HPC Summer School
Computational Fluid Dynamics Simulation and its Parallelization

Kentaro Sano
Processor Research Team, R-CCS Riken

Agenda

- **PART-I**
  - Introduction of Application: 2D CFD Simulation
    - Lecture
    - Hands-on Practice
- **PART-II**
  - Parallelization of the 2D CFD Simulation
    - Lecture
    - Hands-on Practice
PART-I
Introduction of Application:
2D CFD Simulation

Introduction

- What’s CFD (Computational Fluid Dynamics) simulation?

A computer simulation of high velocity air flow around the Space Shuttle during re-entry.

2D CFD example
How to Compute Fluid Flow?

Repeating grid update for \( \Delta t \) fluid change. How to update?

Incompressive Viscous Fluid Flow

Governing Equations with partial differential equations

\[
\nabla V = 0
\]

\[
\frac{\partial V}{\partial t} + (V \cdot \nabla)V = -\nabla \varphi + \nu \nabla^2 V
\]

Equation of continuity (incompressive flow)

Navier–Stokes equations (incompressive flow)

\[ V \text{ velocity } = (u, v) \]
\[ \nu \equiv \frac{\mu}{\rho} \text{ kinematic viscosity} \]
\[ P \text{ pressure} \]
\[ \varphi \equiv \frac{P}{\rho} \]
Fractional-Step Method

1. Calculate the tentative velocity \( V^* \) without the pressure-term.

\[
V^* = V^n + \Delta t \left\{ -(V^n \cdot \nabla)V^n + \nu \nabla^2 V^n \right\} \tag{1}
\]

2. Calculate the pressure field \( \phi^{n+1} \) of the next time-step with \( V^* \) by solving the Poisson’s equation.

\[
\nabla^2 \phi^{n+1} = \frac{\nabla \cdot V^*}{\Delta t} \tag{2}
\]

3. Calculate the true velocity \( V^{n+1} \) of the next time-step with \( V^* \) and \( \phi \).

\[
V^{n+1} = V^* - \Delta t \nabla \phi^{n+1} \tag{3}
\]

Finite Difference Schemes

We can make discrete forms by substituting difference schemes.

\[
\frac{du}{dx} \approx \frac{u_{i+1} - u_{i-1}}{2\Delta x}
\]

\[
\frac{d^2 u}{dx^2} \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{(\Delta x)^2}
\]

Central difference schemes
(\( \Rightarrow \) Finite difference scheme)

2D collocate mesh
(Each grid point has all variables: \( u, v, \phi \).)

See “staggered mesh” for more advanced study.
**Discrete Form of Step 1**

**Step 1: Calculate the tentative velocity: \( \mathbf{u^*, v^*} \)**

\[
\begin{align*}
\mathbf{u^*}_{i,j} &= u_{i,j} + \Delta t \left\{ -u_{i,j} \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} - v_{i,j} \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} + \frac{NU}{(\Delta x)^2} \left( \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} \right) \right\}
\end{align*}
\]

\( NU \) is kinematic viscosity.

A similar equation for \( v \).

**Discrete Form of Step 2**

**Step 2: Calculate the pressure by Jacobi method.**

Iterating phi’s update until residual met a certain condition.

\[
\begin{align*}
\phi'_{i,j} &= \alpha \left( \frac{\phi_{i+1,j} + \phi_{i-1,j}}{(\Delta x)^2} + \frac{\phi_{i,j+1} + \phi_{i,j-1}}{(\Delta y)^2} - D_{i,j} \right)
\end{align*}
\]

where

\[
\alpha = \frac{(\Delta x^2)(\Delta y^2)}{2(\Delta x^2 + \Delta y^2)}
\]

and

\[
D_{i,j} = \frac{1}{\Delta t} \left( \frac{u_{i+1,j}^* - u_{i-1,j}^*}{2\Delta x} + \frac{v_{i,j+1}^* - v_{i,j-1}^*}{2\Delta y} \right)
\]

\( D_{i,j} \) is referred to as a source term of Poisson’s equation.
Discrete Form of Step 3

Step 3: Calculate the true velocity of the next time-step

\[ u_{i,j}^{\text{true}} = u_{i,j}^* - \Delta t \frac{\varphi_{i+1,j}^t - \varphi_{i-1,j}^t}{2\Delta x} \]

\[ v_{i,j}^{\text{true}} = v_{i,j}^* - \Delta t \frac{\varphi_{i,j+1}^t - \varphi_{i,j-1}^t}{2\Delta y} \]

Stencil Computation

Common form in Steps 1, 2, and 3

\[ q_{i,j}^{\text{new}} = A + B q_{i,j} + C q_{i+1,j} + D q_{i-1,j} + E q_{i,j+1} + F q_{i,j-1} \]

Each point is computed only with its adjacent points.

Stencil (adjacent region of each point)
Hands-on : Let’s read the codes!

klogin5$ mkdir programs
klogin5$ cd programs
klogin5$ cp /scratch/ra001016/day2/serial_0703.tgz ./
klogin5$ tar zxfv serial_0703.tgz
klogin5$ cd serial_0703/

Source files – You modify them!
(program codes)

klogin5$ ls

Source files – You modify them!
(program codes)

Rules for compilation with “make”

Information on how to compile, execute, etc.

