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 is Computational Fluid Dynamics (CFD) simulation?

- Prediction of the Drag by 2.3 billion meshes.
- Simulation of high velocity air flow around the Space Shuttle during re-entry.
- Simulation of 2D viscous flow with circular obstacle.
How to Compute Fluid Flow?

Initialize

\[ t = t + \Delta t \]

Grid update for \( \Delta t \)

Terminate?

no

yes

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

How to update?
Incompressive Viscous Fluid Flow

Governing Equations with partial differential equations

**Equation of continuity**
(incompressive flow)

\[ \nabla V = 0 \]

**Navier–Stokes equations**
(incompressive flow)

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>( \nu \equiv \frac{\mu}{\rho} )</th>
<th>kinematic viscosity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V )</td>
<td>velocity ((u, v))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P )</td>
<td>pressure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \rho )</td>
<td>density</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**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 $\phi^{n+1}$ of the next time-step with $V^*$ by solving the Poisson’s equation.

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

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

$$V^{n+1} = V^* - \Delta t \nabla \phi^{n+1} \quad (3)$$
Finite Difference Schemes

We can make discrete forms by substituting difference schemes.

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

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

Central difference schemes
(=> Finite difference scheme)

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

See “staggered mesh” for more advanced study.
Step 1: Calculate the tentative velocity: \( u^*, v^* \)

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

\( NU \) 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.

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

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

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

\[ v_{i,j}^{\text{true}} = v_{i,j}^* - \Delta t \left( \frac{\phi_{i,j+1}^* - \phi_{i,j-1}^*}{2\Delta y} \right) \]
Stencil Computation

Common form in Steps 1, 2, and 3

\[ q_{i,j}^{\text{new}} = A + Bq_{i,j} + Cq_{i+1,j} + Dq_{i-1,j} + Eq_{i,j+1} + Fq_{i,j-1} \]

Each point is computed only with its adjacent points.
Data Dependency among Steps

\[
\begin{align*}
&\text{u}(i,j) \quad &\text{v}(i,j) \quad &\text{uTant}(i,j) \quad &\text{vTant}(i,j) \quad &\text{d}(i,j) \quad &\phi(i,j) \quad &\phi_{\text{Temp}}(i,j) \\
&\text{calcTantVelocity} \\
&\text{calcPoissonSourceTerm} \\
&\text{calcPoisson}_{\text{Jacobi}} \quad \text{(repeated)} \\
&\text{calcVelocity}
\end{align*}
\]
Hands-on : Let’s read the codes!

Go to your home directory
@obcx02 cd

Create work directory
@obcx02 mkdir programs_cfd
@obcx02 cd programs_cfd

Copy tgz archive of source files
@obcx02 cp /work/gt57/t57004/share/serial_0920_final.tgz ./
@obcx02 tar xzvf serial_0920_final.tgz
@obcx02 cd serial_0920/

Decompress it

Source files – You modify them!
(codes)

Rules for compilation with “make”

Information on how to compile, execute, etc.

Script programs for execution and visualization

@obcx02 ls

cfd.cpp

cfd.h

main.cpp

main.h

stopwatch3.h

Makefile

README.txt

scripts
Program Structure

- **Data structures (cfd.h)**
  - typedef struct array2D_ { ... } array2D;  // 2D array of a scalar value
  - typedef struct grid2D_ { ... } grid2D;   // 2D grid for fluid using multiple array2Ds

- **Functions for array2D**
  - 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; // 2D array of a scalar value
  - typedef struct grid2D_ { ... } grid2D; // 2D grid for fluid using multiple array2Ds

