HPC Summer School
Computational Fluid Dynamics Simulation and its Parallelization

Kentaro Sano

Processor Research Team, R-CCS Riken

3 July, 2018

Agenda

- **PART-I**

  Introduction of Application: 2D CFD Simulation

  ✓ Lecture

  ✓ Hands-on Practice

- **PART-II**

  Parallelization of the 2D CFD Simulation

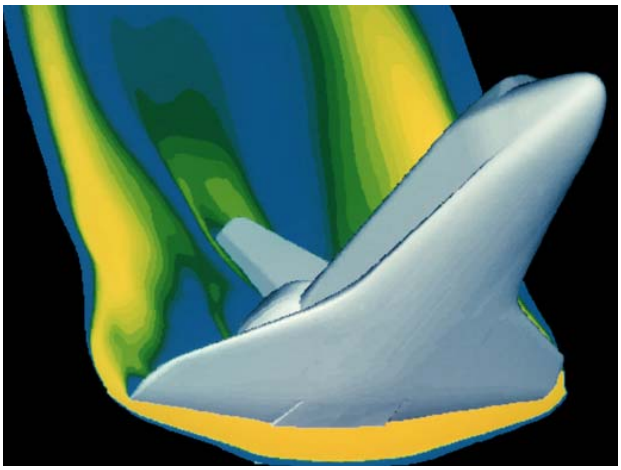
  ✓ Lecture

  ✓ Hands-on Practice

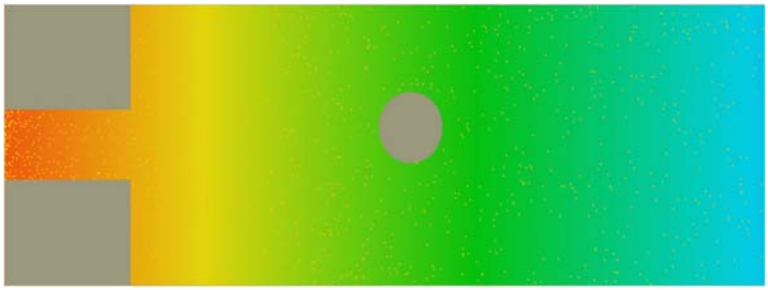
PART-I
Introduction of Application:
2D CFD Simulation

Introduction

- **What's CFD** (Computational Fluid Dynamics) **simulation ?**

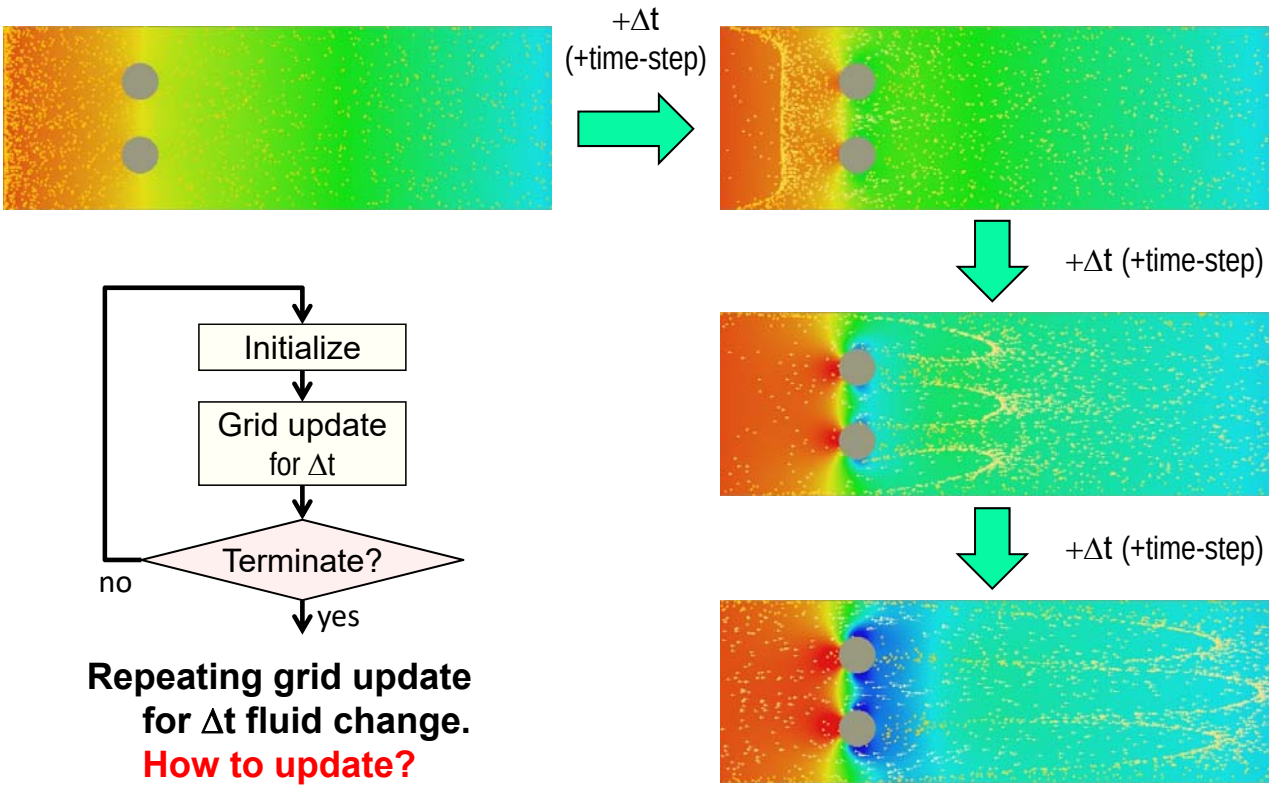


A computer simulation of high velocity air flow around the Space Shuttle during re-entry.