Script programs for execution with K-computer

Program Structure

- **Data structures (cfd.h)**
  - typedef struct array2D_ : array2D;
  - typedef struct grid2D_ : grid2D;

- **Functions (member functions of the data-structures)**
  - void array2D_initialize(array2D *a, …); // initialize 2D array : row x col
  - void array2D_resize(array2D *a, …); // resize 2D array : row x col
  - void array2D_copy(array2D *a, …); // copy src to dst (by resizing dst)
  - void array2D_clear(array2D *a, …); // clear 2D array with value of v
  - void array2D_show(array2D *a, …); // print 2D array in text
  - double linear_intp(array2D *a, …); // get value with linear interpolation
  - inline int array2D_getRow(array2D *a, …); // get size of row
  - inline int array2D_getCol(array2D *a, …); // get size of col
  - inline double *at(array2D *a, …); // get pointer at (row, col)
  - inline double L(array2D *a, …) // Look up value at (row, col)
Program Structure (cont’d)

- **Data structures (cfd.h)**
  - typedef struct array2D_: array2D;
  - typedef struct grid2D_: grid2D;

- **Functions (member functions of the data-structures)**
  - void grid2D_initialize(grid2D *g, …);
  - void grid2D_calcTantVelocity(grid2D *g);
  - void grid2D_calcPoissonSourceTerm(grid2D *g);
  - void grid2D_calcPoisson_Jacobi(grid2D *g, , …);
  - void grid2D_calcVelocity(grid2D *g);
  - void grid2D_calcBoundary_Poiseulle(grid2D *g, , …);
  - void grid2D_calcBoundary_SqObject(grid2D *g, , …);
  - void grid2D_outputAVEseFile(grid2D *g, , …);
  - inline int  grid2D_getRow(grid2D *g);
  - inline int  grid2D_getCol(grid2D *g);

---

Dependency among Steps

![Diagram showing dependencies among steps](image)

- **CalcTantVelocity**
- **CalcPoissonSourceTerm**
- **CalcPoisson Jacobi** (repeated)
- **CalcVelocity**
main.h

/*
  * 2D fluid simulation based on Fractional-step method
  * Written by Kentaro Sano for
  * International Summer school, RIKEN R-CCS
  *
  * Version 2018_0611_1339
  *
  * All rights reserved.
  * (C) Copyright Kentaro Sano 2018.6-
  *
  */
#endif

#include "main.h"
int main(int argc, char** argv)
{
  grid2D g;
  std::StopWatchClass2 time;
  tstep = 0;
  grid2D_initialize(&g, ROW, COL, PHI_IN, PHI_OUT);

  while(tstep < END_TIMESTEP) {
    fractionalStep_MainLoop(&g, SAVE_INTERVAL);
    grid2D_outputAVEseFile(&g, "AVEse", tstep, 1.0);
    time.stop();
    printf("Time-step=%d : ElapsedTime=%3.3f sec\n",
           tstep, time.get());
    return 0;
  }
}
void fractionalStep_MainLoop(grid2D *g, int numTSteps)
{
  for (int n=0; n<numTSteps; n++) {
    grid2D_calcTantVelocity(g);
    grid2D_calcPoissonSourceTerm(g);
    grid2D_calcPoisson_Jacobi(g, TARGET_RESIDUAL_RATE);
    grid2D_calcVelocity(g);
    grid2D_calcBoundary_Poiseulle(g, PHI_IN, PHI_OUT);
    grid2D_calcBoundary_SqObject(g, OBJ_X, OBJ_Y, OBJ_W, OBJ_H);
    tstep++;
  }
}
cfd.h 1 of 4

#include <stopwatch2.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cfd.h"

int main(int argc, char** argv);
void fractionalStep_MainLoop(grid2D *g, int numTSteps);
#include "main.h"
#define TARGET_RESIDUAL_RATE (1.0e-2) // Termination condition: 1.0e-3 recommended rather than 1.0e-1)
#define JACOBIITER_INTERVAL (200) // interval of time-step to report # of Jacobi-loop iterations
#define END_TIMESTEP (20000) // time-step to end computation
#define SAVE_INTERVAL (200) // Timestep interval to save flow-field files (*.dat)
//========================================================================
#define HEIGHT 0.5 // Grid Height is set a length of 0.5 (dimensionless length)
#define WIDTH (0.5*(double)ROW/(double)COL) // Width is calculated with the ratio of ROW to COL
#define DX (WIDTH/(ROW-1))
#define DY (HEIGHT/(ROW-1))
#define DX2 (DX*DX)
#define DY2 (DY*DY)

// Boundary conditions for Poiseuille flow
#define U_IN (1.0) // X velocity of inlet (incoming) flow (unused)
#define V_IN (0.0) // Y velocity of inlet (incoming) flow (unused)
#define PHI_IN (200.0) // Pressure of inlet (incoming boundary)
#define PHI_OUT (100.0) // Pressure of outlet (outgoing boundary)

// Rectangle object for internal boundary
#define OBJ_X (ROW*0.25) // X-center of object
#define OBJ_Y (COL*0.5) // Y-center of object
#define OBJ_W (COL*0.2) // Width (in x) of object
#define OBJ_H (COL*0.30) // Height (in y) of object

// Global variables
extern int tstep; // time-step

// Definition of data structure (grid and common variables)

// Data structure of 2D array (resizable)
typedef struct array2D {
    int row; // ROW resolution of a grid
    int col; // COL resolution of a grid
    double *v; // Pointer of 2D array
} array2D;

// Member functions for array2D
void array2D_initialize(array2D *a, int row, int col); // initialize 2D array : row x col
void array2D_resize(array2D *a, int row, int col); // resize 2D array : row x col
void array2D_copy(array2D *src, array2D *dst); // copy src to dst (by resizing dst)
void array2D_clear(array2D *a, double v); // clear 2D array with value of v
void array2D_show(array2D *a); // print 2D array in text
double linear_intp(array2D *a, double x, double y); // get value at (x,y) with linear interpolation
inline int array2D_getRow(array2D *a) { return (a->row); } // get size of row
inline int array2D_getCol(array2D *a) { return (a->col); } // get size of col
inline double *at(array2D *a, int i, int j) { return *(a->v + i + j * a->row); } // get pointer at (row, col)

if ((i<0) || (j<0) || (i>=a->row) || (j>=a->col)) {
    printf("Out of range : (%d, %d) for %d x %d array in at(). Abort.
    "); i, j, a->row, a->col);
    exit(EXIT_FAILURE);
}
return (a->v + i + j * a->row);
}
inline double L(array2D *a, int i, int j) { return *(at(a,i,j)); } // Look up value at (row, col)
// Data structure of 2D grid for fluid flow
typedef struct grid2D_
{
    array2D u, v, phi;    // velocity (u, v), pressure phi
    array2D phiTemp;      // tentative pressure (temporary for update)
    array2D uTant, vTant; // tentative velocity (u, v)
    array2D d;            // source term of a pressure poisson's equation
} grid2D;

