Step by Step of Allen-Cahn equation Test Code

2018/04/30 updated

1. Prerequisites

- download the source code of the mini test framework from Github https://github.com/xjtju/PinT
- read the overview of the code from the homepage and related wiki pages, get a overall understanding of the code https://github.com/xjtju/PinT/wiki

2. Key implementation of the code

Necessary comments are embedded in the source files, when you want to understand the code further or modify it for your problem, please check these comments. Though object oriented C++ is used to improve the extensibility of the code, variables and methods of objects/classes are not rigorously encapsulated in order to operate them easily. That is you can access most inner variables of most classes directly in your code, of coarse reading operation is no problem, but please be careful to make a modification. Note that there is no easy way to treat 1D, 2D, 3D in one same segment of code simultaneously , the automatically switch between multiple dimensions is implemented by the traditional if/else (switch/case) clause.

2.1 Grid - space parallel

MPI cartesian topology routines are used to simplify space division and communication. The grid initializes and manages five general variables of parareal algorithm. You can directly manipulate these variables through the pointers of them.

```
double *u_f;
double *u_c;
// the temporary solution variables of fine/coarse solver,
// before starts a new iteration, the Driver will copy the correct initial value to u_f/u_c

double *u_cprev;
// the structure holder for the coarse solver at the previous iteration of the the same slice

double *u_start;  // the latest solution of the current time slice at start point or the previous slice end
double *u_end;   // the latest solution of the current time slice at end point or the next slice start
double *u;          // only a pointer, pointing the same variable with u_end
```

Various solvers are responsible to manage their own variables, most of them have the same structure with the five general grid variables, usually used to restore the temporary values during the computing process. With the help of Grid, you can output these variables in text or HDF5 format at anytime, see Output for detail.

Grid can apply boundary condition and synchronize guard cells automatically according the parameter file for any variables with the same size of the grid. Usually it is not necessary to invoke them manually except when some new variable or solver is introduced.

Topology:

```
X (0 ... i) :              left    …   right
Y (0 ... j) :            front    …   back
Z (0 ... k) : down(under)    …   up
```

In source code, '**l, r, f, b, d, u**' are used to indicate the position or direction of the grid. For example: function 'bc_val_1d_l()' means applying boundary condition to the left of x in 1D.

Guard cell:

Grid will create its variables containing correct guard cells according the parameter file, you can call the function void Grid::guardcell(double* d) explicitly for any variable created in your code. If you want

to introduce a new parallel linear solver, it is inevitably to synchronize guard cells when change happens. See examples in PBiCGStab, SOR, NewtonSolver etc.

Boundary condition:
Grid can support two types of boundary condition automatically:

- the first is constant boundary (Dirichlet condition),
- the second is homogeneous Neumann boundary (that is du/dx=0 => U_{n+1}=U_{n}).

You can add any boundary condition by overwriting virtual bc() functions in your problem-specific grid. The framework can apply your customized boundary conditions properly. The boundary of each direction can be configured separately or together, the specific settings will overwrite the general settings. See the parameter file and PFMGrid for details.

2.2 Driver - time parallel

The driver implemented the skeleton of the parareal algorithm, and controls the execution flow of the whole program. Some flow control functions are provided, including init, evolve, finalize, Abort et al.

- init: check the consistency of parameter file, and initialize MPI running context…
- evolve: the core template of parareal algorithm
- finalize: collect the performance data, and exit MPI normally
- Abort: abort the running program, and force exit when some error occurs, you can abort a running at anytime and anywhere.

It can support some optimized method, such as pipelined parareal, skipped useless time slices, and relax factor (减速係数) etc. Performance monitoring codes are also embedded in Driver. Maybe, you can develop another Driver for a new parallel-in-time method.

