Overview of the **MicroSim** project and software stack

Abhik Choudhury

November 4, 2021

1 Introduction

MicroSim short for **Micro**structural **Sim**ulator is a software stack consisting of phase-field codes based on differing discretization strategies (finite difference, finite volume and Fast Fourier transform) along with implementation on varied high-performance computing infrastructure such as MPI(CPU), CUDA(GPU) and OpenCl(GPU). The motivation for the creation of this software stack is to provide the academic and industrial user base a open-source software framework that can be utilized readily or with minimal modifications for routine problems relating to phase transformations during materials processing. The present release is the first among many of our software stack that will be presently be able to address phase transformations such as solidification and precipitation.

2 Generic structure of the stack

The software stack is described in Fig.1. A brief description of the modules follows:

- **Grand-potential based solver**: This is a multi-phase multicomponent phase-field solver based on a regular-grid finite-difference discretization, with a simple Euler forward time marching scheme. It is based on the phase-field model presented in *Phys. Rev. E 85, 021602 (2012)*. Two versions of the solver are available, one which is a serial code that runs on the CPU and the second that is parallelized using a simple block domain-decomposition using MPI. Presently, these solvers are discretized in 2D and future releases will contain more features as well as domain discretization in 3D. A dedicated module related to its usage can be found in the solver folders.
- **Kim Kim Suzuki(KKS) based model (OpenCl)**: This is a multi-component based solver that is based on the Kim Kim Suzuki model which uses a finite difference discretization with explicit time stepping. The code utilizes the OpenCl based framework for utilizing both CPU and GPU infrastructure. The solver allows for parsing in a .tdb file for reading in the thermodynamic information directly into the phase-field formulation. Presently the code has been implemented for 2D and future releases will

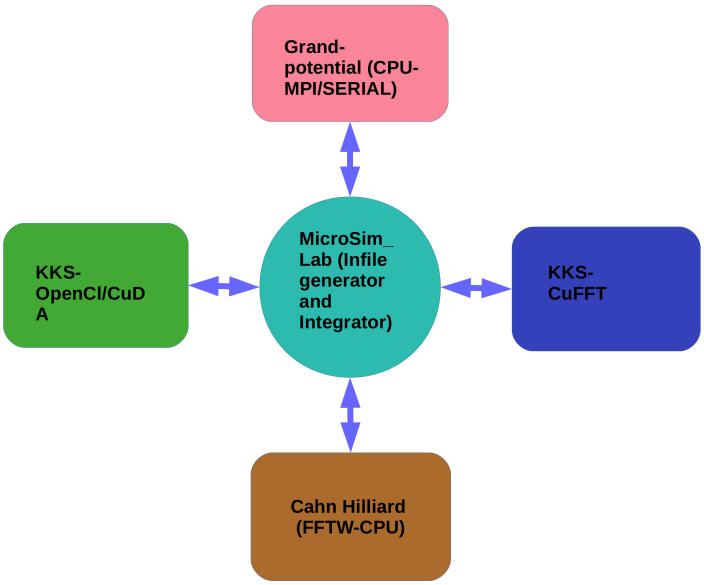


Figure 1: Schematic the different solvers and their integration using a python integrator MicroSim_lab.

contain more features as well as the more generic 3D solver. A dedicated module related to its usage can be found in the solver folders.

- **Kim Kim Suzuki(KKS) based model (CUDA)**: This is a binary alloy solver that is based on the Kim Kim Suzuki model and utilizes the Fourier transform for the solution of PDEs by leveraging the CuFFT libraries as part of CUDA. The solver is meant for utilizing the NVIDIA GPU graphic cards particularly the Tesla-P100 and later cards. Because of the Fourier transform discretization this solver module has by default periodic boundary conditions. The code can be utilized for both 2D and 3D geometries and is presently written for utilizing single GPU cards. Later versions of the code will be extended for the treatment of multiple phases and components as well as parallelization on multiple GPUs. A dedicated module related to its usage can be found in the solver folders.

- **Cahn-Hilliard model (FFTW)**: This is a binary alloy solver that is based on the Cahn-Hilliard formulation and utilizes the Fourier transform based solution of PDEs by leveraging the FFTW3 libraries. The solver runs on single CPUs and it is meant for quick solutions to problems in smaller domains. Because of the Fourier transform discretization this solver module has by default periodic boundary conditions. The present solver is programmed for 2D domains that will be generalized for 3D domains in future releases. Along with this, the solver will also be extended for simulating multicomponent and multi-phase systems. A dedicated module related to its usage can be found in the solver folders.

- **MicroSim_lab**: MicroSim_lab is a Python interface that allows the user to create the input files and choose the solver of his choice for the execution. It integrates the preceding modules that are a result of in-house phase-field code development. The interface also allows for a direct call to the visualization software Paraview enabling the user to view the results of the simulation.