// Member functions for grid2D
void grid2D_initialize(grid2D *g, int row, int col, double phi_in, double phi_out);
void grid2D_calcTantVelocity(grid2D *g);
void grid2D_calcPoissonSourceTerm(grid2D *g);
void grid2D_calcPoisson_Jacobi(grid2D *g, double target_residual_rate);
void grid2D_calcVelocity(grid2D *g);
void grid2D_calcBoundary_Poiseulle(grid2D *g, double phi_in, double phi_out);
void grid2D_calcBoundary_SqObject(grid2D *g, int obj_x, int obj_y, int obj_w, int obj_h);
void grid2D_outputAVEseFile(grid2D *g, char *base, int num, double scaling);
inline int grid2D_getRow(grid2D *g) { return( array2D_getRow(&(g->u)) ); }
inline int grid2D_getCol(grid2D *g) { return( array2D_getCol(&(g->u)) ); }
#endif

#include "cfd.h"

int tstep; // time-step

// Member functions for array2D
void array2D_initialize(array2D *a, int row, int col)
{
    a->row = 0;
    a->col = 0;
    a->v = (double*)NULL;
    array2D_resize(a, row, col);
    array2D_clear(a, 0.0);
}

void array2D_resize(array2D *a, int row, int col)
{
    if (a->v != (double*)NULL) free(a->v);
    if ((row*col) <= 0) a->v = (double*)NULL;
    else
    {
        a->v = (double*)malloc(row * col * sizeof(double));
        a->row = row;
        a->col = col;
    }
    if (a->v == NULL) {
        printf("Failed with malloc() in array2D_resize().\n");
        exit(EXIT_FAILURE);
    }
}
void array2D_copy(array2D *src, array2D *dst)
{
    if ( (array2D_getRow(src) != array2D_getRow(dst)) ||
        (array2D_getCol(src) != array2D_getCol(dst)) )
        array2D_resize(dst, src->row, src->col);
    for (int j=0; j<(dst->col); j++)
        for (int i=0; i<(dst->row); i++) *(at(dst, i, j)) = L(src, i, j);
}

void array2D_clear(array2D *a, double v)
{
    for (int j=0; j<(a->col); j++)
        for (int i=0; i<(a->row); i++) *(at(a, i, j)) = v;
}

void array2D_show(array2D *a)
{
    printf("2D Array of %d x %d (%d elements)\n", a->row, a->col, a->row * a->col);
    for (int j=0; j<(a->col); j++)
    {
        printf("j=%4d : ", j);
        for (int i=0; i<(a->row); i++)
            printf(" %3.1f", *(at(a, i, j)));
        printf("\n");
    }
}

double linear_intp(array2D *a, double x, double y)
{
    int int_x = (int)x;
    int int_y = (int)y;
    double dx = x - (double)int_x;
    double dy = y - (double)int_y;
    double ret = 0.0;
    if ((x<0.0) || (y<0.0) || (x>=(double)(a->row - 1)) || (y>=(double)(a->col - 1)))
    {
        //printf("Out of range : (%f, %f) for %d x %d array in at(). Abort.\n", x, y, a->row, a->col);
        //exit(EXIT_FAILURE);
        return ret;
    }
    ret = ((double)L(a, int_x, int_y)*(1.0-dx) + (double)L(a, int_x+1, int_y )*dx)*(1.0-dy) +
          ((double)L(a, int_x, int_y+1)*(1.0-dx) + (double)L(a, int_x+1, int_y+1)*dx)*dy;
    return ret;
}
// Member functions for grid2D
void grid2D_initialize(grid2D *g, int row, int col, double phi_in, double phi_out)
{
    array2Dinitialize(&g->u, row, col);
    array2Dinitialize(&g->v, row, col);
    array2Dinitialize(&g->phi, row, col);
    //array2Dinitialize(&g->phiTemp, row+2, col+2); // for halo?
    array2Dinitialize(&g->phiTemp, row, col);
    array2Dinitialize(&g->uTant, row, col);
    array2Dinitialize(&g->vTant, row, col);
    array2Dinitialize(&g->d, row, col);
    array2Dclear (g->u, 0.01);
    array2Dclear (g->v, 0.01);
    array2Dclear (g->phi, 0.0);
    array2Dclear (g->phiTemp, 0.0);
    array2Dclear (g->uTant, 0.01);
    array2Dclear (g->vTant, -0.01);
    array2Dclear (g->d, 0.0);

    // Initialize the pressure field with constant gradient
    array2D a = &(g->phi);
    double row_minus_one = (double)array2DgetRow(a) - 1.0;
    for (int j=0; j<(a->col); j++)
        for (int i=0; i<(a->row); i++)
            *(at(a,i,j)) = phi_out * (double)i/row_minus_one + phi_in * (1.0 - (double)i/row_minus_one);
    // Update cells for boundary condition of Poiseulle flow
    grid2D_calcBoundary_Poiseulle(g, phi_in, phi_out);
}

void grid2D_calcTantVelocity(grid2D *g)
{
    array2D *u = &(g->u);
    array2D *v = &(g->v);
    array2D *uT = &(g->uTant);
    array2D *vT = &(g->vTant);
    int row_m_1 = array2D_getRow(u) - 1;
    int col_m_1 = array2D_getRow(u) - 1;
    int i, j;
    #pragma omp parallel for private(i)
    for (j=1; j<row_m_1; j++)
        for (i=1; i<col_m_1; i++)
            *(at(uT,i,j)) =
                L(u,i,j) + DT*(L(u,i,j)*(L(u,i+1,j) - L(u,i-1,j)) / 2.0 / DX
                         -L(v,i,j)*(L(v,i,j+1) - L(v,i,j-1)) / 2.0 / DY +
                         NU*(L(u,i+1,j) - 2.0*L(u,i,j) + L(u,i-1,j)) / DX2 +
                         L(u,i,j+1) - 2.0*L(u,i,j) + L(u,i,j-1)) / DY2);