- **Functions for grid2D**
  - void grid2D_initialize(grid2D *g, ...); // Initialize 2D grid (row x col) for CFD
  - void grid2D_calcTantVelocity(grid2D *g); // Step 1 of Fractional-step method
  - void grid2D_calcPoissonSourceTerm(grid2D *g); // Step 2 (Calculation of source terms)
  - void grid2D_calcPoisson_Jacobi(grid2D *g, ...); // Step 2 (Iterative solver: time-consuming)
  - void grid2D_calcVelocity(grid2D *g); // Step 3
  - void grid2D_calcBoundary_Poiseulle(grid2D *g, ...); // Set boundary condition for top & bottom walls
  - void grid2D_calcBoundary_SqObject(grid2D *g, ...); // Set boundary condition for a square obstacle
  - void grid2D_outputAVEseFile(grid2D *g, ...); // Output a grid data to a file
  - inline int grid2D_getRow(grid2D *g); // Get size of row
  - inline int grid2D_getCol(grid2D *g); // Get size of col
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 2020_0919
 * All rights reserved.
 * (C) Copyright Kentaro Sano 2018.6-
 */
#ifndef ___MAIN_H___
#define ___MAIN_H___
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cfd.h"
#include "stopwatch3.h"
#endif

int main(int argc, char** argv)
{
... tstep = 0;
grid2D_initialize(&g, ROW, COL, PHI_IN, PHI_OUT);
printf("====== Computation started for (%d x %d) grid with dT=%f\n", ROW, COL, DT);
while(tstep < END_TIMESTEP) {
    time2.start();
    tstep_start = tstep;
    fractionalStep_MainLoop(&g, SAVE_INTERVAL);
    time2.stop();
    printf("[tstep=%5d to %5d] (%f sec)", tstep_start, tstep, time2.get());
    grid2D_outputAVEseFile(&g, "AVEse", tstep, 240.0/grid2D_getRow(&g));
}

    time.stop();
    printf("====== Computation finished.\n");
    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++;
    }
}
#ifndef __CFD_H__
define __CFD_H__

#include <string.h>
...

// You can select one of the conditions.

#if defined CONDITIONX
#define CONDITION0
#elif defined CONDITION1
#elif defined CONDITION2
#elif defined CONDITION3
#endif

//===================================================================
#define TARGET_RESIDUAL_RATE (1.0e-2) // Termination condition
#define SAVE_INTERVAL JACOBIREP_INTERVAL // Interval to save file

#define HEIGHT 0.5 // Grid Height is set a length of 0.5 (dimension-less 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
// 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_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 double at(array2D *a, int i, int j) { return *(a + i + j * a->row); }  // Look up value at (row, col)
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 + i + j * a->row); }  // get pointer 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);  // initialize grid2D : row x col
void grid2D_calcTantVelocity(grid2D *g);  // calculate tentative velocity
void grid2D_calcPoissonSourceTerm(grid2D *g);  // calculate source term for poisson's equation
void grid2D_calcVelocity(grid2D *g);  // calculate velocity
void grid2D_calcBoundary_Poiseulle(grid2D *g, double phi_in, double phi_out);  // calculate boundary condition for poiseulle
void grid2D_calcBoundary_SqObject(grid2D *g, int obj_x, int obj_y, int obj_w, int obj_h);  // calculate boundary condition for square object
void grid2D_outputAVEseFile(grid2D *g, char *base, int num, double scaling);  // output average file
inline int grid2D_getRow(grid2D *g) { return( array2D_getRow(&g->u) ); }  // get size of row
inline int grid2D_getCol(grid2D *g) { return( array2D_getCol(&g->u) ); }  // get size of col
inline double L(array2D *a, int i, int j) { return(at(a,i,j)) ; }  // Look up value 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.
", i, j, a->row, a->col);
  exit(EXIT_FAILURE);
}
#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)) = *(at(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++)
        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 %dx%d array in at(). Abort.\n", x, y, a->row, a->col);
        //exit(EXIT_FAILURE);
        return ret;
    }

    ret = ((double)\(a, int_x , int_y \))*(1.0-dx) +
          ((double)\(a, int_x+1, int_y \))*dx*\(1.0-dy\) +
          ((double)\(a, int_x , int_y+1\))*dx*dy;
    return ret;
}