2D CFD example

How to Compute Fluid Flow?



Repeating grid update for Δt fluid change.
How to update?

Incompressive Viscous Fluid Flow

Governing Equations with partial differential equations

$$\nabla V = 0$$

Equation of continuity
(incompressive flow)

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

Navier–Stokes equations
(incompressive flow)

V	velocity = (u, v)	$\nu \equiv \mu / \rho$	kinematic viscosity
P	pressure	$\varphi \equiv P / \rho$	
ρ	density		

Fractional-Step Method

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

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

2. Calculate **the pressure field** ϕ^{n+1} **of the next time-step with** V^* **by solving the Poisson's equation.**

$$\nabla^2 \varphi^{n+1} = \frac{\nabla \cdot V^*}{\Delta t} \quad (2)$$

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

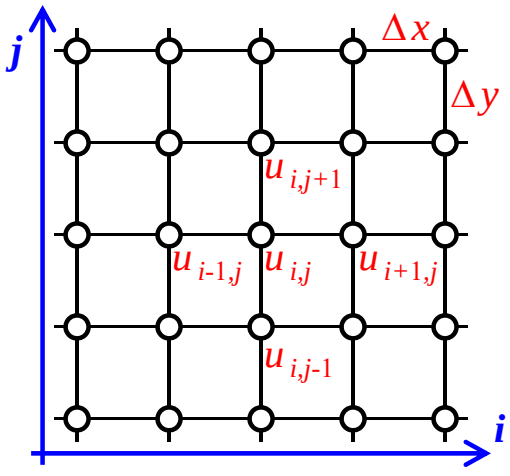
$$V^{n+1} = V^* - \Delta t \nabla \varphi^{n+1} \quad (3)$$

Finite Difference Schemes

We can make discrete forms by substituting difference schemes.

$$\frac{du}{dx} \simeq \frac{u_{i+1} - u_{i-1}}{2\Delta x}$$

$$\frac{d^2u}{dx^2} \simeq \frac{u_{i-1} - 2u_i + u_{i+1}}{(\Delta x)^2}$$



Central difference schemes
(=> Finite difference scheme)

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

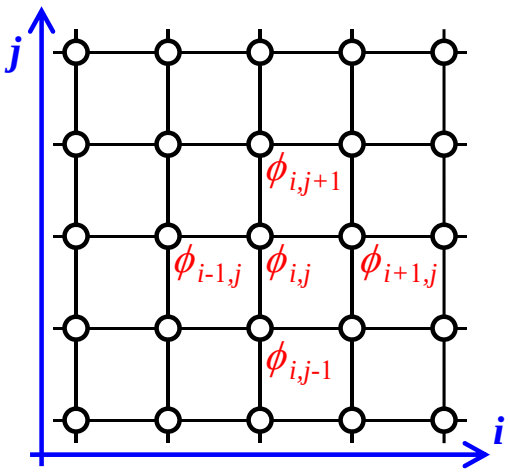
See "staggered mesh" for more advanced study.

Discrete Form of Step1

Step1 : Calculate the tentative velocity : *u*, v**

$$u_{i,j}^{*} = u_{i,j} + \Delta t \left\{ \begin{array}{c} -u_{i,j}\dfrac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} - v_{i,j}\dfrac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} + \\ NU\left(\dfrac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} - \dfrac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2}\right) \end{array} \right\}$$

NU is kinematic viscosity.
A similar equation for v.



Discrete Form of Step2

Step2 : Calculate the pressure by Jacobi method.

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

$$\varphi'_{i,j} = \alpha\left(\frac{\varphi_{i+1,j} + \varphi_{i-1,j}}{(\Delta x)^2} + \frac{\varphi_{i,j+1} + \varphi_{i,j+1}}{(\Delta y)^2} - D_{i,j}\right)$$

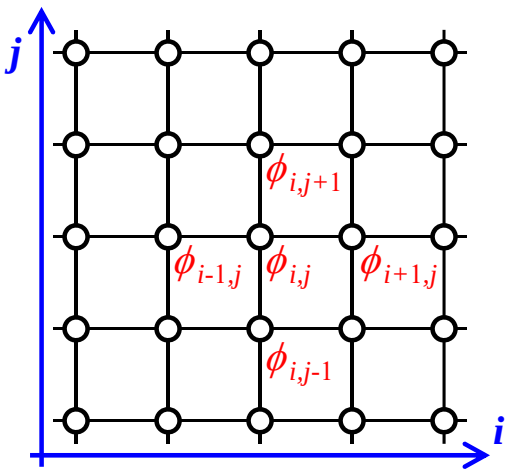
where

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

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.

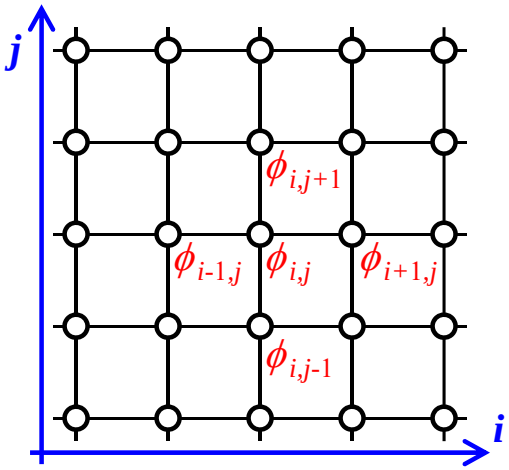


Discrete Form of Step3

Step3 : Calculate the true velocity of the next time-step

$$u_{i,j}^{true} = u_{i,j}^{*} - \Delta t \frac{(\varphi'_{i+1,j} - \varphi'_{i-1,j})}{2\Delta x}$$

$$v_{i,j}^{true} = v_{i,j}^{*} - \Delta t \frac{(\varphi'_{i,j+1} - \varphi'_{i,j-1})}{2\Delta y}$$