    *(at(vT,i,j)) =
        L(v,i,j) + DT*(L(v,i,j)*(L(v,i+1,j) - L(v,i-1,j)) / 2.0 / DX
                    -L(v,i,j)*(L(v,i,j+1) - L(v,i,j-1)) / 2.0 / DY +
                    NU*(L(v,i+1,j) - 2.0*L(v,i,j) + L(v,i-1,j)) / DX2 +
                    L(v,i,j+1) - 2.0*L(v,i,j) + L(v,i,j-1)) / DY2);
}
void grid2D_calcPoissonSourceTerm(grid2D *g) 
{
array2D *uT  = &(g->uTant);
array2D *vT  = &(g->vTant);
array2D *d   = &(g->d);
int row_m_1 = array2D_getRow(uT) - 1;
int col_m_1 = array2D_getCol(uT) - 1;
int i, j;
#pragma omp parallel for private(i)
for(j=1; j<col_m_1; j++)
for(i=1; i<row_m_1; i++) {
*(at(d,i,j)) = ((L(uT,i+1,j)  -L(uT,i-1,j)) /DX /2.0 +
   (L(vT,i,j+1) - L(vT,i,j-1)) /DY /2.0) / DT;
}
}

void grid2D_calcPoisson_Jacobi(grid2D *g, double target_residual_rate) 
{
int i,j,k=0;
register double const1 = DX2*DY2/2/(DX2+DY2);
register double const2 = 1.0/DX2;
register double const3 = 1.0/DY2;
double residual = 0.0;
double residualMax = 0.0;
double residualMax_1st = 0.0;
array2D *phi = &(g->phi);
array2D *phiT = &(g->phiTemp);
array2D *d   = &(g->d);
int row_m_1 = array2D_getRow(phi) - 1;
int col_m_1 = array2D_getCol(phi) - 1;

   // Jacobi iteration until residual becomes a target value
   do { // while()
     // loop to set phiTemp by computing with phi
     #pragma omp parallel for private(i)
     for(j=1; j<col_m_1; j++)
     for(i=2; i<row_m_1 - 1; i++) {
       *(at(phiT,i,j)) = const1 * ( (L(phi,i+1,j) + L(phi,i-1,j)) * const2 +
                   (L(phiT,i ,j+1)  + L(phiT,i ,j-1)) * const3 - L(d,i,j));
     }
     k++;
     grid2D_calcBoundary_SqObject(g, OBJ_X, OBJ_Y, OBJ_W, OBJ_H);
     residualMax = 0.0; // Calculating residual
     for(j=2; j<col_m_1 - 1; j++)
     for(i=2; i<row_m_1 - 1; i++) {
       if(residualMax < fabs(L(phi,i,j)-L(phiT,i,j))))
         residualMax = residual;
     }
     if (k == 1) residualMax_1st = residualMax;
     //printf("k=%d : max residual=\%3.6f\n", k, residualMax);
   } while(residualMax > residualMax_1st / target_residual_rate);
   // iterate until residual becomes less than X% of the 1st one
   if((tstep%JACOBIREP_INTERVAL) == 0) printf("Time-step=%d : %3d iterations in Jacobi loop\n", tstep, k);
}
void grid2D_calcVelocity(grid2D *g)
{
    array2D *u = &(g->u);
    array2D *v = &(g->v);
    array2D *uT = &(g->uTant);
    array2D *vT = &(g->vTant);
    array2D *phi = &(g->phi);
    int row_m_1 = array2D_getRow(u) - 1;
    int col_m_1 = array2D_getCol(u) - 1;
    int i, j;

    #pragma omp parallel for private(i)
    for(j=1; j<col_m_1; j++)
    for(i=1; i<row_m_1; i++)
    {
        *(at(u,i,j)) = L(uT,i,j) - DT/2/DX*(L(phi,i+1,j) - L(phi,i-1,j));
        *(at(v,i,j)) = L(vT,i,j) - DT/2/DY*(L(phi,i,j+1) - L(phi,i,j-1));
    }
}

// Boundary conditions of outer cells for Poiseulle flow
void grid2D_calcBoundary_Poiseulle(grid2D *g, double phi_in, double phi_out)
{
    int i, i1, i2, j, j1, j2;
    array2D *u = &(g->u);
    array2D *v = &(g->v);
    array2D *phi = &(g->phi);
    int row = array2D_getRow(u);
    int col = array2D_getCol(u);
    j1 = 1; // bottom
    j2 = col-2; // top

    #pragma omp parallel for
    for(i=0; i<row; i++)
    {
        *(at(u,i,j1)) = 0.0;
        *(at(v,i,j1)) = 0.0;
        *(at(phi,i,j1)) = phi_in;
        *(at(phi,i,j1-1)) = L(phi,i,j1);
        *(at(u,i,j2)) = 0.0;
        *(at(v,i,j2)) = 0.0;
        *(at(phi,i,j2)) = L(phi,i,j2-1);
        *(at(phi,i,j2+1)) = L(phi,i,j2-1) - ((2.0*NU/DY)*L(v,i,j2-1));
    }
    // Pressure condition
    *(at(u,i2+1,j)) = L(u,i2-1,j);
    *(at(v,i2+1,j)) = L(v,i2-1,j);
    *(at(phi,i2+1,j)) = phi_out;
    *(at(phi,i2+1,j)) = L(phi,i2-1,j);
    // Pressure condition
    *(at(u,i1-1,j)) = L(u,i1+1,j);
    *(at(v,i1-1,j)) = L(v,i1+1,j);
    *(at(phi,i1-1,j)) = phi_in;
    *(at(phi,i1-1,j)) = L(phi,i1+1,j);
}
void grid2D_calcBoundary_SqObject(grid2D *g, int obj_x, int obj_y, int obj_w, int obj_h)
{
    int i, i1, i2, j, j1, j2;
    int sta_i = (int)(obj_x - obj_w/2); // pos of left surface
    int end_i = (int)(sta_i + obj_w);  // pos of right surface
    int sta_j = (int)(obj_y - obj_h/2); // pos of bottom surface
    int end_j = (int)(sta_j + obj_h);  // pos of top surface

    array2D *u = &(g->u);
    array2D *v = &(g->v);
    array2D *phi = &(g->phi);
    array2D *phiT = &(g->phiTemp);

    i1 = sta_i;  // left surface of the obstacle
    i2 = end_i;  // right surface of the obstacle
    j1 = end_j;  // top surface of the obstacle
    j2 = sta_j;  // bottom surface of the obstacle