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->phiTant, row, col);
    array2D_initialize(&g->vTant, row, col);
    array2D_initialize(&g->d, row, col);
    array2D_clear (&g->u, 0.01);
    array2D_clear (&g->v, 0.00);
    array2D_clear (&g->phi, 0.0);
    array2D_clear (&g->phiTemp, 0.0);
    array2D_clear (&g->phiTant, 0.00);
    array2D_clear (&g->vTant, 0.00);
    array2D_clear (&g->d, 0.0);

    // Initialize the pressure field with constant gradient
    double *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);
}
```c
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(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(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
                + 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
                + 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 ;
        }
}
```

```c
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 +
               (L(vT,i ,j+1) - L(vT,i ,j-1)) / DY ) / DT;
        }
}
```

```c
void grid2D_calcPoisson_Jacobi(grid2D *g, double target_residual_rate)
{
    int i,j,k=0;
    register double const1 = DX2*DY2/(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;

    array2D_copy(&(g->phi), &(g->phiTemp));
```

```cpp
#pragma omp parallel private(i, loc_residualMax, loc_residual)
{
  // Jacobi iteration
  do { // Loop to set phiTemp by computing with phi
   #pragma omp for
    for (i=1; i<row_m_1; i++)
      for (j=1; j<col_m_1; j++)
        *(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));
  #pragma omp barrier
  #pragma omp single
  { k++;
    grid2D_calcBoundary_SqObject(g, OBJ_X, OBJ_Y, OBJ_W, OBJ_H);
    residualMax_prev = residualMax;
    residualMax = 0.0;
  }
  // Calculate residual
  loc_residualMax = 0.0;
  #pragma omp for
  for (i=1; i<row_m_1; i++)
    for (j=1; j<col_m_1; j++)
      { loc_residual = fabs(L(phi,i,j) - L(phiT,i,j));
        if (loc_residualMax < loc_residual) loc_residualMax = loc_residual;
      }
  #pragma omp barrier
  #pragma omp single if (k == 1) residualMax_1st = residualMax;
  // Loop to set phi by computing with phiTemp
  #pragma omp for
  for (i=1; i<row_m_1; i++)
    for (j=1; j<col_m_1; j++)
      *(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));
}

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 (i=1; i<row_m_1; i++)
    for (j=col_m_1; j++) {
      *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

```c
void grid2D_calcBoundary_Poiseulle(grid2D *g, double phi_in, double phi_out) {
    // COL-1 A
    //   | => flowing dir
    //   | => 0
    // 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(phi,i,j1)) = phi_in;
        *(at(phi,i,j1+1)) = phi_in;
        *(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);
    }

#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,j)) = phi_in;
        *(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);
    }
}
```
void grid2D_calcBoundary_SqObject(grid2D *g, int obj_x, int obj_y, int obj_w, int obj_h) 
{ 
    // j 
    // | ++ 
    // | | # 
    // | | | + 
    // +--------------
    //                     --> i 
    // int sta_i = (int)(obj_x - obj_w/2); // pos of left surface
    int sta_j = (int)(obj_y - obj_h/2); // pos of bottom surface
    int end_i = (int)(sta_i + obj_w); // pos of right 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
    
    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(v,i1+1,j)) = L(v,i1-1,j) + ((2.0*NU/DX)*L(u,i1-1,j)); 
        *(at(phi,i1,j)) = L(phi,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(v,i2+1,j)) = L(v,i2+1,j) + ((2.0*NU/DX)*L(u,i2+1,j)); 
        *(at(phi,i2,j)) = L(phi,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
    
    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 the time consuming part is already parallelized by using OpenMP. See “#pragma omp parallel private(i)” in grid2D_calcPoisson_Jacobi().
Compile and Execute Interactively