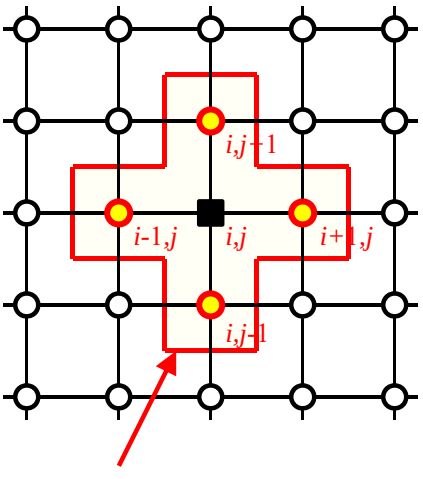


Stencil Computation

Common form in Steps 1, 2, and 3

$$q_{i,j}^{new} = A + Bq_{i,j} + Cq_{i+1,j} + Dq_{i-1,j} + Eq_{i,j+1} + Fq_{i,j-1}$$

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 zxvfp serial_0703.tgz
klogin5$ cd serial_0703/
klogin5$ ls
cfd.cpp
cfd.h
main.cpp
main.h
stopwatch2.h
Makefile
README.txt
scripts
```

Copy tgz archive of source files

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

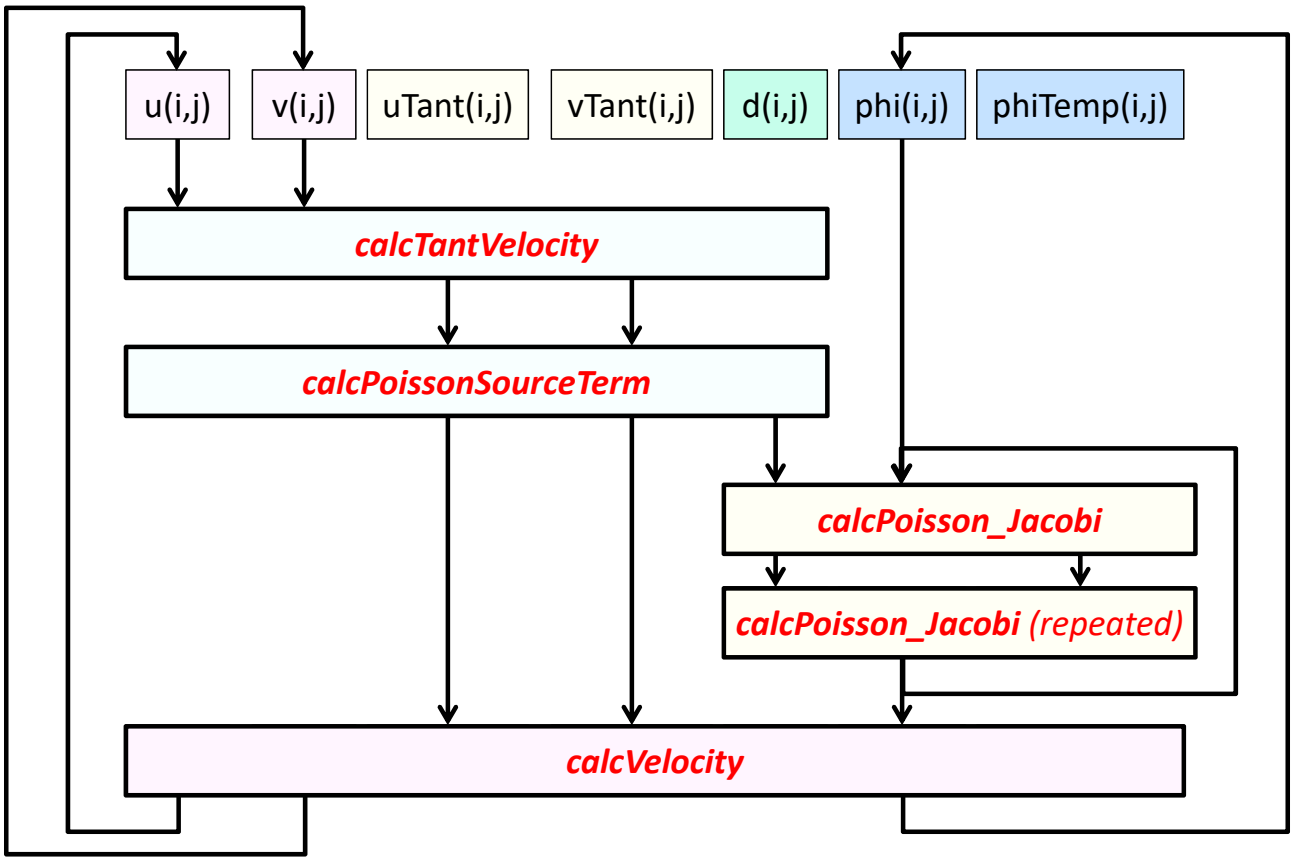
Program Structure

- **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



main.{h, cpp}

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-
 *
 */
#ifndef ___MAIN_H___
#define ___MAIN_H___

#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);

#endif
```

main.cpp

```
#include "main.h"

int main(int argc,char** argv)
{
 grid2D g;
 std::StopWatchClass2 time;
 time.start();
 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

cfd.h