    #pragma omp parallel for
    for(j=sta_j; j<=end_j; j++) {
        *at(u,i1,j) = 0.0;
        *at(v,i1,j) = 0.0;
        *at(u,i1+1,j) = L(u,i1-1,j);
        *at(phi,i1,j) = L(phi,i1-1,j) + ((2.0*NU/DX)*L(u,i1-1,j));
        *at(phiT,i1,j) = L(phi,i1,j);
        *at(u,i2,j) = 0.0;
        *at(v,i2,j) = 0.0;
        *at(u,i2-1,j) = L(u,i2+1,j);
        *at(phi,i2,j) = L(phi,i2+1,j) + ((2.0*NU/DX)*L(u,i2+1,j));
        *at(phiT,i2,j) = L(phi,i2,j);
    }

    #pragma omp parallel for
    for(i=sta_i+1;i<end_i;i++) {
        *at(u,i,j1) = 0.0;
        *at(v,i,j1) = 0.0;
        *at(v,i,j1-1) = L(v,i,j1+1);
        *at(phi,i,j1) = L(phi,i,j1+1) + ((2.0*NU/DY)*L(v,i,j1+1));
        *at(phiT,i,j1) = L(phi,i,j1);
        *at(u,i,j2) = 0.0;
        *at(v,i,j2) = 0.0;
        *at(v,i,j2+1) = L(v,i,j2-1);
        *at(phi,i,j2) = L(phi,i,j2-1) + ((2.0*NU/DY)*L(v,i,j2-1));
        *at(phiT,i,j2) = L(phi,i,j2);
    }
}

Hands-on : Non (MPI)-parallelized CFD simulation

Note that each loop is already parallelized by using OpenMP. See “#pragma omp parallel for private(i)” of each loop. Compare execution time with or without OpenMP description.
Compile and Execute interactively

```bash
Klogin5$ make
========================================================================
Compilation starts for solver_fractional.
========================================================================
FCCpx -Kfast,openmp -l./ -o main.o -c main.cpp
FCCpx -Kfast,openmp -l./ -o cfd.o -c cfd.cpp
FCCpx -o solver_fractional main.o cfd.o -Kfast,openmp -L./ -Im

klogin5$ pjsub --interact --rsc-list "node=1" --rsc-list "elapse=00:20:00" --sparam "wait-time=30"
[INFO] PJM 0000 pjsub Job 7399283 submitted.
[INFO] PJM 0081 .connected.
[INFO] PJM 0082 pjsub Interactive job 7399283 started.
[a03574@k05-036 fracStep_KarmanVotex_20180607_mod1]$ source ./scripts/setenv_interact_1mpi.sh
Env_base: K-1.2.0-24
[a03574@k05-036 fracStep_KarmanVotex_20180607_mod1]$ ./solver_fractional
Time-step=0 : 1122 iterations in Jacobi loop
Writing to AVEse_000200.dat
Time-step=200 : 76 iterations in Jacobi loop
Writing to AVEse_000400.dat
Time-step=400 : 44 iterations in Jacobi loop
...
[a03574@k05-036 fracStep_KarmanVotex_20180607_mod1]$ exit
```

Visualize AVEse_00****.dat

- Copy Visualization script to "python" dir in your home
  ```bash
  cp /scratch/ra001016/viewer_multi.sh ~/python/
  ```

- Move to a directory which contains *.dat files
  ```bash
  > ls
  > AVEse_000200.dat  AVEse_000400.dat  AVEse_000800.dat
  > AVEse_001000.png  AVEse_001200.png  AVEse_001400.png
  ```

- Execute script to convert *.dat files to image files
  ```bash
  > viewer_multi.sh
  > Visualizing AVEse_000200.dat
  ```

- View the image files
  ```bash
  > animate *.png
  ✓ (Note that X-server needs to be available.)
  ✓ Mouse-right click -> Menu -> Speed -> You can change animation speed.
  ```
PART-II
Parallelization of the 2D CFD Simulation

Overview

- **Scaling performance beyond a single node**
  - ✓ Parallelization with a distributed-memory nodes requires message passing
  - ✓ One of the approaches to partition the entire computation is “Domain Decomposition”

- **Domain decomposition**
  - ✓ Decompose the computational grid to create sub-computation
  - ✓ Data communication and synchronization are made when necessary.
Decompose the entire grid into subgrids
✓ Perform stencil computation with each subgrid in parallel
✓ Exchange boundary data when necessary

Halo: Overlapped boundary region
✓ Halo data are exchanged all at once in advance to the loop, so that no communication occurs during the loop.
Parallelization Overview

Serial program

- $\text{calcTantVelocity}$
- $\text{calcPoissonSourceTerm}$
- $\text{calcPoisson_Jacobi}$
- $\text{calcPoisson_Jacobi (repeated)}$
- $\text{calcVelocity}$

MPI-parallel program

- $\text{Halo exchg. of u, v}$
- $\text{calcTantVelocity}$
- $\text{Halo exchg. of uTant, vTant}$
- $\text{calcPoissonSourceTerm}$
- $\text{Halo exchg. of phi}$
- $\text{calcPoisson_Jacobi}$
- $\text{Halo exchg. of phiTemp}$
- $\text{calcPoisson_Jacobi (repeated)}$
- $\text{Halo exchg. of uTant, vTant, phi}$
- $\text{calcVelocity}$