2.2.1 Convergence consideration

Usually the convergence check is problem specific, the default residual check formula is a relative form:

$$res_{abs}^{(K^{par})} = max_n \left[\sqrt{\frac{\sum_{i=1}^{N_{DOF}} |U_{i,n-1}^{K^{par}} - U_{i,n-1}^{K^{par}-1}|^2}{\sum_{i=1}^{N_{DOF}} |U_{i,n-1}^{K^{par}}|^2}} \right]$$

We also used a absolute form in our previous tests:

$$res_{abs}^{(K^{par})} = max_n \left[\sqrt{\frac{\sum_{i=1}^{N_{DOF}} |U_{i,n-1}^{K^{par}} - U_{i,n-1}^{K^{par}-1}|^2}{N_{DOF}}} \right]$$

But in the current version, the above residual calculation formulas can be configured through the parameter file, the .ini file can only support the convergence criteria value (converge_eps). If you want to use the absolute form and other forms dedicated to your problem, you have to modified the source code. The correction and residual calculation is encapsulated in one function of the Driver class.

```
void pint_sum(Grid *grid, double *u, double *f, double *g, double *g_,
                          double *relax_factor, double *res, double *sml)
```

The above function is just an interface, the real calculations are performed by Fortran (blas.f90):

```
subroutine blas_pint_sum_3d(nxyz, ng, u, f, g, g_, factor, res, sml)
   res  = tmp1 =  tmp2 =  u_nrm2 = 0.0
   do …
      tmp1 = ( g(i,j,k) - g_(i,j,k) )*factor + f(i,j,k)  !! correction step of parareal algorithm with relax factor
      tmp2 = u(i,j,k) - tmp1
      u(i,j,k) = tmp1
      res    = res + tmp2*tmp2
      u_nrm2 = u_nrm2 + tmp1*tmp1
   end do
   res  = res / u_nrm2       !! relative
   !res =  res / (ix*jy*kz)    !! absolute
end subroutine blas_pint_sum_3d     !! pseudo code only, see the source file for detail
```

2.3 Output

Output function is supported by the Grid and Output object together. Grid collects data from the whole space domain, calculates the coordinate for each cell, and provides calling interface for outer classes. You can collect and output the entire grid to one single file or output each sub-grid to its local file independently.

```
// flag will be added to the file name, in order to avoid overwriting some existing file
void output_global(const char* flag, bool h5=false);
void output_global_h5(const char* flag);  // the shortcut for hdf5

void output_local_h5(double *data) ;
// for text format, you can choose whether output the guard cell or not for debugging.
void output_local(double *data, bool inner_only);
```

The only thing left for Output is writing the data to file correctly. Two formats are supported: ASCII and HDF5. Theoretically HDF5 is enough, but we found sometimes, text file is very helpful for debugging when the grid size is small.

If using HDF5, it is recommended to download and install the HDFView from its official website, and JRE is necessary to run HDFView. Using this tool, you can explore hdf5 files directly. But HDFView cannot visualize the data, we always need the matplotlib and other related python libraries. Sample scripts for plotting the result file are located in the 'test' directory.

Remind again, you can use Grid to output any variables at anytime for debugging, as long as the variable has the same structure (size) with the basic grid variables (u_f, u_c …).

2.4 Performance Monitor

As the HDF5 output, the performance monitor is also an optional function. If you don't care the result, it is not necessary to install HDF5 library and activate the HDF5 output function. In that case, when you call the hdf5 output methods of the Grid, nothing will happen except a tip. Similarly, the Monitor is also a light wrapper of PMLib (https://github.com/avr-aics-riken/PMlib/), a simple but powerful performance monitor library. When the compile option for PMLib is closed, the Monitor will do nothing. The content of performance report generated by PMLib is quick intuitive, please see our wiki for detail. PMLib already provided several Makefile templates for different compilers and run environments, only a few changes are needed to install it.

At current version, only 5 operations can be monitored:

- Calculation : coarse solver, fine solver, residual check
- Communication: send/receive between time slices

Note that, it is not easy to distinguish the calculation and the communication totally, for example, parallel linear solver has many communication in space domain, and the main time of the residual check is used by the 'all reduce' operation in the entire MPI domain. On the other hand, we can't measure some fine-grained level performance due to the overhead introduced by the monitor itself. This measuring cost will cause the performance data inaccurate. For example, we had measured the guard cell synchronization, the very frequent operation during the program, but the accumulated time measured by PMLib is much bigger than the whole time when it is not monitored.

The monitor only contains several lines of code. If you want to monitor other operations, the only thing need to do is to add a distinct label for the operation and call the timer exactly before and after the operation. See Driver for detail.