```
#ifndef ___CFD_H___
#define ___CFD_H___

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

// You can change these MACROs (parameters).
//=========================================================================
//Flow condition 1  (taking long time on a single process)
//#define ROW (540)            // cell resolution for row
//#define COL (180)            // cell resolution for column
//#define DT  (0.00003)        // delta of time (difference between timesteps)
//#define NU  (0.01)           // < 0.01 for Karman vortices after 15000 timesteps

//Flow condition 2  (good condition balanced for resolution and execution time)
#define ROW (360)
#define COL (120)
#define DT  (0.00005)
#define NU  (0.01)           // < 0.01 for Karman vortices after 5000 timesteps

//Flow condition 3  (good condition, very fast execution)
//#define ROW (180)
//#define COL (60)
//#define DT  (0.00005)
//#define NU  (0.01)               // < 0.01 for Karman vortices after 10000 timesteps
```

cfd.h 2 of 4

cfd.h

```
#define TARGET_RESIDUAL_RATE (1.0e-2) // Termination condition: 1.0e-3 recommended rather than 1.0e-1)
#define JACOBIREP_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 (dimentionless 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/(COL-1))
#define DX2   (DX*DX)
#define DY2   (DY*DY)

// Boundary conditions for Poiseulle 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
```


cfd.h 3 of 4

cfd.h

```
// 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 ay (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)               // get pointer at (row, col)
{
#if 0
  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.¥n", i, j, a->row, a->col);
    exit(EXIT_FAILURE);
  }
#endif
  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)
```

cfd.h 4 of 4

cfd.h

```
// 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
```


cfd.cpp 1 of 10

cfd.cpp

```
#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);
      }
    }
}
```

cfd.cpp 2 of 10

cfd.cpp

```
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");
   }
}
```


cfd.cpp 3 of 10

cfd.cpp

```
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;
}
```

cfd.cpp 4 of 10

cfd.cpp

```
// Member functions for grid2D
void grid2D_initialize(grid2D *g, int row, int col, double phi_in, double phi_out)
{
 array2D_initialize(&g->u,      row, col);
 array2D_initialize(&g->v,      row, col);
 array2D_initialize(&g->phi,    row, col);
 //array2D_initialize(&g->phiTemp, row+2, col+2); // for halo?
 array2D_initialize(&g->phiTemp, row, col);
 array2D_initialize(&g->uTant,   row, col);
 array2D_initialize(&g->vTant,   row, col);
 array2D_initialize(&g->d,      row, col);
 array2D_clear    (&g->u,      0.01);
 array2D_clear    (&g->v,      0.01);
 array2D_clear    (&g->phi,    0.0);
 array2D_clear    (&g->phiTemp, 0.0);
 array2D_clear    (&g->uTant,   0.01);
 array2D_clear    (&g->vTant,  -0.01);
 array2D_clear    (&g->d,      0.0);

 // Initialize the pressure field with constant gradient
 array2D *a = &(g->phi);
 double row_minus_one = (double)array2D_getRow(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);
}
```


cfd.cpp 5 of 10

cfd.cpp

```
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_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(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(u,i  ,j+1) - L(u,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(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 ) );
  }
}
```

cfd.cpp 6 of 10

cfd.cpp

```
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;
```


cfd.cpp 7 of 10

cfd.cpp

```
  // 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(phi,i  ,j+1) + L(phi,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 < (residual = 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);

    // loop to set phi by computing with phiTemp
#pragma omp parallel for private(i)
    for(j=1; j<col_m_1; j++)
      for(i=2; i<row_m_1 - 1; i++)
        *(at(phi,i,j)) = const1 * ( (L(phiT,i+1,j  ) + L(phiT,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);

  } 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);
}
```

cfd.cpp 8 of 10

cfd.cpp

```
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) );
    }
}
```


cfd.cpp 9 of 10

cfd.cpp

```
// Boundary conditions of outer cells for Poiseulle flow
void grid2D_calcBoundary_Poiseulle(grid2D *g, double phi_in, double phi_out)
{
  //     j
  // COL-1 A
  //      |  =>
  //      |  => flowing dir
  //      |  =>
  //    0 +----------> i
  //      0        ROW-1
  //
  // phi[i][j] : i for x direction, j for y direction
  // [0:ROW-1], inlet(left) boundary at i==1, outlet(right) boundary at i==(ROW-2)
  // [0:COL-1], top boundary at j==(COL-2), bottom boundary at j==1
  // One-cell boundary (one-cell most outer layer) is dummy cells for boundary condition.

  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(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(phi,i,j1-1)) = 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(phi,i,j2+1)) = L(phi,i,j2);
  }

  i1 = 1; // inlet(left, flow incoming)
  i2 = row-2; // outlet(right, flow outgoing)
#pragma omp parallel for
  for(j=1; j<col-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,j))   = phi_in;
    *(at(phi,i1-1,j)) = L(phi,i1+1,j);
    // 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,j))   = phi_out;
    *(at(phi,i2+1,j)) = L(phi,i2-1,j);
  }
}
```

cfd.cpp 10 of 10

cfd.cpp

```
void grid2D_calcBoundary_SqObject(grid2D *g, int obj_x, int obj_y, int obj_w, int obj_h)
{
 // j
 // A
 // |   +-+
 // |   |#|
 // |   |#|
 // |   +-+
 // |
 // +--------------> i
 //
 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
#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);
 }
```

```
 j1 = end_j;  // top surface of the obstacle
 j2 = sta_j;  // bottom surface of the obstacle
#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

cfd.cpp

```
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./ -lm

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]$
[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**

  ```
  cp /scratch/ra001016/viewer_multi.sh ~/python/
  ```

- **Move to a directory which contains *.dat files**

  ```
  > 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**

  ```
  > viewer_multi.sh
  > Visualizing AVEse_000200.dat
  ...
  ```

- **View the image files**

  ```
  > animate *.png
  ```

  ✓ (Note that X-server needs to be available.)