Let's Read the parallelized code!

```
klogin5$ cd programs
klogin5$ cp /scratch/ra001016/day2/parallel_incomplete_0703.tgz ./
klogin5$ tar zxvf parallel_incomplete_0703.tgz
klogin5$ cd parallel_incomplete_0703/
```

New files.
Codes for subgrid management.

```
klogin5$ ls
cfd.c
  cfd.h
  domain_decomp.c
domain_decomp.h
main.cpp
main.h
Makefile
README.txt
scripts
```

MPI parallelization is introduced.
typedef struct array2D_ {
    int nx;            // NX resolution of a grid
    int ny;            // NY resolution of a grid
    double *v;        // Pointer of 2D array
    double *l_send, *r_send, *l_recv, *r_recv; // Buffer for communicate
} array2D;

// Member functions for array2D
void array2D_initialze(array2D *a, int nx, int ny);       // initialize 2D array : nx x ny
void array2D_resize(array2D *a, int nx, int ny);          // resize 2D array : nx x ny
void array2D_copy(array2D *src, array2D *dst);            // copy src to dst (by resizing dst)
void array2D_clear(array2D *a, double v);                 // clear 2D array with value of v
void array2D_show(array2D *a);                            // print 2D array in text
double linear_intp(array2D *a, double x, double y);       // get value ay (x,y) with linear interpolation
inline int array2D_getNx(array2D *a) { return (a->nx); }  // get size of nx
inline int array2D_getNy(array2D *a) { return (a->ny); }  // get size of ny

inline double *at(array2D *a, int i, int j)             // get pointer at (nx, ny)
{
    if ((i<0-HALO) || (j<0-HALO) || (i>=a->nx+HALO) || (j>=a->ny+HALO)) {
        printf("Out of range: (%d, %d) for %dx%d array in at(). Abort.\n", i, j, a->nx, a->ny);
        exit(EXIT_FAILURE);
    }
    return (a->v + i+ j * (a->nx+2*HALO));
}

typedef struct grid2D_ {
    array2D u, v, phi;        // velocity (u, v), pressure phi
    array2D phiTemp;          // tentative pressure (temporary for update)
    array2D uTant, vTant;     // tentative velocity (u, v)
    array2D d;                // source term of a pressure poisson's equation
} grid2D;

// Member functions for grid2D
void grid2D_initialze(grid2D *g, int nx, int ny, double phi_in, double phi_out, const info_domain mpd);
void grid2D_calcTantVelocity(grid2D *g, const info_domain mpd);
void grid2D_calcPoissonSourceTerm(grid2D *g, const info_domain mpd);
void grid2D_calcPoisson_Jacobi(grid2D *g, double target_residual_rate, const info_domain mpd);
void grid2D_calcVelocity(grid2D *g, const info_domain mpd);
void grid2D_calcBoundary_Poiseulle(grid2D *g, double phi_in, double phi_out, const info_domain mpd);
void grid2D_calcBoundary_SqObject(grid2D *g, int obj_x, int obj_y, int obj_w, int obj_h, const info_domain mpd);
void communicate_neighbor(array2D *a, const info_domain mpd);
void communicate_neighbor_debug(array2D *a, const info_domain mpd);
inline int grid2D_getNx(grid2D *g) { return( array2D_getNx(&g->u) ); }
inline int grid2D_getNy(grid2D *g) { return( array2D_getNy(&g->u) ); }
#ifndef ____DOMAIN_DECOMP_H____
#define ____DOMAIN_DECOMP_H____

#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <stdio.h>

#define MCW MPI_COMM_WORLD
#define HALO (1)

//Data structure for mpi
typedef struct info_domain_ {
  int dims[2]; //Dimension
  int coord[2]; //Coord of me_proc
  int east, west, north, south; //Neighbor procs ID
  int nx, ny, gnx, gny; // (gnx, gny) : resolution of entire grid, (nx, ny) : resolution of each subgrid
  int sx, ex, sy, ey; // start_x, end_x, start_y, end_y
} info_domain;

void info_domain_initialize(info_domain *mpd, const int num_procs, const int me_proc);
void calc_range(info_domain *mpd, const int nx, const int ny);

#endif
New file : domain_decomp.c

```c
void info_domain_initialize(info_domain *mpd, const int num_procs, const int me_proc)
{
    mpd->dims[0] = sqrt(num_procs / 3);
    mpd->dims[1] = num_procs / mpd->dims[0];
    if(mpd->dims[0] * mpd->dims[1] != num_procs)
    {
        if(me_proc == 0) {
            printf("Number of processes is invalid. Please choose the valid condition.\n");
            printf("Number of processes must be 3n^2. ("n" is arbitrary value.) \n");
        }
        MPI_Abort(MCW, -1);
    }
    mpd->coord[1] = me_proc % mpd->dims[1];
    mpd->coord[0] = me_proc / mpd->dims[1];
    mpd->east = mpd->coord[1]<mpd->dims[1]-1 ? me_proc+1 : MPI_PROC_NULL;
    mpd->west = mpd->coord[1]>0 ? me_proc-1 : MPI_PROC_NULL;
    mpd->north = mpd->coord[0]<mpd->dims[0]-1 ? me_proc+mpd->dims[1] : MPI_PROC_NULL;
}

void calc_range(info_domain*mpd, const int nx, const int ny)
{
    mpd->gnx= nx;
    mpd->gny= ny;
    mpd->nx = nx / mpd->dims[1];
    mpd->ny = ny / mpd->dims[0];
    sx= 0; if (mpd.west== MPI_PROC_NULL) sx= 1;
    ex = array2D_getNx(u); if (mpd.east== MPI_PROC_NULL) ex = ex-1;
    sy= 0; if (mpd.south== MPI_PROC_NULL) sy= 1;
    ey= array2D_getNy(u); if (mpd.north== MPI_PROC_NULL) ey= ey-1;
}
```