There is a problem with the monitor at **supercomputer**. Perhaps due to some I/O cache in these large scale architectures, sometimes the performance report file can't be written to the disk completely, but the I/O redirection is always ok. So the report output code becomes:

```
monitor.printDetail(stderr); // redirect report to the standard error/output, don't use common file
```

3. Developing a problem specific test case

There are two test cases in the current version, the one is the heat equation, a simple case, introduced in the homepage of the project; the other is the Allen-Cahn equation, a more complicated case, have been tested in the ITO supercomputer. In the following, I will give a step by step tutorial on how the build the case based on the base code. The source files are located in the 'pfm' directory.

The steps of integrating a new physical model to the framework are as follows:

- setup the problem-specific parameters (optional)
- setup the problem-specific grid (optional)
- implement the fine solver and coarse solver based on proper numerical schemes
- assemble all the components in a the main function
- prepare parameter file
- compile and run

Before starting programming, let us review the Allen-Cahn equation and related numerical schemes.

3.1 Allen-Cahn equation

$$\begin{aligned}\frac{\partial \phi}{\partial t} &= RHS \\ &= \lambda \nabla^2 \phi - \kappa \phi(\phi - 1.0)(\phi - 0.5 + \beta)\end{aligned} \quad (3.1)$$

where λ is the diffusion coefficient, κ is inversely proportional to the square of the interfacial width, and β is a parameter which impacts the direction and movement speed of phase interface.

3.1.1 Numerical schemes

space : central differencing

time :

- Forward Euler 1st order (EU)
- Crank-Nicolson 2nd order (CN)
- Backward Euler 4th order (BD4) (non-standard parareal, see section 4 for more detail)

For simplicity, 1D case is used to describe the discretization formulas.

$$RHS_i = \frac{\lambda}{\Delta x^2}(\phi_{i-1} - 2.0\phi_i + \phi_{i+1}) - \kappa\phi(\phi_i - 1.0)(\phi_i - 0.5 + \beta) \quad (3.2)$$

$$F_i^{CN}(\phi_{i-1}, \phi_i, \phi_{i+1}) = \phi_i^l - \phi_i^{l-1} - 0.5\Delta t(RHS_i^l + RHS_i^{l-1}) = 0 \quad (3.3)$$

$$F_i^{BD4}(\phi_{i-1}, \phi_i, \phi_{i+1}) = 25\phi_i^l + (-48\phi_i^{l-1} + 36\phi_i^{l-2} + 16\phi_i^{l-3} + 3\phi_i^{l-4}) - 12\Delta t \cdot RHS_i^l = 0 \quad (3.4)$$

where 'i' is the index of grid, 'l' is the index of time step. Newton-Raphson method is used to solve the two nonlinear equations. The final linear equations (Ax=b) derived from Newton method is described by (3.5), where 'k' is the iteration number of the Newton method.

$$\left[\frac{\partial F_i}{\partial \phi_{i-1}}, \frac{\partial F_i}{\partial \phi_i}, \frac{\partial F_i}{\partial \phi_{i+1}}\right] \begin{Bmatrix} d\phi_{i-1}^l \\ d\phi_{i1}^l \\ d\phi_{i+1}^l \end{Bmatrix}^k = -F_i(\phi_{i-1}^{l,k-1}, \phi_i^{l,k-1}, \phi_{i+1}^{l,k-1}) \quad (3.5)$$

For code implementation, we introduced two symbols G1 and G2 to describe to expanded formulas.

CN: (θ=0.5)

$$\begin{aligned}F_i^l &= \phi_i^i - \phi_i^{l-1} - \Delta t\left\{\theta \cdot RHS_i^l + (1-\theta) \cdot RHS_i^{l-1}\right\} \\ &= \phi_i^i - \phi_i^{l-1} - \theta \cdot G2 - (1-\theta) \cdot G1\end{aligned} \quad (3.6)$$

$$G2 = \frac{\Delta t\lambda}{\Delta x^2}(\phi_{i-1}^l - 2\phi_i^l + \phi_{i+1}^l) - \Delta t\kappa\phi_i^l(\phi_i^l - 1.0)(\phi_i^l - 0.5 + \beta) \quad (3.7)$$

$$G1 = \frac{\Delta t\lambda}{\Delta x^2}(\phi_{i-1}^{l-1} - 2\phi_i^{l-1} + \phi_{i+1}^{l-1}) - \Delta t\kappa\phi_i^{l-1}(\phi_i^{l-1} - 1.0)(\phi_i^{l-1} - 0.5 + \beta) \quad (3.8)$$