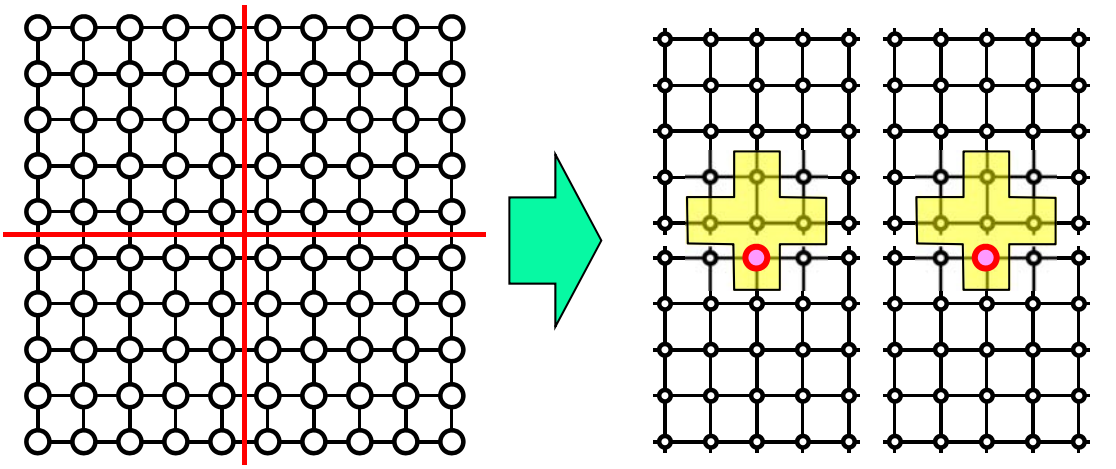
  ✓ Mouse-right click -> Menu -> Speed -> You can change animation speed.

PART-II
Parallelization of the 2D CFD Simulation

Overview

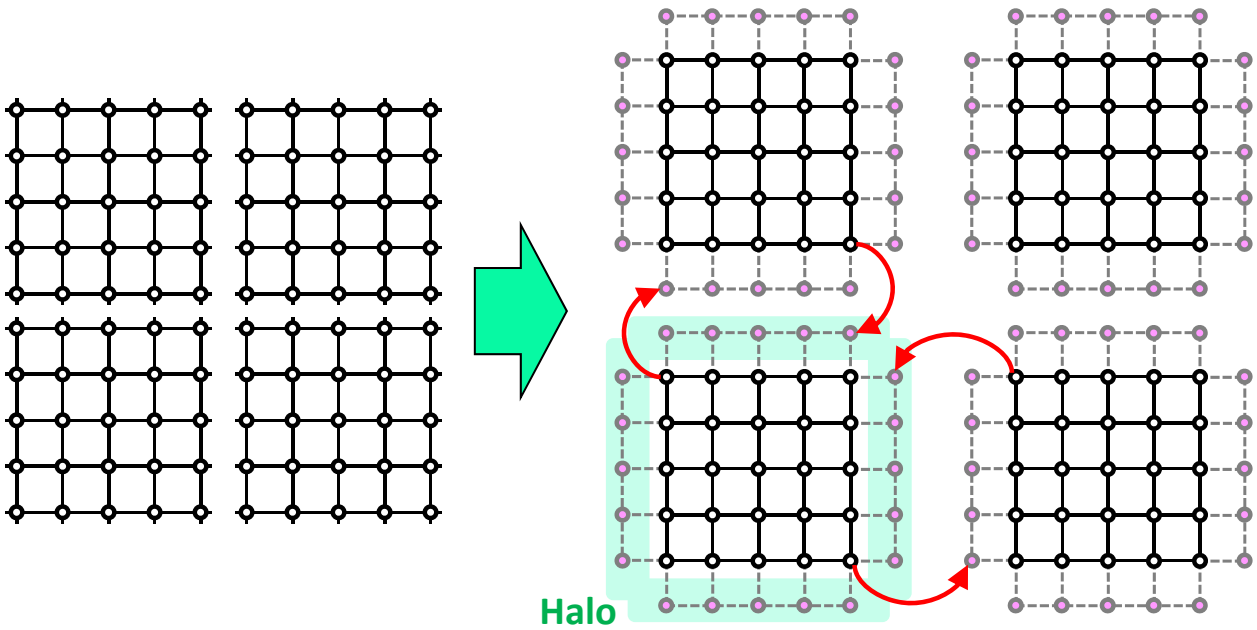
- **Scaling performance beyond a single node**
  - ✓ Parallelization with a distributed-memory nodes requires message passing
  - ✓ One of the approaches to partition the entire computation is “Domain Decomposition”
- **Domain decomposition**
  - ✓ Decompose the computational grid to create sub-computation
  - ✓ Data communication and synchronization are made when necessary.