grid2D_calcTantVelocity()

```c
void grid2D_calcTantVelocity(grid2D *g, const info_domain mpd)
{
    array2D *u = &(g->u);
    array2D *v = &(g->v);
    array2D *uT = &(g->uTant);
    array2D *vT = &(g->vTant);
    int i, j, sx, ex, sy, ey;

    sx = 0; if (mpd.west == MPI_PROC_NULL) sx = 1;
    ex = array2D_getNx(u); if (mpd.east == MPI_PROC_NULL) ex = ex - 1;
    sy = 0; if (mpd.south == MPI_PROC_NULL) sy = 1;
    ey = array2D_getNy(u); if (mpd.north == MPI_PROC_NULL) ey = ey - 1;

    #pragma omp parallel for private(i)
    for(j=sy; j<ey; j++)
        for(i=sx; i<ex; i++) {
            *(at(uT,i,j)) =
                L(u,i,j) + DT*(-L(u,i,j)*(L(u,i+1,j ) - L(u,i-1,j )) / 2.0 / DX
                -L(v,i,j)*(L(v,i ,j+1) - L(v,i ,j-1)) / 2.0 / DY +
                NU*( (L(u,i+1,j ) - 2.0*L(u,i,j ) + L(u,i-1,j )) / DX2 +
                  (L(v,i ,j+1) - 2.0*L(v,i,j ) + L(v,i ,j-1)) / DY2 ) ;

            *(at(vT,i,j)) =
                L(v,i,j) + DT*(-L(u,i,j)*(L(v,i+1,j ) - L(v,i-1,j )) / 2.0 / DX
                -L(v,i,j)*(L(v,i ,j+1) - L(v,i ,j-1)) / 2.0 / DY +
                NU*( (L(v,i+1,j ) - 2.0*L(v,i,j ) + L(v,i-1,j )) / DX2 +
                  (L(v,i ,j+1) - 2.0*L(v,i,j ) + L(v,i ,j-1)) / DY2 ) ;
        }
    communicate_neighbor(uT, mpd);
    communicate_neighbor(vT, mpd);
}
```
communicate_neighbor() for Halo Exchange

Exchange Halo of Array u in Grid g by communicating data with adjacent subgrids.
Usage: communicate_neighbor(&g->u, mpd);

```c
void communicate_neighbor(array2D *a, const info_domain mpd)
{
    int x, y, nx, ny;
    MPI_Status st;

    nx = array2D_getNx(a);
    ny = array2D_getNy(a);

    //Please write communication routines between own and neighbor processes.
}
```

Hint:
Row Halo (top and bottom) are continuously arranged in a memory while column Halo (left and right) are NOT. Since MPI_sendrecv() requires continuity for transferred data, you need to copy non-continuous data into some buffer before executing MPI_sendrecv() so that the copied data are continuous in the buffer.

You can use array2D's `double *l_send, *r_send, *l_recv, *r_recv;` as buffers for Halo communication. Memory regions are allocated in array2D_resize().

---

How to Implement Halo Exchange?

To obtain the top Halo of mine with south subgrid,

The row of (nx+2*HALO)*HALO cells starting at (-HALO, ny-Halo) should be sent to the bottom Halo of the south at (-HALO, 0).
  * The coordinate of origin in the subgrid is (0, 0)

The top Halo of mine starting at (-HALO, ny) should be received from the row of the south starting at (-HALO, -HALO).

Notice:
Think carefully about source and destination processes.

Sendrecv(......, north, ......, north, ...)
← Is this right?
Deadlock occurs?
Compile and Execute interactively

```
Klogin5$ make
========================================================================
= Compilation starts for solver_fractional.
========================================================================
FCCpx -Kfast,openmp -I./ -o main.o -c main.cpp
FCCpx -Kfast,openmp -I./ -o cfd.o -c cfd.cpp
FCCpx -o solver_fractional main.o cfd.o -Kfast,openmp -L./ -Im

Klogin5$ pjsub --interact --rsc-list "node=3" --rsc-list "elapse=00:20:00" // Reserve nodes for "3" MPI procs for interactive use.
[INFO] PJM 0000 pjsub Job 7419075 submitted.
[INFO] PJM 0081 connected.
[INFO] PJM 0082 pjsub Interactive job 7419075 started.

[a03555@w05-036 cfd_ss2018_strong_scaling]$ source /work/system/Env_base
[a03555@w05-036 cfd_ss2018_strong_scaling]$ export PARALLEL=3
[a03555@w05-036 cfd_ss2018_strong_scaling]$ export OMP_NUM_THREADS=8
Env_base: K-1.2.0-24

[a03555@w05-036 cfd_ss2018_strong_scaling]$ mpiexec ./solver_fractional
Writing to AVEse_000000.dat
Time-step=0 : 440 iterations in Jacobi loop
Time-step=200 : 38 iterations in Jacobi loop
Time-step=400 : 40 iterations in Jacobi loop
Time-step=600 : 40 iterations in Jacobi loop
...
// In the interactive mode, the num of maximum MPI procs is 384.
```
Execute parallel computation by Batch

```
execute parallel computation by Batch
klogin9$ make
========================================================================
|= Compilation starts for solver_fractional.
========================================================================
mpifcpx -Kfast,openmp -l./ -o main.o -c main.c
mpifcpx -Kfast,openmp -l./ -o cfd.o -c cfd.c
PLEASE IGNORE WARNINGS
mpifcpx -Kfast,openmp -l./ -o domain_decomp.o -c domain_decomp.c
mpifcpx -o solver_fractional main.o cfd.o domain_decomp.o -L./ -lm -Kfast,openmp
```

```
klogin5$ pjsub ./scripts/run_batch_3mpi.sh
klogin9$ pjstat
ACCEPT QUEUED STGIN READY RUNNING RUNOUT STGOUT HOLD ERROR TOTAL
 0 0 1 0 0 0 0 0 1
s 0 0 1 0 0 0 0 0 1
JOB_ID JOB_NAME         MD ST USER GROUP      START_DATE ELAPSE_TIME NODE_REQUIRE RSC_GRP SHORT_RES
7419161 run_batch_3mpi.s NM SIN a03574 ra001016 [07/02 16:42:00] 0000:00:00 3:- small -

( klogin9$ pjdel <JOB_ID> ) to kill a queued MPI process
```

when MPI parallel execution finishes, there files containing standard-outputs and generated *.dat files.
...