- **OpenFoam based solvers**: Along with the preceding softwares that are a result of home-grown codes that have been integrated over the years, the software stack will also contain the Finite Volume based modules that are utilize the OpenFoam multiphysics platform. These will utilize the already developed interfaces as part of OpenFoam and the modules that will be shared as part of this software stack will contain an independent documentation of the module and its usage. Presently, a single-phase binary alloy solidification module that is based on the Grand-potential based formulation and utilizes adaptive mesh refinement will be part of this release. Future release will contain modules for two-phase eutectic growth. Since the modules utilize the OpenFoam framework, they are generic solvers for both 2D and 3D domains.

3 Process flow

In the following we describe briefly the workflow for getting the user ready with his/her simulation, which is schematically described in Fig.2. The MicroSim_lab interface allows one to gain experience in using the software by enabling navigation through the parameter set required for the solvers for their execution. The interface can be utilized for the generation of the "Infile" that details the important parameters that are needed for a particular solver along with the boundary conditions as well as the information of the initialization of the domain. The experienced user will be able to modify the Input and Filling files manually while the early users are encouraged to utilize the interface to gain familiarity with the different solvers and their input parameters. The interface can also be utilized as quick check for the parameters in the Input files as it enables one to run and visualize the simulation results in a single workflow. For the experienced user who makes a change to the original codes can also make use of the interface for compiling his codes readily.

4 System requirements

The minimum system requirements for compiling and executing the codes are mentioned in each of the solver manuals and is elaborated here.

- **Grand-potential based solver(SERIAL)**: Linux, gcc/icc compiler

- **Grand-potential based solver(MPI)**: Linux, h5pcc compiler

- **Kim Kim Suzuki(KKS) based model (OpenCl)**: Linux, OpenCl compiler, pycalphad and related dependencies. Sympy, pandas

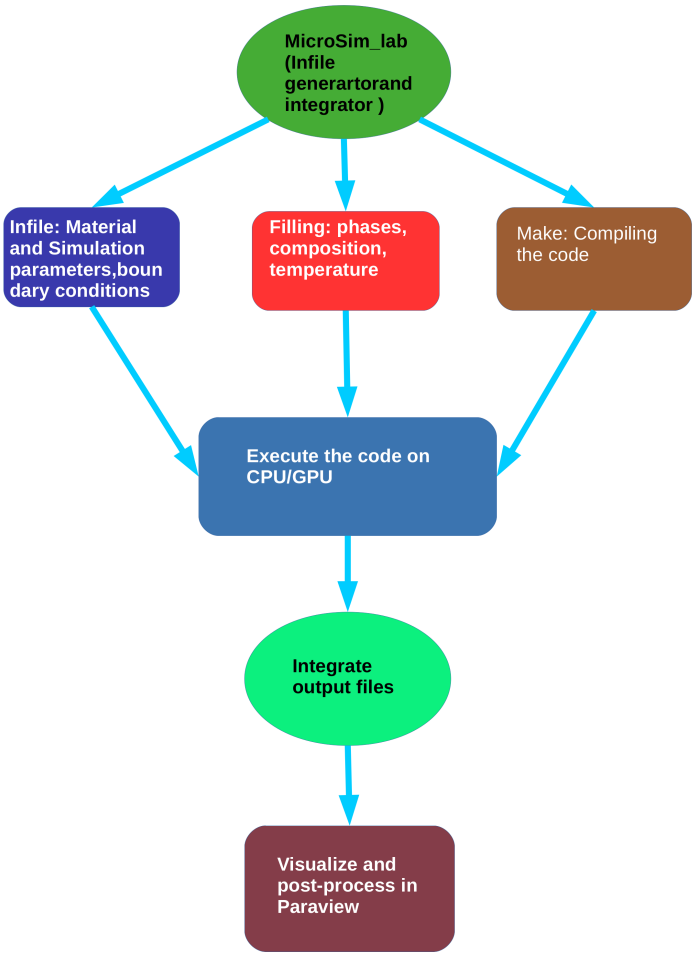


Figure 2: A typical workflow for executing a simulation using the software stack

- **Kim Kim Suzuki(KKS) based model (CUDA)**: Nvidia Tesla P100/V100 cards, CUDA version 11, CUFFT and CUDA-CUB libraries, nvcc version 11.2
- **Cahn-Hilliard model (FFTW)**: Linux, gcc/icc compiler, FFTW3 libraries
- **OpenFoam based solvers**: Linux, OpenFoam v6, dynamicInterfaceFvMesh Library, gcc and OpenFoam related dependencies

5 Visualization and File types

Paraview is the default visualization software for the simulation results as well as associated post-processing. All the output files are written in a "DATA" folder in the execution directory. The files are written in .vtk format, ASCII/BINARY depending on the user specification in the Infile for the serial or single GPU codes. For the MPI-based solvers (Grand-potential based solver(MPI)) the file formats are .vtk(ASCII) or .h5. For the MPI based solvers the .h5 files are reconstructed into .xml format for viewing in paraview, while a built-in executable ./reconstruct is utilized for reconstruction of .vtk(ASCII) files. Detailed description of the execution commands may be found in the individual manuals present in the solver folders.