```bash
[t57004@obcx04 serial_0920]$ ./scripts/do_clean.sh
rm -f solver_fractional main.o cfd.o depend_c.inc depend_cpp.inc *.o *.dat *.sh.[eos]*
rm: cannot remove 'stdout.lst': No such file or directory
rm: cannot remove 'err': No such file or directory
```

**Clean up previous compilation and computational results. (Recommended if you modified program.)**

```bash
[t57004@obcx04 serial_0920]$ make
========================================================================
= Compilation starts for solver_fractional.                          
========================================================================
```

**Compile (OpenMP is already used here).**

```bash
icc -O3 -xCORE-AVX512 -align -qopenmp -no-multibyte-chars -ipo -L/ -o main.o -c main.cpp
icc -O3 -xCORE-AVX512 -align -qopenmp -no-multibyte-chars -ipo -L/ -o cfd.o -c cfd.cpp
icc -o solver_fractional main.o cfd.o -qopenmp -L/ -lm
main.cpp(4): remark #15009: main has been targeted for automatic cpu dispatch
```

**Execute program on a login server**

```bash
[t57004@obcx04 serial_0920]$ ./scripts/do_execute_on_frontend.sh
================
Computation started for (540 x 180) grid with dT=0.000025.
> 104 iterations in Jacobi (tstep= 0, residualMax=0.006943), [tstep= 0 to 200] (0.879091 sec) > AVEse_000200.dat
>  6 iterations in Jacobi (tstep=24800, residualMax=0.073900), [tstep=24800 to 25000] (0.312015 sec) > AVEse_025000.dat
================
Elapsed time for entire execution = 120.261 sec
```

**Computational results are in "sim_data/", which is automatically created by "do_execute_on_frontend.sh"**

```bash
[t57004@obcx04 serial_0920]$ ls
Makefile README.txt cfd.cpp cfd.h cfd.o depend_cpp.inc main.cpp main.h main.o old scripts sim_data solver_fractional stopwatch3.h
```

**Readme.txt is also available for your reference.**
**Speed up Execution by OpenMP**

Set the number of OpenMP threads
(Try 1, 2, 4, 8, 16, ..., 56)

- Compile and execute with a different number of OpenMP threads
  - 1, 2, 4, 8, 16, 20, 28, 32, 48, 56

- How scalable is it?
  - When 8 times more threads are used, is the exec time reduced to 1/8?

Please do NOT run with many OMP threads frequently.
Cores of frontend servers are Limited!
Execute with Batch-job Scheduler

Input job script "./scripts/go1.sh" into a job queue.

Check the status of my job queue.

Delete a job in the queue.

Please use Batch-job mode mainly.

- Settings and executed program (script) are written in "go1.sh".
  - You can edit them.
- Standard output / error output are written into a file.
  - such like "go1.sh.o541501"
  - tail -f go1.sh.o541501 for watching the file

```
#!/bin/sh
#PJM -N "go1.sh"
##PJM -L rscgrp=lecture7
#PJM -L rscgrp=lecture
#PJM -L node=1
#PJM --omp thread=16
#PJM -L elapse=00:15:00
#PJM -g gt57
#PJM -j
export KMP_AFFINITY=granularity=fine,compact
../../../scripts/do_execute_on_frontend.sh
```

= OMP_NUM_THREADS
= Max. execution time allowed
= executed program
Visualize Computational Results

Convert \( ./\text{sim\_data/}*\).dat to png files, and pop up an animation window for them. (X-window server is required on your PC.)

Just animate png files existing in \( ./\text{sim\_data/}\)

Animation speed depends on network bandwidth between OCBX machine and your home.

If it's too slow, try the followings to make mp4 file

Then, download the mp4 file, and view it on your PC.

README.txt is also available for your reference.
Change Simulation Parameters

- You can select one of the predefined conditions in "cfd.h"
  // You can select one of the conditions.
  //#define CONDITIONX
  //#define CONDITION0
  #define CONDITION1
  //#define CONDITION2
  //#define CONDITION3

✓ To select, uncomment another line.

- Try to change the condition and run
  ✓ How does the exec time change?

