.g. use `MPI_Send` and `MPI_Recv`). Be careful to avoid dead locks.
4. Pass “shared” data between the root process and all other process (e.g. use `MPI_Bcast` and `MPI_Reduce`).

Domain Decomposition (1)

- 1D Decomposition:

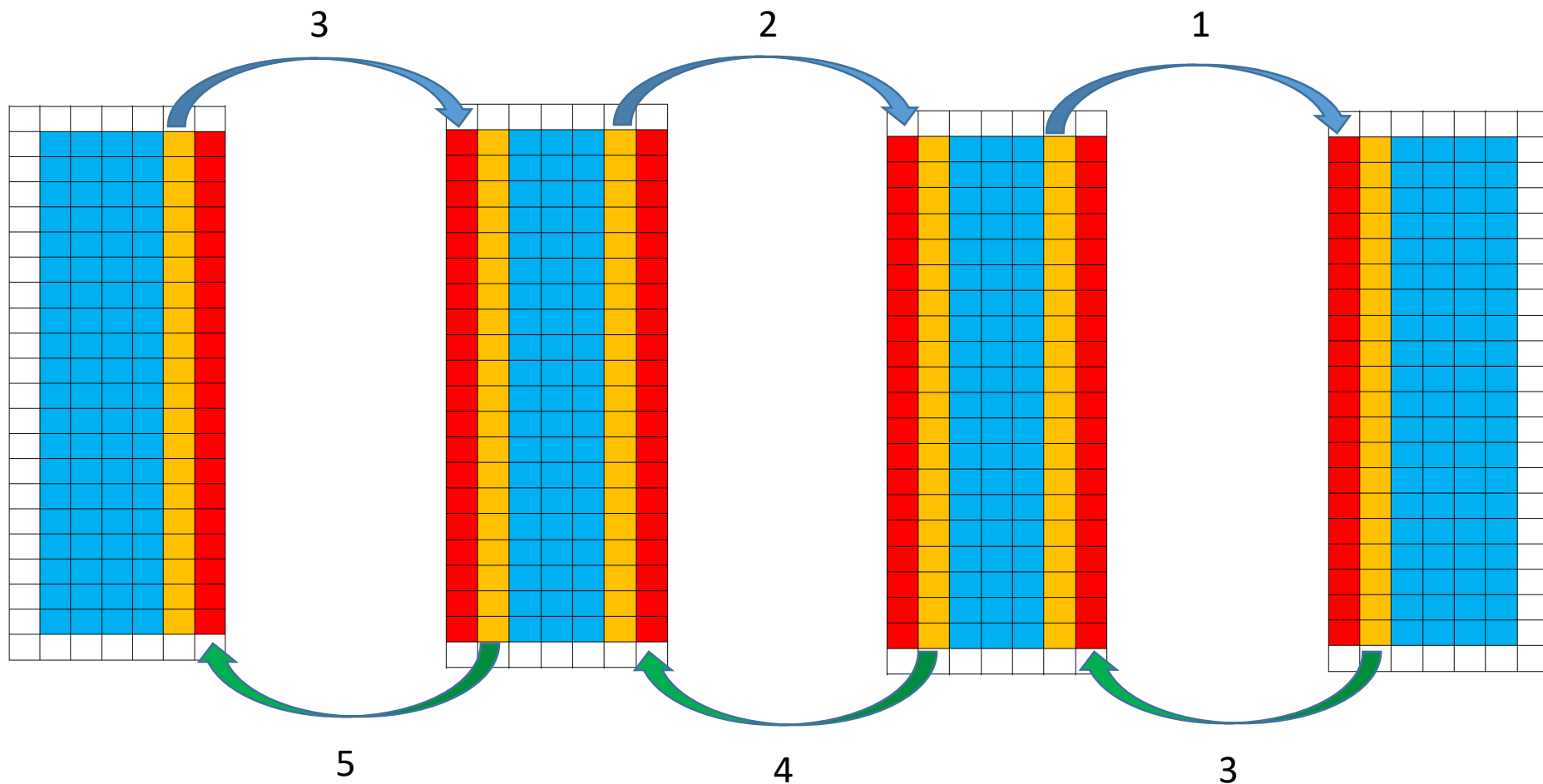
Divide columns in Fortran or divide rows in C.



- ✓ Blue zone: The real grids.
- ✓ White zone: An additional layer for boundary condition.
- ✓ Yellow zone: The data on these grids need to be sent to neighbor process(es).