Parallel Computation w/ Domain Decomposition



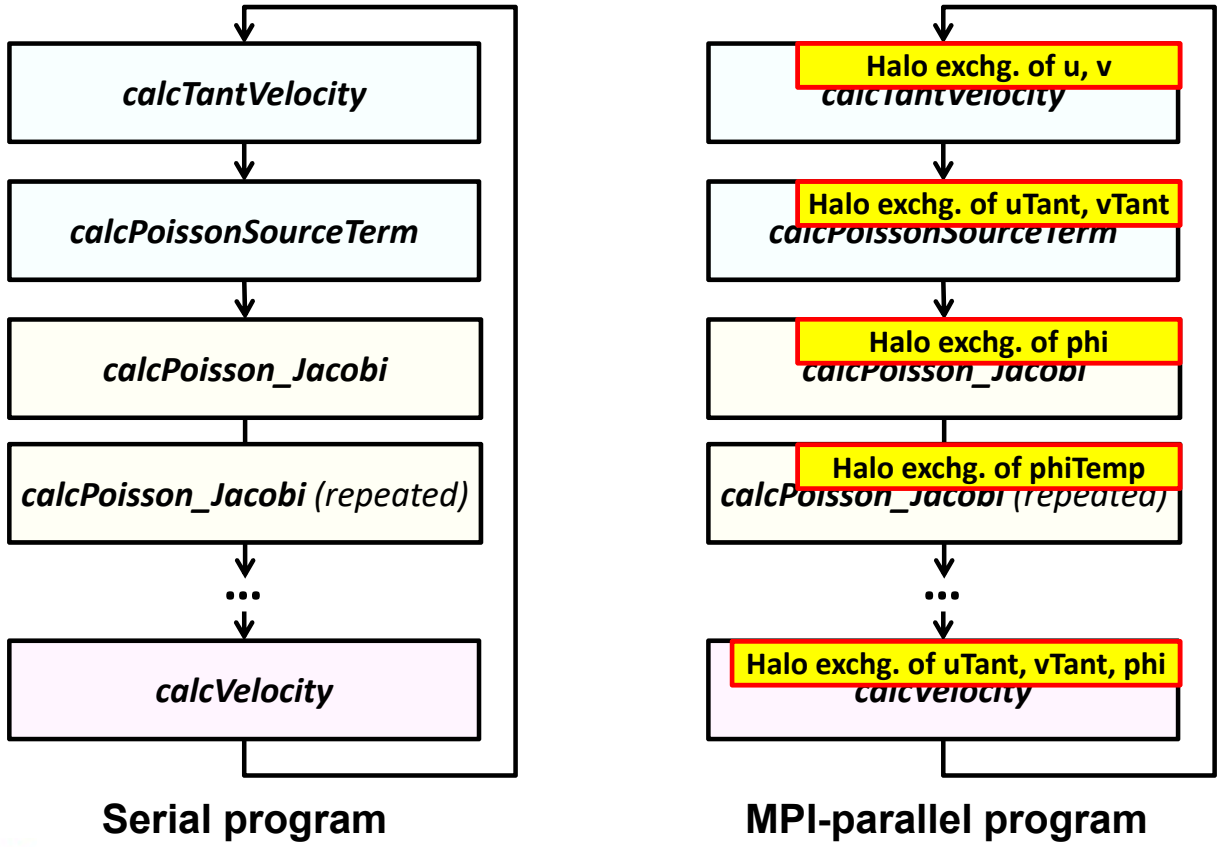
- **Decompose the entire grid into subgrids**
  - ✓ Perform stencil computation with each subgrid in parallel
  - ✓ Exchange boundary data when necessary

Exchanging Halo for Coarse Grain Communication



- **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



Let's Read the parallelized code!

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

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

MPI parallelization is introduced.

New files.
Codes for subgrid management.

cfd.h 1 of 2

cfd.h

```
...
// Data structure of 2D array (resizable)
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_initialize(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 Debug
  if ((i<0-HALO) || (j<0-HALO) || (i>=a->nx+HALO) || (j>=a->ny+HALO)) {
    printf("Out of range : (%d, %d) for %d x %d array in at(). Abort.¥n", i, j, a->nx, a->ny);
    exit(EXIT_FAILURE);
  }
#endif
  return (a->v + i + j * (a->nx+2*HALO));
}
```


cfd.h 2 of 2

cfd.h

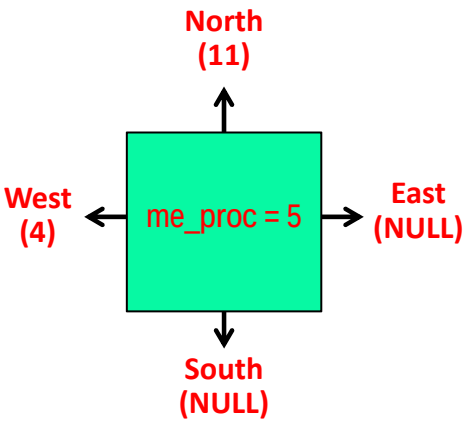
```
...
// Data structure of 2D grid for fluid flow
typedef struct grid2D_ {
  array2D u, v, phi;          // velocity (u, v), pressure phi
  array2D phiTemp;            // tentative pressure (temporary for update)
  array2D uTant, vTant;       // tentative velocity (u, v)
  array2D d;                  // source term of a pressure poisson's equation
} grid2D;
...

// Member functions for grid2D
void grid2D_initialize(grid2D *g, int 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);
void grid2D_outputAVEseFile(grid2D *g, const char *base, int num, double scaling, 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)) ); }
```