Measure Exec Time w/o Saving Files

When you measure the elapsed time by excluding file-writing time, please
un-comment 35th line and
comment out 34th line in Makefile.

```
#CFLAGS  = -Kfast,openmp -DMEASURE_TIME $(INCLUDE_DIR)
CFLAGS  = -Kfast,openmp $(INCLUDE_DIR)
↓
CFLAGS  = -Kfast,openmp -DMEASURE_TIME $(INCLUDE_DIR)
#CFLAGS  = -Kfast,openmp $(INCLUDE_DIR)
```
Observe Speedup by Changing num_proc

### Strong scaling
- Parallel computation with 3n^2 MPI processes for the same grid size
- Job scripts in ./scripts/run_batch_XXmpi.sh for parallel run
- Fill out the table as below
  (No need to change the parameters in cfd.h)
- Draw the graph: # of MPI processes vs. Elapsed time

<table>
<thead>
<tr>
<th>n procs</th>
<th>Time [sec]</th>
<th>Speedup (ideal)</th>
<th>grid points per proc</th>
<th>nx=ny</th>
<th>gnx</th>
<th>gny</th>
<th>DT</th>
<th>END_TIMESTEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>?</td>
<td>1</td>
<td>36864</td>
<td>192</td>
<td>576</td>
<td>192</td>
<td>0.00003</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>?</td>
<td>4</td>
<td>9216</td>
<td>96</td>
<td>576</td>
<td>192</td>
<td>0.00003</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>?</td>
<td>9</td>
<td>4096</td>
<td>64</td>
<td>576</td>
<td>192</td>
<td>0.00003</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>?</td>
<td>16</td>
<td>2304</td>
<td>48</td>
<td>576</td>
<td>192</td>
<td>0.00003</td>
</tr>
<tr>
<td>6</td>
<td>108</td>
<td>?</td>
<td>25</td>
<td>1024</td>
<td>32</td>
<td>576</td>
<td>192</td>
<td>0.00003</td>
</tr>
<tr>
<td>8</td>
<td>192</td>
<td>?</td>
<td>36</td>
<td>576</td>
<td>24</td>
<td>576</td>
<td>192</td>
<td>0.00003</td>
</tr>
<tr>
<td>12</td>
<td>432</td>
<td>?</td>
<td>49</td>
<td>256</td>
<td>16</td>
<td>576</td>
<td>192</td>
<td>0.00003</td>
</tr>
<tr>
<td>16</td>
<td>768</td>
<td>?</td>
<td>64</td>
<td>144</td>
<td>12</td>
<td>576</td>
<td>192</td>
<td>0.00003</td>
</tr>
<tr>
<td>1728</td>
<td>?</td>
<td>81</td>
<td></td>
<td>64</td>
<td>8</td>
<td>576</td>
<td>192</td>
<td>0.00003</td>
</tr>
</tbody>
</table>

### Weak scaling
- Parallel computation with the same grid size per MPI process
- Change the parameters in cfd.h in the list, and then run the job script in ./scripts/run_batch_XXmpi.sh
  - NX, NY, DT, END_TIMESTEP
- Fill out the table as below, and draw the graph (# of MPI process)

<table>
<thead>
<tr>
<th>n procs</th>
<th>Time [sec]</th>
<th>Speedup (ideal)</th>
<th>grid points per proc</th>
<th>nx=ny</th>
<th>gnx</th>
<th>gny</th>
<th>DT</th>
<th>END_TIMESTEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>?</td>
<td>1</td>
<td>10800</td>
<td>3600</td>
<td>60</td>
<td>0.000100</td>
<td>2000</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>?</td>
<td>4</td>
<td>43200</td>
<td>3600</td>
<td>120</td>
<td>0.000050</td>
<td>4000</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>?</td>
<td>9</td>
<td>172800</td>
<td>3600</td>
<td>240</td>
<td>0.000025</td>
<td>8000</td>
</tr>
<tr>
<td>6</td>
<td>108</td>
<td>?</td>
<td>16</td>
<td>388800</td>
<td>3600</td>
<td>360</td>
<td>0.000017</td>
<td>12000</td>
</tr>
<tr>
<td>8</td>
<td>192</td>
<td>?</td>
<td>25</td>
<td>691200</td>
<td>3600</td>
<td>480</td>
<td>0.000013</td>
<td>16000</td>
</tr>
<tr>
<td>10</td>
<td>300</td>
<td>?</td>
<td>36</td>
<td>1080000</td>
<td>3600</td>
<td>600</td>
<td>0.000010</td>
<td>20000</td>
</tr>
<tr>
<td>14</td>
<td>588</td>
<td>?</td>
<td>49</td>
<td>2116800</td>
<td>3600</td>
<td>840</td>
<td>0.000007</td>
<td>28000</td>
</tr>
<tr>
<td>17</td>
<td>867</td>
<td>?</td>
<td>64</td>
<td>3121200</td>
<td>3600</td>
<td>1020</td>
<td>0.000006</td>
<td>34000</td>
</tr>
<tr>
<td>18</td>
<td>972</td>
<td>?</td>
<td>81</td>
<td>3499200</td>
<td>3600</td>
<td>1080</td>
<td>0.000006</td>
<td>36000</td>
</tr>
</tbody>
</table>
Strong scaling (gnx=576, gny=192)

![Strong scaling graph](image)

Weak scaling (60x60 per each proc)

![Weak scaling graph](image)