Ghost Zone (1)

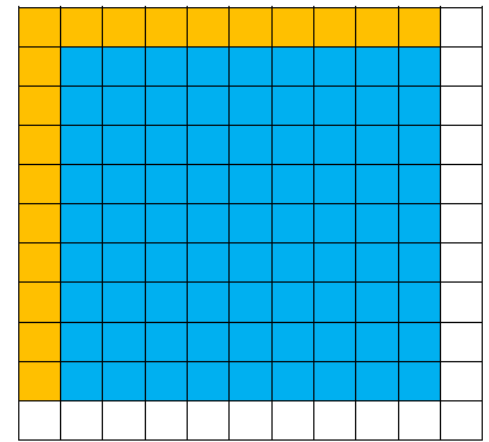
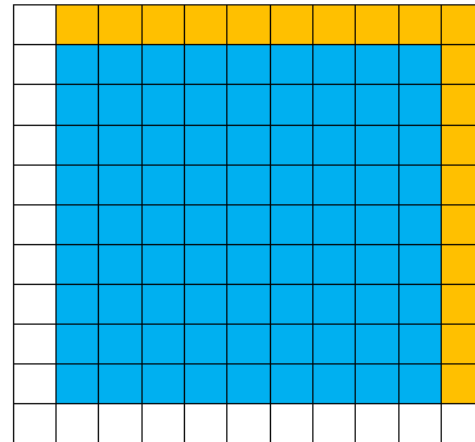
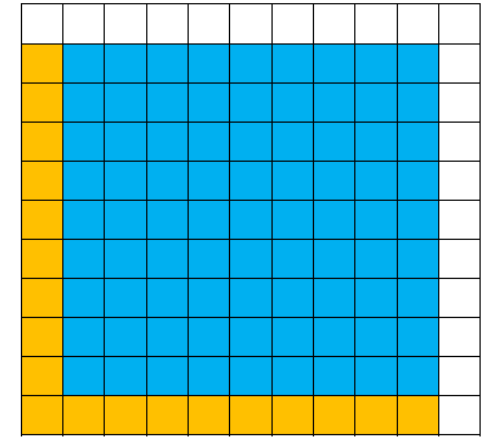
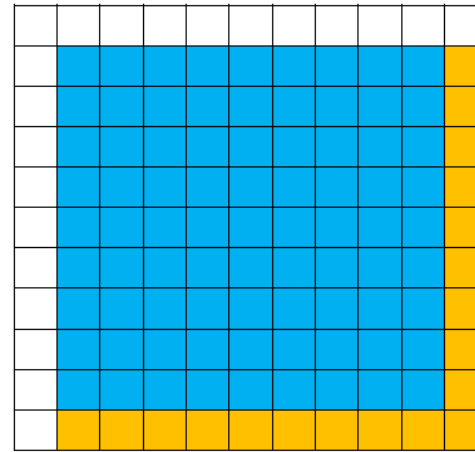
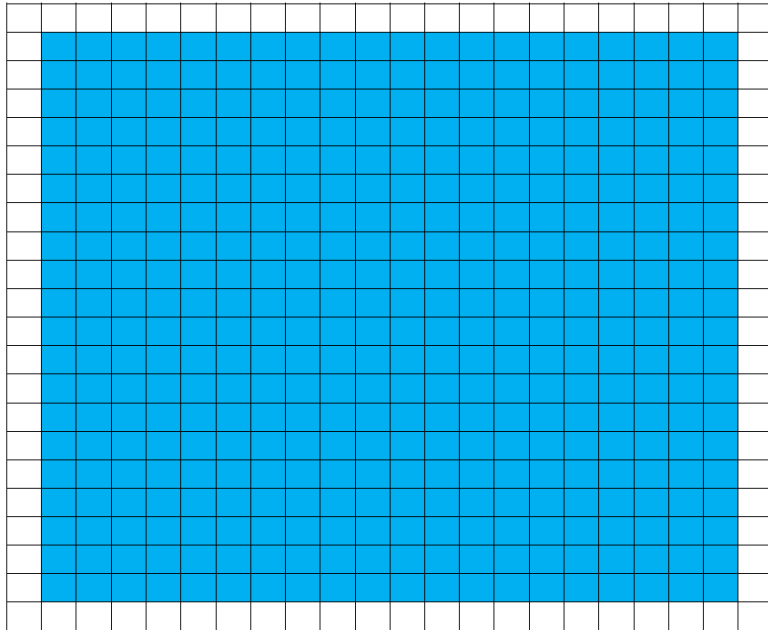
- ✓ Red zone: An additional layer to receive the data from neighbor process(es). Also called “ghost zone”.



Domain Decomposition (2)

- 2D Decomposition:

Divide both rows and columns



Ghost Zone (2)

- ✓ Send and receive row-type and column-type data.