BD4:

$$\begin{aligned} F_i^l =& 25\phi_i^l + (-48\phi_i^{l-1} + 36\phi_i^{l-2} + 16\phi_i^{l-3} + 3\phi_i^{l-4}) \\ & - 12\left\{\frac{\Delta t\lambda}{\Delta x^2}(\phi_{i-1}^l - 2\phi_i^l + \phi_{i+1}^l) - \Delta t\kappa\phi_i^l(\phi_i^l - 1.0)(\phi_i^l - 0.5 + \beta)\right\} \\ =& 25\phi_i^l + G1 - 12G2 \end{aligned} \tag{3.9}$$

here G2*(BD4)* = G2*(CN)*.

Now the right side of (3.5) is ready, from the left of (3.5), we can derive the following Jacobi matrix, finally the system of linear equation (Ax=b) can be obtained.

$$\frac{\partial F_i}{\partial \phi_i^l} = 1 + \frac{2\theta\Delta t\lambda}{\Delta x^2} + \theta\Delta t\kappa\left\{(\phi_i^{l,k-1} - 1)(\phi_i^{l,k-1} - 0.5 + \beta) + \phi_i^{l,k-1}(\phi_i^{l,k-1} - 0.5 + \beta) + \phi_i^{l,k-1}(\phi_i^{l,k-1} - 1)\right\}$$

$$\frac{\partial F_i}{\partial \phi_{i\pm1}^l} = -\frac{\theta\Delta t\lambda}{\Delta x^2} \tag{3.10-3.11}$$

Note that, do not confuse the two similar symbols ‘κ’(kappa) and ‘k’. ‘κ’ is a physical parameter, see formula (3.1), while ‘k’ or ‘k-1’ is the iteration number of Newton’s method, also see formula (3.5). In source code, the diffusion coefficient ‘λ’ is replaced by ‘d’, and lamda_x/y/z is used to describe a compound value $\lambda \cdot \Delta t/\Delta x^2$, the value is also an indicator of the stability of the explicit Euler method. All the formulas are derived in 1D cases, 2D/3D is similar, see the source code for detail.

3.2 Problem-specific parameter

In order to tune the parameter easily, it is better to add problem-specific parameters into the global parameter file. We used **inih**, an open source .INI file parser, to load all the parameters, the global parameters are managed by the class ‘PinT’. It is recommended to add a dedicated section in the global ini file for clearly separating the framework and the physical modules.

Usually, a simple struct (c++) is enough for holding parameters if there is no complicated operations. In the case, a new struct called ‘PFMParams’ is created, in which besides physical parameters, some frequently used intermediate variables are also defined for reducing duplicated calculations.

```
struct PFMParams {
   double k;       // κ,
   double d;       // λ,  diffuse coefficient (lambda)
   double beta;  // β,

   double theta = 0.5;  // 0.5: Crank-Nicolson; 0: Ex.Euler; 1: Im.Euler
   double lamda_x;     // d*dt/(dx**2)
   int fsolver  = 0;       // 0: CN; 1: EU; 2: EU
   int csolver = 0;       // 0: CN;  1: BD4
}
```

According the specification of **inih**, a callback function must be provided for parsing the ini file. The signature of the function is :

```
int pfm_inih(void* obj, const char* section, const char* name, const char* value);
```

You can employ the wrapper provided by PinT to call **inih** transparently like below:

```
void init(PinT* conf){
      conf->init_module(this, pfm_inih); // load and parse parameters from the global ini file
}
```

We placed the declaration and definition of the callback function in PFMParams.h and PFMGrid.cpp separately, because defining the function in the header file will cause some compiling problems.