```c
#ifndef CONDITIONX
#define ROW (2160)  // cell resolution for row
#define COL (720)   // cell resolution for column
#define DT (0.0000075) // delta t (difference between timesteps)
#define NU (0.0075)  // < 0.01 for Karman vortices
#define JACOBIREP_INTERVAL (500) // interval to report in Jacobi
#define END_TIMESTEP (80000) // tstep to end computation
#endif

#if defined CONDITIONX
  // Flow condition X (taking super long time)
  #define ROW (2160)
  #define COL (720)
  #define DT (0.0000075)
  #define NU (0.0075)
  #define JACOBIREP_INTERVAL (500)
  #define END_TIMESTEP (80000)
#endif

#elif defined CONDITION0
  // Flow condition 0 (taking very long time)
  #define ROW (1080)
  #define COL (360)
  #define DT (0.000015)
  #define NU (0.0075)
  #define JACOBIREP_INTERVAL (250)
  #define END_TIMESTEP (40000)
#endif

#elif defined CONDITION1
  // Flow condition 1 (taking long time)
  #define ROW (540)
  #define COL (180)
  #define DT (0.000025)
  #define NU (0.0075)
  #define JACOBIREP_INTERVAL (200)
  #define END_TIMESTEP (25000)
#endif

#elif defined CONDITION2
  // Flow condition 2
  #define ROW (360)
  #define COL (120)
  #define DT (0.00005)
  #define NU (0.0075)
  #define JACOBIREP_INTERVAL (150)
  #define END_TIMESTEP (20000)
#endif

#elif defined CONDITION3
  // Flow condition 3 (easy condition, fast execution)
  #define ROW (180)
  #define COL (60)
  #define DT (0.00005)
  #define NU (0.0075)
  #define JACOBIREP_INTERVAL (100)
  #define END_TIMESTEP (16000)
#endif
```

360 x 120 grid
delta T = 0.00005
Interval for file save = 150
End of time step = 20000
PART-II
Parallelization of
the 2D CFD Simulation
Overview

- **Parallelization with "shared memory", which is done by OpenMP, is limited to a node.**
  - Many cores in multiple sockets share the same memory space.

- **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 performed 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

Serial program

MPI-parallel program

- calcTantVelocity
- calcPoissonSourceTerm
- calcPoisson_Jacobi
- calcPoisson_Jacobi (repeated)
- calcVelocity

- Halo exchg. of u, v
  - calcTantVelocity

- Halo exchg. of uTant, vTant
  - calcPoissonSourceTerm

- Halo exchg. of phi
  - calcPoisson_Jacobi

- Halo exchg. of phiTemp
  - calcPoisson_Jacobi (repeated)

- Halo exchg. of uTant, vTant, phi
  - calcVelocity
Let's Read the parallelized Code!

@obcx02 cd programs_cfd
@obcx02 cp /work/gt57/t57004/share/parallel_complete_0920_r1.tgz ./
@obcx02 tar zxfp parallel_complete_0920_final.tgz
@obcx02 cd parallel_complete_0920/

@obcx02 ls
cfd.cpp
cfd.h
domain_decomp.cpp
domain_decomp.h
main.cpp
main.h
Makefile
README.txt
scripts

MPI parallelization is introduced.

New files.
Codes for subgrid management.
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;

// 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 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 at (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) { return (a+i\*nx+j); } // get pointer at (nx, ny)

# If Debug
if ((i<0‐HALO) || (i>=(a‐nx+HALO)) || (j<0‐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+i+j * (a‐nx+2*HALO));

...
#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

Details of Domain Decomposition

num_procs = 12
dims[0] = sqrt(12/3) = 2
 dims[1] = 12 / 2 = 6

In the case that me_proc == 5,
mpd->coord[1] = 5 % 6 = 5;
mpd->coord[0] = 5 / 6 = 0;
mpd->east = MPI_PROC_NULL
mpd->west = me_proc - 1 = 4
mpd->north = me_proc + mpd->dims[1] = 5 + 6 = 11
mpd->south = MPI_PROC_NULL

This is the case where n = 2, with 3*2^2 = 12 procs
### 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 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;
}
```