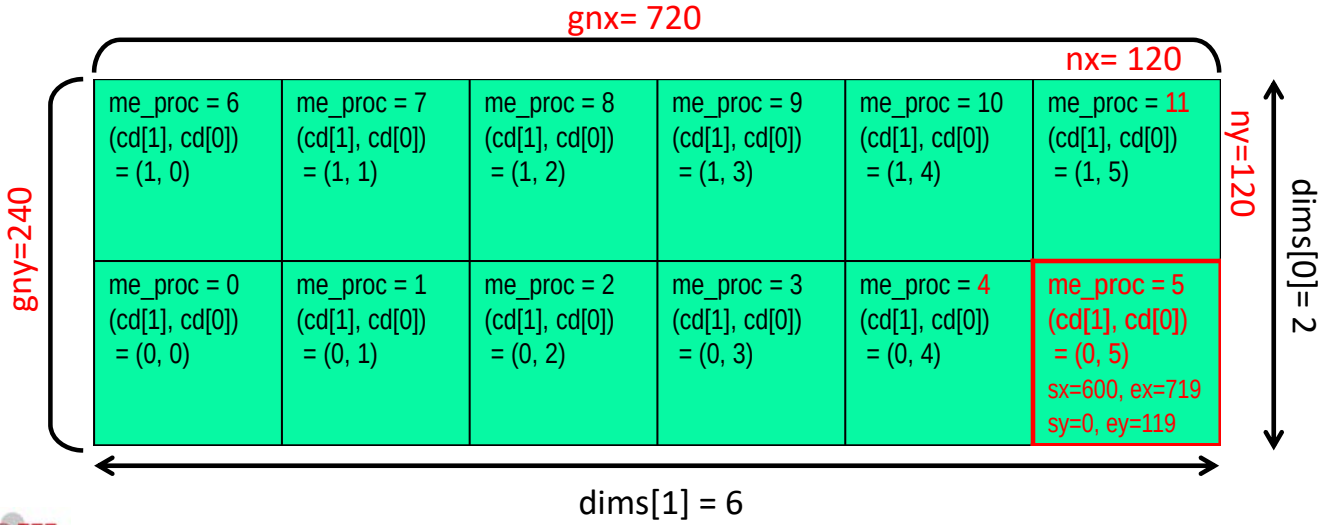
New file : domain_decomp.h

domain_decomp.h

```
#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
```


num_procs = 12 // me_proc is 0 to 11.
dims[0] = sqrt(12/3) = 2 // num of subgrids
dims[1] = 12 / 2 = 6
In the case that me_proc == 5,
mpd->coord[1] = 5 % 6 = 5; // coord of subgrid
mpd->coord[0] = 5 / 6 = 0;
mpd->east = MPI_PROC_NULL // No proc of adjacent subgrid
mpd->west = me_proc - 1 = 4 //proc of adjacent subgrid
mpd->north = me_proc + mpd->dims[1] = 5 + 6 = 11
mpd->south = MPI_PROC_NULL





New file : domain_decomp.c

domain_decomp.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 invalide. Please choose the valid condition.¥n");
      printf("Number of processes must be 3n^2¥n. (""n"" is arbitrary value.) ");
    }
    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;
  mpd->south = mpd->coord[0]>0              ? 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];
  mpd->sx = mpd->nx * mpd->coord[1];
  mpd->ex = mpd->nx * (mpd->coord[1]+1)-1;
  mpd->sy = mpd->ny * mpd->coord[0];
  mpd->ey = mpd->ny * (mpd->coord[0]+1)-1;
}
```


grid2D_calcTantVelocity()

cfd.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(u,i  ,j+1) - L(u,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(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

cfd.c

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

```
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.

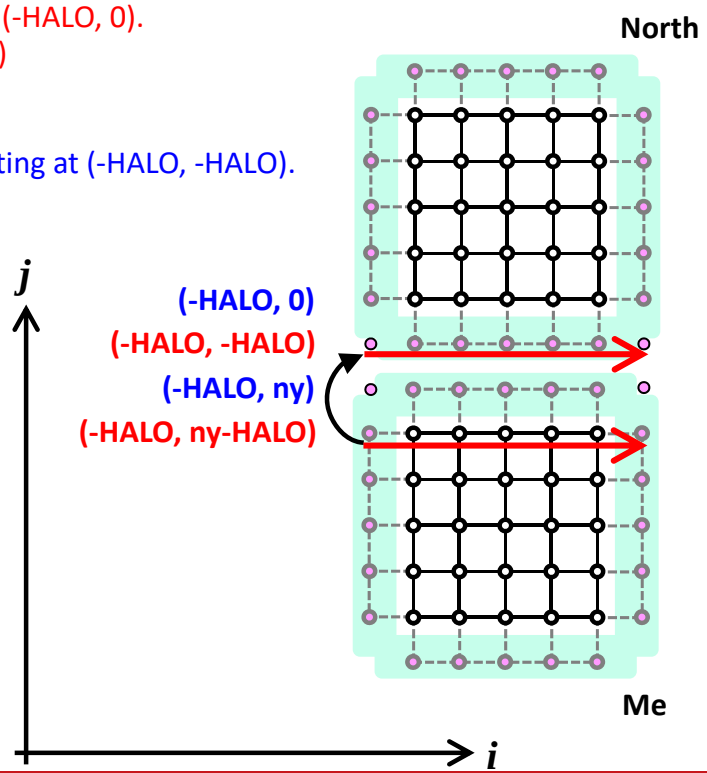
You can use array2D's **double *l_send, *r_send, *l_recv, *r_recv;** as buffers for Halo communication. Memory regions are allocated in array2D_resize().

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?