3.3 Grid and initialization

It is not always necessary to create a problem-specific grid by inheriting from the basic Grid, but the dedicated grid is helpful for managing the initialization, and implementing any boundary condition. The

PFMGrid has three main functions.
The first is to initialize PFMParams, and the second is to initialize the grid variables. The two steps are described together.

```
void PFMGrid::init() {
    param.init(conf,this);   // call the init method of PFMParams
    if(ndim == 1) init1d();  // init grid variables for 1D/2D/3D separately
    if(ndim == 2) init2d();
    if(ndim == 3) init3d();
}
// the 2D case example
void PFMGrid::init2d(){
    long     ind = 0;     // index for variables
    double r = 0.4;     // side length
    double xdist, ydist, unk; // temporary variables

    for(int j = nguard; j<ny+nguard; j++)  // iterate the local grid
    for(int i = nguard; i<nx+nguard; i++){
      // get the distance of the current point(cell) from the center of the whole space domain
      xdist = this->getX(i) -  conf->Xspan/2 ;
      ydist = this->getY(j) -  conf->Yspan/2 ;

      ind = this->getOuterIdx(i, j, 0);   // get the index of the current point(cell) at the grid
      if( fabs(xdist)<=r )  && fabs(ydist)<=r )
         unk = 1.0;
      else unk = 0.0;

      this->set_val4all(ind, unk);  //  init the current point for all the five basic variables (u_f, u_c …)
    }
}
```

The framework will automatically create a proper local PFMGrid object including the guard cells for current process. It is not necessary to populate guard cells manually when initialization, the framework will do it at proper time. In the init2d function, some helper functions of the basic Grid is used to get the physical coordinate of the point, and the local linear index (including the guard cells). For more helper functions that facilitate the programming, please see the Grid.h source file, where '**inner**' indicates not containing the guard cells, while '**outer**' means containing the guard cells. You can manipulate any variable manually as you want in the initialization, but usually it is not necessary to introduce new variable into the Grid object.

The third function of a problem-specific Grid is to implement boundary conditions that the framework cannot support automatically, it is also the only formal reason why the subclass of the basic Grid exists. It is easy to support a special boundary conditions. Firstly, you need to overwrite the corresponding virtual functions of the super class, for a 1D case example:

```
// specific boundary condition
void bc_1d_l(double *d){ bc_pfm_ac_1d_l_(nxyz, &nguard, d); }
void bc_1d_r(double *d){ bc_pfm_ac_1d_r_(nxyz, &nguard, d); }
```

For easy manipulation of the underlying multi-dimensional data structure, we strongly recommend using FORTRAN language to tackle these array-related tasks. Though it is not absolutely necessary, it is very convenient and efficient.

```
subroutine bc_pfm_ac_1d_l(nxyz, ng, soln)
implicit none
    integer, dimension(3) :: nxyz  !! the shape of the grid, not including the guard cells
    integer ::  ng, i, nx                !! ng is the size of guard cells,  nx is the inner size of the grid
```

```
    real, dimension(1-ng:nxyz(1)+ng ) :: soln   !! remapping the grid array

    nx = nxyz(1)
    soln(1-ng:0) = 2.0 - soln(1)  !! set the left boundary of the x
  end subroutine bc_pfm_ac_1d_l
```

Finally, don't forget to set the correct boundary type in parameter file for that the framework can apply the customized boundary condition automatically.

3.4 Fine solver / Coarse solver

In order to use the general template of parareal algorithm provided by the **Driver** class, the fine solver and coarse solver has to implement the interface defined by the **Solver** class. We also provide two basic implementations of the interface, one is the simple **DefaultSolver** for linear problems, see HeatSolver in heat module for example; and the other is **NewtonSolver** for nonlinear problems, see CNSolver and BD4Solver in PFM module for example. You can also implement a specific solver directly from the virtual interface of Solver.

In the Allen-Cahn case, we used explicit Euler scheme to create the fine solver. The core part of the **EulerSolver** contains two methods: evolve() and euler().

```
 // integrate the equation in the time slice
unsigned long EulerSolver::evolve() {
   // step0: set initial value
   soln = getSoln();        // pointer to the latest solution
   grid->guardcell(soln); // make sure that guard cells are at synchronization state

   unsigned long counter = 0;  // total iteration counter, for performance monitor
   for(int i=0; i<steps; i++){
      // step1 : set boundary condition
      grid->bc(soln);
      // step2 : call the forward Euler calculation
      euler();
      counter++;
      grid->guardcell(soln); // do not forget synchronize guard cells when any change occurs
   }
   // step3: return latest solution to the framework
   // For forward Euler scheme, nothing need to do
   return counter;
}
// first order forward Euler method
void EulerSolver::euler() {
   rhs();        // calcaluate RHS
   update();   // X_{n+1} = X_{n} + RHS (delta quantity)
}
  // specific calculation tasks
   void rhs() {
      if(ndim==3)   euler_rhs_ac_3d_(grid->nxyz, param.lamdaxyz, &nguard, b, soln, &dtk, &beta_);
       … …
   }
   void update() {
      if(ndim==3)   update_soln_3d_(grid->nxyz, &nguard, soln, b);
      … …
   }
```