---

New file: 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->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];`
- `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()

```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(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);
        }
    communicate_neighbor(uT, mpd);
    communicate Neighbor(vT, mpd);
}
```

Modify start\_\{x,y\} and end\_\{x,y\} for a sub-grid with Halo region

Exchange Halo with neighbor MPI processes (see Next Page).
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 read the code written here to understand MPI communications.
}
```

**Hint to understand:**
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 with MPI?

To obtain the top Halo of mine with south subgrid,

The row of \((nx+2*\text{HALO})*\text{HALO}\) cells starting at \((-\text{HALO}, ny-\text{HALO})\) should be sent to the bottom Halo of the south at \((-\text{HALO}, -\text{HALO})\).

* The coordinate of origin in the subgrid is \((0, 0)\)

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

**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 by Batch

```bash
[t57004@obcx04 parallel_0920]$ ./scripts/do_clean.sh
...
[t57004@obcx04 parallel_0920]$ make
========================================================================
= Compilation starts for solver_fractional.
========================================================================
...
[t57004@obcx04 parallel_0920]$ pjsub ./scripts/go3.sh
pjsub scripts/go3.sh
[INFO] PJM 0000 pjsub Job 541545 submitted.
[t57004@obcx04 parallel_0920]$ pstat
Oakbridge-CX scheduled stop time: 2020/09/25(Fri) 09:00:00 (Remain: 4days 13:59:26)

JOB_ID   JOB_NAME  STATUS   PROJECT   RSCGROUP   START_DATE  ELAPSE   TOKEN NODE
541552   go3.sh   RUNNING  gt57      lecture   09/20 19:00:06   <00:00:28    -  4

[t57004@obcx04 parallel_0920]$ ls go3.sh.o*
go3.sh.o541501
[t57004@obcx04 parallel_0920]$ less go3.sh.o541501
...
[t57004@obcx04 parallel_0920]$ tail -f go3.sh.o541501
...
```

- **Input job script "./scripts/go3.sh" into a job queue.**
- **Or, try "watch -n 1 pstat"**
- **Watch the N last lines added to the file.**
- **If you want to kill a job,**
  ```bash
  > pjdel <Job ID>
  ```
Batch Job Script : go3.sh

- Check the output file of MPI-parallel execution
  - less go3.sh.o541501
  - The last line show the execution time and the number of MPI processes.
Measure Exec Time without Saving Files

When you measure the elapsed time by excluding file-writing time, please un-comment 37th line and comment out 36th line in Makefile.

```make
ifeq (${BASE_COMPILER},mpiicpc)
CFLAGS = -O3 -axCORE-AVX512 -align -qopenmp -no-multibyte-chars -ipo $(INCLUDE_DIR)
# CFLAGS = -O3 -axCORE-AVX512 -align -qopenmp -no-multibyte-chars -ipo $(INCLUDE_DIR) -DMEASURE_TIME
↓
ifeq (${BASE_COMPILER},mpiicpc)
# CFLAGS = -O3 -axCORE-AVX512 -align -qopenmp -no-multibyte-chars -ipo $(INCLUDE_DIR)
CFLAGS = -O3 -axCORE-AVX512 -align -qopenmp -no-multibyte-chars -ipo $(INCLUDE_DIR) -DMEASURE_TIME
```

Then, read the last line of the output file:

```
Time-step=40000 : (MPI-Procs, Elapsed Time)=$(3, 754.547 sec), (MPI*OpenMP, Time)=$(3, 754.547 sec)
```

for MPI parallel  
for OMP-MPI hybrid parallel
Observe Speedup by Changing #PJM --mpi proc

Strong scaling
✓ Parallel computation with 3n^2 MPI processes for the same grid size
✓ Measure execution time by changing n = 1, 2, 3, 4, ... 12 (3n^2 = 432)
  ➢ Don't forget to "un-comment 37th line and comment out 36th line in Makefile."
  ➢ Don't change the size of Grid (Condition in cfd.h)
✓ Fill out the table as bellow
✓ Draw the graph: # of MPI processes vs. Elapsed time

<table>
<thead>
<tr>
<th>Strong</th>
<th>Condition 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>MPI procs</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>108</td>
</tr>
<tr>
<td>8</td>
<td>192</td>
</tr>
<tr>
<td>10</td>
<td>300</td>
</tr>
<tr>
<td>12</td>
<td>432</td>
</tr>
</tbody>
</table>
How to Make a Graph using "gnuplot"

✓ create a data file with text editor (e.g., vim, emacs)
  ➢ Write X-axis data in 1st column, Y-axis data in 2nd column
  ➢ insert a space between the columns
✓ execute gnuplot in your terminal & type following commands