Hands-on : MPI-parallelized CFD simulation

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./ -lm

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

```
klogin9$ make
================================================================================
= Compilation starts for solver_fractional.
================================================================================
mpifccpx  -Kfast,openmp -I./ -o main.o -c main.c
mpifccpx  -Kfast,openmp -I./ -o cfd.o -c cfd.c
              PLEASE IGNORE WARNINGs
mpifccpx  -Kfast,openmp -I./ -o domain_decomp.o -c domain_decomp.c
mpifccpx  -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     0     1
s    0     0     1     0     0     0     0     0     0     1
JOB_ID  JOB_NAME       MD  ST  USER     GROUP     START_DATE      ELAPSE_TIM  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 bellow
    (No need to change the parameters in cfd.h)
  - ✓ Draw the graph: # of MPI processes vs. Elapsed time

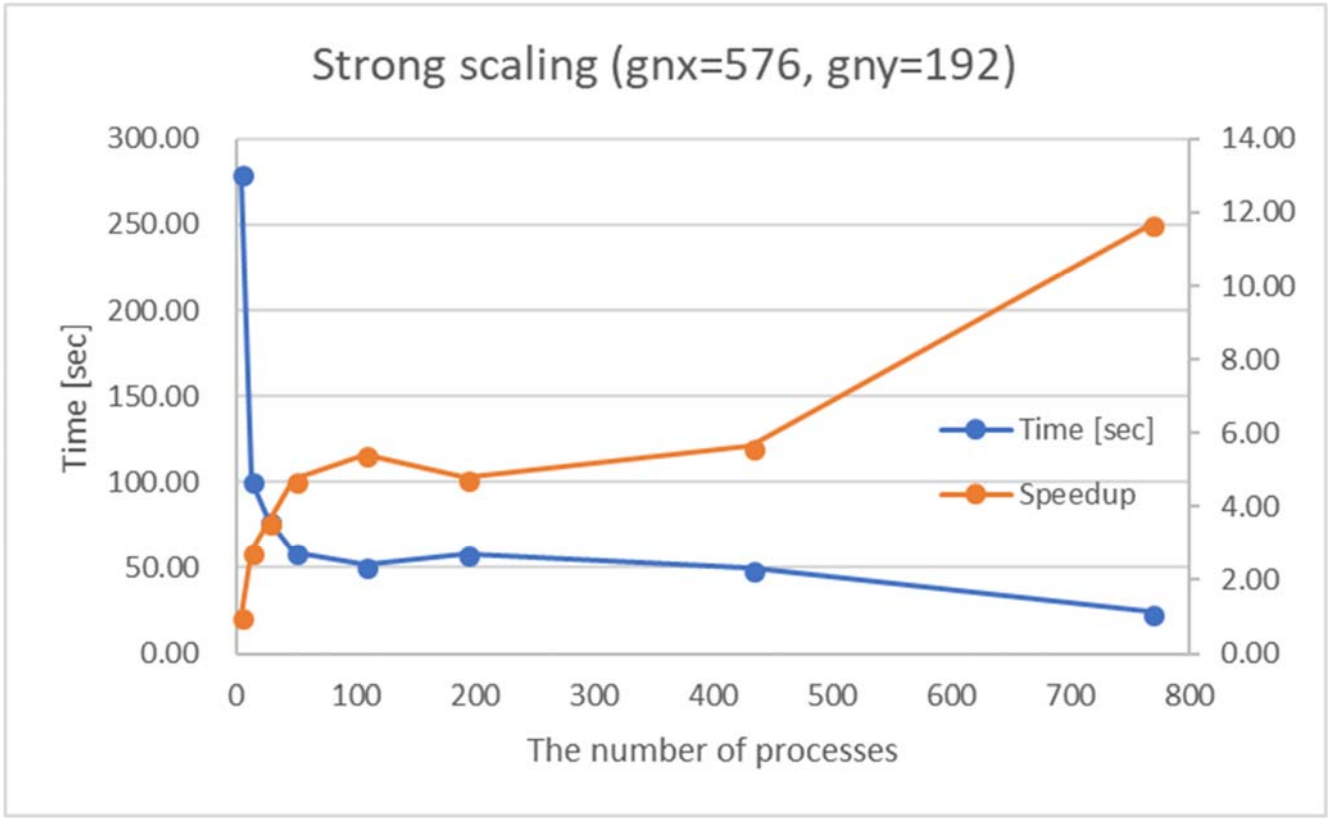
n	procs	Time [sec]	Speedup	(ideal)	grid pointses per proc	nx=ny	gnx	gny	DT	END_TIMESTEP
1	3	?	?	1	36864	192	576	192	0.00003	20000
2	12	?	?	4	9216	96	576	192	0.00003	20000
3	27	?	?	9	4096	64	576	192	0.00003	20000
4	48	?	?	16	2304	48	576	192	0.00003	20000
6	108	?	?	25	1024	32	576	192	0.00003	20000
8	192	?	?	36	576	24	576	192	0.00003	20000
12	432	?	?	49	256	16	576	192	0.00003	20000
16	768	?	?	64	144	12	576	192	0.00003	20000
(24)	1728	?	?	81	64	8	576	192	0.00003	20000

Observe Speedup by Changing num_proc

- **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 bellow, and draw the graph (# of MPI process)

n	procs	Time [sec]	Speedup	(ideal)	grid pointses per proc	nx=ny	gnx	gny	DT	END_TIMESTEP
1	3	?	?	1	10800	3600	180	60	0.000100	2000
2	12	?	?	4	43200	3600	360	120	0.000050	4000
4	48	?	?	9	172800	3600	720	240	0.000025	8000
6	108	?	?	16	388800	3600	1080	360	0.000017	12000
8	192	?	?	25	691200	3600	1440	480	0.000013	16000
10	300	?	?	36	1080000	3600	1800	600	0.000010	20000
14	588	?	?	49	2116800	3600	2520	840	0.000007	28000
17	867	?	?	64	3121200	3600	3060	1020	0.000006	34000
18	972	?	?	81	3499200	3600	3240	1080	0.000006	36000

Strong scaling (gnx=576, gny=192)



Weak scaling (60x60 per each proc)