The above code is intuitive, it also demonstrates how to use these basic functions of the **Grid** class. As mentioned above several times, the C++ code is responsible for flow control, all tedious calculation tasks are left to Fortran.

```
subroutine euler_rhs_ac_3d(nxyz, lamdaxyz, ng, b, soln, dtk, beta_)
implicit none
   integer, dimension(3) :: nxyz
   real   , dimension(3) :: lamdaxyz
   … …
   ix = nxyz(1)
   … …
   lamdax = lamdaxyz(1)
   … …
!$OMP PARALLEL &
!$OMP FIRSTPRIVATE(ix, jy, kz, lamdax, lamday, lamdaz, dtk, beta_)
!$OMP DO SCHEDULE(static) COLLAPSE(2)
   do k=1, kz
   do j=1,  jy
   do i=1,  ix
      ss = lamdax * ( soln(i-1, j,   k  ) - 2*soln(i,j,k) + soln(i+1, j,   k  ) ) … …
      b(i,j,k) = ss - dtk * soln(i,j,k) * ( soln(i,j,k) - 1.0 ) * ( soln(i,j,k) - beta_ )
   end do
   end do
   end do
!$OMP END DO
!$OMP END PARALLEL
end subroutine euler_rhs_ac_3d
```

For coarse solver, a bit more complicated implicit schemes are used, but the coding flow is similar. The **NewtonSolver** provides the integration template, and **CNSolver (Crank-Nicolson)** is no longer to implement its own evolve() method, like the **EulerSolver** does. The only thing CNSolver need to do is linking the Fortran code which performs the calculation defined by formula (3.6). The below pieces of code shows the process.

```
   virtual void stencil() {
     if(ndim==3) stencil_ac_3d_(grid->nxyz, param.lamdaxyz, &nguard, A, soln, &theta, &dtk, &beta_);
      … …
   }

!! setup the stencil struct matrix, A
subroutine stencil_ac_3d(nxyz, lamdaxyz, ng, A, soln, theta, dtk, beta_)
implicit none
   integer, dimension(3) :: nxyz
   real, dimension(1:7,  1-ng:nxyz(1)+ng, 1-ng:nxyz(2)+ng, 1-ng:nxyz(3)+ng ) :: A
   … …

!$OMP PARALLEL &
!$OMP FIRSTPRIVATE(ix, jy, kz, lamdax, lamday, lamdaz, theta, dtk, beta_)
!$OMP DO SCHEDULE(static) COLLAPSE(2)
   do k=1, kz
   do j=1, jy
   do i=1, ix
      A(1,i,j,k) = -theta*lamdax
      A(2,i,j,k) = -theta*lamdax
      … …
      A(7,i,j,k) = 1 + 2*theta*(lamdax + lamday + lamdaz) + theta*dtk * (  &
            (soln(i,j,k)-1.0) * ( soln(i,j,k) - beta_ )  &
         + (soln(i,j,k)    ) * ( soln(i,j,k) - beta_ )  &
         + (soln(i,j,k)    ) * ( soln(i,j,k) - 1.0    )  )
   end do
   end do
```

```
    end do
!$OMP END DO
!$OMP END PARALLEL
end subroutine stencil_ac_3d
```

The structure of **BD4Solver** is similar with the CNSolver, see source files in 'pfm' directory for detail. In order to **link** the C++ and Fortran, a bridge header file is necessary, see '**pfm.h**' for detail.

In addition, usually implicit schemes lead to solve a huge system of linear equations. At current, the framework provides parallel **BiCGStab** and **SOR** linear solvers. You can directly use them or develop new solvers. All the linear solver must be implement the **LS** interface, so that the framework can automatically create an instance of any solver at runtime. This design makes it easy to replace linear solver through parameter file without changing source code.