```
[t57004@obcx04 parallel_0920]$ gnuplot
GNUPLOT
Version 4.6 patchlevel 2  last modified 2013-03-14
...
Terminal type set to 'x11'
gnuplot> set xlabel 'the number of processes'
gnuplot> set ylabel 'speedup'
gnuplot> plot "./graph_speedup.txt" with line
```

For calculation, you can use “bc −l” command (“−l” is of a small character of “−L”)
### Observe Speedup by Changing Problem Size

**The larger grid, The better speedup?**

- ✓ Measure execution time and obtain speedups for Condition 1, 0, X
- ✓ Draw graphs against (MPI procs)
- ✓ How do Speedup change? And why?

#### Strong Condition 1

<table>
<thead>
<tr>
<th>n MPI procs</th>
<th>Time (sec)</th>
<th>Speedup (ideal)</th>
<th>grid points per proc nx=ny</th>
<th>gnx</th>
<th>gny</th>
<th>END_TIMESTEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>?</td>
<td>1</td>
<td>14400</td>
<td>120</td>
<td>360</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>?</td>
<td>4</td>
<td>3600</td>
<td>60</td>
<td>360</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>?</td>
<td>9</td>
<td>1600</td>
<td>40</td>
<td>360</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>?</td>
<td>16</td>
<td>900</td>
<td>30</td>
<td>360</td>
</tr>
<tr>
<td>6</td>
<td>108</td>
<td>?</td>
<td>25</td>
<td>400</td>
<td>20</td>
<td>360</td>
</tr>
<tr>
<td>8</td>
<td>192</td>
<td>?</td>
<td>36</td>
<td>225</td>
<td>15</td>
<td>360</td>
</tr>
<tr>
<td>10</td>
<td>300</td>
<td>?</td>
<td>49</td>
<td>144</td>
<td>12</td>
<td>360</td>
</tr>
<tr>
<td>12</td>
<td>432</td>
<td>?</td>
<td>81</td>
<td>100</td>
<td>10</td>
<td>360</td>
</tr>
</tbody>
</table>

#### Strong Condition 0

<table>
<thead>
<tr>
<th>n MPI procs</th>
<th>Time (sec)</th>
<th>Speedup (ideal)</th>
<th>grid points per proc nx=ny</th>
<th>gnx</th>
<th>gny</th>
<th>END_TIMESTEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>?</td>
<td>4</td>
<td>32400</td>
<td>180</td>
<td>540</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>?</td>
<td>4</td>
<td>8100</td>
<td>90</td>
<td>540</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>?</td>
<td>9</td>
<td>3600</td>
<td>60</td>
<td>540</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>?</td>
<td>16</td>
<td>2025</td>
<td>45</td>
<td>540</td>
</tr>
<tr>
<td>6</td>
<td>108</td>
<td>?</td>
<td>25</td>
<td>900</td>
<td>30</td>
<td>540</td>
</tr>
<tr>
<td>8</td>
<td>192</td>
<td>?</td>
<td>36</td>
<td>506.25</td>
<td>22.5</td>
<td>540</td>
</tr>
<tr>
<td>10</td>
<td>300</td>
<td>?</td>
<td>49</td>
<td>324</td>
<td>18</td>
<td>540</td>
</tr>
<tr>
<td>12</td>
<td>432</td>
<td>?</td>
<td>81</td>
<td>225</td>
<td>15</td>
<td>540</td>
</tr>
</tbody>
</table>

#### Strong Condition X

<table>
<thead>
<tr>
<th>n MPI procs</th>
<th>Time (sec)</th>
<th>Speedup (ideal)</th>
<th>grid points per proc nx=ny</th>
<th>gnx</th>
<th>gny</th>
<th>END_TIMESTEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>?</td>
<td>4</td>
<td>518400</td>
<td>180</td>
<td>2160</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>?</td>
<td>4</td>
<td>129600</td>
<td>360</td>
<td>1080</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>?</td>
<td>9</td>
<td>32400</td>
<td>180</td>
<td>1080</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>?</td>
<td>16</td>
<td>14400</td>
<td>120</td>
<td>1080</td>
</tr>
<tr>
<td>6</td>
<td>108</td>
<td>?</td>
<td>25</td>
<td>8100</td>
<td>90</td>
<td>1080</td>
</tr>
<tr>
<td>8</td>
<td>192</td>
<td>?</td>
<td>36</td>
<td>57600</td>
<td>240</td>
<td>1080</td>
</tr>
<tr>
<td>10</td>
<td>300</td>
<td>?</td>
<td>49</td>
<td>32400</td>
<td>180</td>
<td>1080</td>
</tr>
<tr>
<td>12</td>
<td>432</td>
<td>?</td>
<td>81</td>
<td>51840</td>
<td>180</td>
<td>1080</td>
</tr>
</tbody>
</table>

Note: Computation of Condition X with 3 or 12 MPI procs takes more than 15 min. If you execute it, you need to increase “elapsed time” in go3.sh

This cannot be executed due to some mismatch between the grid size and # of procs.
Strong Scaling Example

The number of processes

The diagram shows the speedup of processes in relation to the number of nodes.

- **4 nodes**
  - Condition 0 (grey line): Performance drops because the four nodes have only 224 cores, which are less than the cores required for parallel execution.
  - Condition 1 (orange line)
  - Condition 2 (blue line)

What's happen with 8 nodes (448 cores?)

Performance drops because the four nodes have only 224 cores, which are less than the cores required for parallel execution.

What's happen with 8 nodes (448 cores?)
Observe Speedup by Hybrid Parallel

If we combine OpenMP and MPI, how do speedups change?

- Edit go3.sh for  `#PJM --omp thread=2, 4, 8`
- Draw graphs against (OMP*MPI threads)
- How do Speedup change? And why?

<table>
<thead>
<tr>
<th>Strong (OMP &amp; MPI)</th>
<th>Condition 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OMP</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strong (OMP &amp; MPI)</th>
<th>Condition 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OMP</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>
More Advanced Exercise

- **Read the codes and optimize them to further speed up execution**
  - Find the optimum numbers for MPI procs and OMP threads; of best hybrid
  - Remove unnecessary codes
  - Reduce the number of barriers IF possible
  - Add OpenMP parallelization to functions that are not parallelized yet

- **Try more advanced modification for speedup**
  - Reduce the number of residual computation (this may change simulation results)
    - Now 1 residual computation per 2 Jacobi computations
    - What’s happen if we have 1 residual computation per 4 Jacobi computations?
      (For speedup, we need to remove unnecessary “barrier”, “critical”, “single” sections)

- **Try what you propose to do …**

- **When you accomplish something interesting, please write it to Slack ch!**