3.5 main program

Now it is the time to assemble the grid, solvers together in the main function. This step is very regular, there is not much room for customization. You can put the main entry in any .cpp files, but it is always a good choice to put the main function in a standalone file for a clear code.

```
int main(int argc, char* argv[]) {
    // 1. load global configuration (parameters) and init MPI
    Driver driver;
    driver.init(argc, argv);

    // 2. get init-ready system information
    // singleton pattern is used in order to access global configuration anywhere
    PinT* conf = PinT::instance();

    // 3. create and init the grid/mesh
    Grid *g = new PFMGrid(conf);
    g->init();

    // 4. choose the proper coarse/fine solver
    Solver *G, *F;
    G = new BD4Solver(conf, g, false);
    F = new EulerSolver(conf, g, true);

    // for debugging only, this is also a trick to quit the running program safely at any time
    // driver.Abort("高次元テスト for 3D Phase Field Model");   // don't call it at a normal running

    // 5. run the parareal algorithm (in space-time parallel)
    driver.evolve(g, G, F);

    // 6. output result to disk and for post-processing
    // For large scale performance test, it is best to close the output
    g->output_local(g->u_end, true); // ASCII format
    g->output_global_h5("pfm");      // HDF5 format

    // 7. quit MPI
    driver.finalize();
    return 0;
}
```

The above main function is also a standard template for any problem based on the framework, see 'pfm_main.cpp' for detail. For a new problem, you just need to replace the name of grid and solvers in the **'new'** clause. Note that only one main() function is allowed in one program, generally each module has its own main() function, in order to avoid conflicts, we create a preprocessor option for

each module. For example in pfm_main.cpp :

```
#ifdef _TEST_PFM_
int main(int argc, char* argv[]) {…}
#endif
```

when compiling use '-D _TEST_PFM_' to tell the compiler(g++/icc) to include the target module.

3.6 parameter file

Parameter file consists of several sections, and the each section includes several key-value pairs. You can rename the parameter file as you want, the default parameter file is **pint.ini** within the same directory of the executable file. **';'** is used to make comments, the following text of ';' will be ignored by the parser. **'[ ]'** is used to denote a section. Three types of value are supported: integer, float and string. For 1D and 2D case, parameters of useless dimensions will be ignored automatically. The usage of each parameter is well explained at both the wiki and the sample parameter file in detail.

```
[domain]          ; time-space domain of the problem
ndim =  1         ; dimension number : 1,2,3.
Tspan = 0.005     ; time domain
Nt = 100000       ; the time steps of the whole time domain serially if using fine solver
Nx = 128          ; the whole grid size, all discrete cell numbers.
Ny = 128
Nz = 128
… …
[parareal]        ; space-time parallel
tsnum    = 10     ; the number of time partitions(slices), parallel processes along time domain
spnumx = 2        ; the number of space partitions (CPU cores) in one time slice, x direction
spnumy = 2        ; y direction
spnumz = 3        ; z direction
… …
[pfm]               ; problem-specific configuration
theta     =  0.5    ; 0.5: Crank-Nicolson; 0: Ex.Euler; 1: Im.Euler
ac_kval  = 16000
ac_beta = -0.128
… …
```

Note that If any **inconsistency** between parameters and runtime environment occurs, the program will forcibly break, and print the error information. For large scale running, the root error is not easy to be found, you can try to search the two keyword 'ERROR', 'WARN' (case sensitive).

4. Extensibility for non-standard parareal algorithm

In a classical parareal implementation, solutions of only one previous time step need to be transferred to the next time slice**,** such as Crank-Nicolson method. But some time integration schemes need more previous solutions, such as Backward Differentiation Formula series, for example **BD4Solver** used in 'pfm' module, it need solutions from 4 previous time steps, that is previous solutions need to be **preserved** for a while. The original standard template can't provide a general support for this condition, in order to make the critical Driver class general and independent, we introduced several extensible interfaces into the Solver class, the basic class for all coarse/fine solvers in the framework. Though we tried our best to make the design clear, it still made the logic of the **evolve()** method of Driver class a bit more complicated than the classical description of parareal algorithm.

To support like BD4 schemes, we introduced several dedicated access(getter) methods into Solver class to avoid accessing important variables directly, this style also makes the code more self-explanatory. All the virtual methods can support the standard parareal flow by default, specific implementations have to be provided by the subclass when necessary, such as BD4Solver. Just through these interfaces (virtual methods) of Solver, the **Driver** class can provide a general skeleton of parareal algorithm.

```
// return the solutions of previous iteration(K^par-1) of coarse solver
virtual double* prev_solns()   // default is grid->u_cprev

// return the current solutions
 virtual double* curr_solns()  // default is grid->u_c for coarse solver or grid->u_f for fine solver

 // preserve the solution of coarse solver
 virtual void backup_prevs()  // preserving solutions of previous iteration for coarse solver

 // return the size of solutions need to be sent or received
 virtual size_t solnsize()   // default is grid->size, 4 times of the default size for BD4Solver

 // return the pointer of variables to be sent
 virtual double* sendslns() // default is grid->u_end

 // return the pointer of variables to be received
 virtual double* recvslns()  // default is grid->u_start

 /**
  * pack and unpack the send/recv data, nothing need to do in the standard parareal,
  * for some particular coarse solver, like BD4,
  * you must pack the preserved four previous solutions into send buffer of MPI manually.
  * It is not an easy task to make the pair functions general for all the children solvers
  * in the current version, concrete class must provide its own implementation as needed
  */
 virtual void pack()
 virtual void unpack()

 // write back the final solution (u_end) from correction item (see Driver.pint_sum)
 virtual void update_uend()
```

NOTE: in order to make the framework adaptable to more time integration schemes, the two variables **u_start** and **u_end** are strictly limited to store the solution at the start or end time step of the current time slice separately (see section 2.1). So before staring integration at a time slice, you must make sure the correct init value is already copied into the u_start, and after applying correction operation of parareal , don't forget to update u_end with the latest solution. For the common parareal, the default implementation is OK, but for a non-standard case, like BD4, you must do this by yourself. See source code of the BD4Solver in 'pfm' module for detail.

In order to tackle some schemes like BD4 automatically, a new parameter '**num_std**' is introduced, the parameter indicates the mount of data transferred between neighbor time slices. The default value is **1** (ONE) for the common parareal, and for BD4, the value should be **4** (four), it means 4 times of the common case. Besides the above particular process of BD4 coarse solver, fine solver must also provide similar data structure for preserving necessary previous solutions used in correction phase of parareal, see EulerSolver for details.

5. Building and Running

The Makefile is simple, and you can easily adapt it to any UNIX-like system. In order to manage new modules easily, we used several option flags, for the following PMLib example:

```
_PMLib_ =1
ifdef _PMLib_
PMLIB_HOME   = /Users/bamboo/Libs/PMlib
PMLIB_INC      = -I${PMLIB_HOME}/include
LDFLAGS_PM   = -L${PMLIB_HOME}/lib -lPMmpi
OPTFLAGS_PM= -fopenmp -D_PMLib_
```

endif

If you don't want to use PMLib to collect the performance data, you can comment out the '_PMLib_=1' using the '#' character. If you add a new physical module to the code base, in order to make each module independently, we suggest you following the makefile style, that is creating a dedicated segment for the new module, see the Makefile for detail.

Note that, the framework only support double precision at the current version, when compiling Fortran, make sure the default **real** type is double precision. Each compiler has its own style to achieve this. For GFortran, the option is '-fdefault-real-8', for Intel's IFort, it is '-r8', for Fujitsu' compiler, it is 'cdRR8'.

The executable file can receive two optional command line parameters.

- 1st param: the relative path of global parameter file
- 2nd param: job id, in order to distinguish the performance report for multiple runs.

To run it on PC, execute the below shell scripts in a terminal.

```
> mpirun -host localhost -np 4  ./pfm_alpha.exe  src/pfm/pfm.ini  bd4
```

To run it on HPC, usually job script is necessary, for example in ITO supercomputer, please see the ito_pfm.sh in 'test' directory.

6. Next…

If the current solver templates can not satisfy your problem, you can develop new solvers easily and quickly based on the basic space-time parallel functionality provided by the framework. The **Grid** can provide the space parallel support, the **Driver** provides the skeleton of the parareal algorithm. As long as the new solver follows the interface of the virtual class '**Solver**', the **Driver** can drive it in space-time parallel.

We have just completed a large scale evaluation for parareal algorithm on Allen-Cahn equation based on the sample code in the 'pfm' directory. We plan to reduce the iteration numbers of linear solver in Newton solver by pipelining the Newton iterations in time in the coming future.