6 Infile

The input parameters required for the solver are derived from an infile. This contains information about the domain geometry, the thermodynamic functions(free energy), material properties such as the interfacial energies and their anisotropies as well as boundary conditions. It could also contain special flags related to the running of the solver. The following is a basic description of the keys in the infile. **Each key must end with a semicolon**. Additionally, all lines beginning with "#" will be treated as comments in the infile.

```
##Geometrical dimensions of the simulation domain
DIMENSION = 2;
MESH_X = 50;
MESH_Y = 100;
MESH_Z = 1;
##Discretization, space and time
DELTA_X = 2.0;
DELTA_Y = 2.0;
DELTA_Z = 2.0;
DELTA_t = 0.08;
##Number of phases and composition
NUMPHASES = 2;
NUMCOMPONENTS = 2;
#Running and saving information
NTIMESTEPS = 10000;
NSMOOTH = 10;
SAVET = 1000;
```

```
## Component and Phase names
# COMPONENTS = {Al,Cu,B};
COMPONENTS = {Al, Cu};
PHASES = {alpha, beta};
##Material properties
##GAMMA={12, 13, 14, 23, 24...}
GAMMA = {1.0};
# Diffusivity = {Diagonal:0/1, phase, 11,22,33, 12, 13, 23...};
DIFFUSIVITY = {1, 0, 0};
DIFFUSIVITY = {1, 1, 1};
##Gas constant and molar volume
R = 1.0;
V = 1.0;
##Elasticity related parameters
EIGEN_STRAIN = {0, 0.01, 0.01, 0.0, 0.0, 0.0, 0.0};
EIGEN_STRAIN = {1, 0.01, 0.01, 0.0, 0.0, 0.0, 0.0};
VOIGT_ISOTROPIC = {0, 270, 187.5, 125.0};
VOIGT_ISOTROPIC = {1, 270, 187.5, 125.0};
#VOIGT_CUBIC = {phase, c11, c12, c44};
#VOIGT_TETRAGONAL = {phase, c11, c12, c13, c33, c44, c66};
##Boundary conditions
#0: Free, 1: Neumann, 2: Dirichlet, 3: Periodic, 4: Complex
#Boundary = {phase, X+, X-, Y+, Y-, Z+, Z-}
BOUNDARY = {phi, 1, 1, 1, 1, 0, 0};
BOUNDARY = {mu, 1, 1, 1, 1, 0, 0};
BOUNDARY = {c, 1, 1, 1, 1, 0, 0};
BOUNDARY = {T, 1, 1, 1, 1, 0, 0};
# Boundary = {phi, 1, 1, 0};
# Boundary = {"u", 3, 3, 2, 2};
#Boundary_value = {Value X+, Value X-, Value Y+, Value Y-, Value
    ↪ Z+, Value Z-}
BOUNDARY_VALUE = {phi, 0, 0, 0, 0, 0, 0};
BOUNDARY_VALUE = {mu, 0, 0, 0, 0, 0, 0};
BOUNDARY_VALUE = {c, 0, 0, 0, 0, 0, 0};
BOUNDARY_VALUE = {T, 0, 0, 0, 0, 0, 0};
##Type of simulation
ISOTHERMAL = 1;
BINARY = 1;
#TERNARY
DILUTE = 0;
T = 0.97;
##FILEWRITING and OUTPUTTING TO SCREEN
## WRITEFORMAT ASCII/BINARY/HDF5(Only for MPI)
##TRACK_PROGRESS: interval of writing out the progress of the
    ↪ simulation to stdout.
WRITEFORMAT = BINARY;
WRITEHDF5 = 1;
TRACK_PROGRESS = 10;
##Model-specific parameters: Grand-potential model
##Phase-field parameters; epsilon:interface width; it is not the
```

```
    ↪ gradient energy coefficient
epsilon = 8.0;
tau = 1.31;
Tau = {0.28};
##Anisotropy functions
##Anisotropy mode, FUNCTION_ANISOTROPY=0 is isotropic
Function_anisotropy = 1;
Anisotropy_type = 4;
dab = {0.02};
#Rotation_matrix = {0, 1, Euler_x(ang), Euler_y(ang), Euler_z(ang
    ↪ )};
Rotation_matrix = {0, 1, 0, 0, 0};
##Potential function
Function_W = 1;
Gamma_abc = {};
#Writing of composition fields along with the chemical potential
    ↪ fields
Writecomposition = 0;
#Noise
Noise_phasefield = 0;
Amp_Noise_Phase = 0.001;
##Temperature
Equilibrium_temperature = 1.0;
Filling_temperature = 1.0;
##Function_F
Function_F = 1;
A = {0, 1};
A = {1, 1};
ceq = {0, 0, 0.78125};
ceq = {0, 1, 0.5};
ceq = {1, 1, 0.5};
ceq = {1, 0, 0.5};
cfill = {0, 0, 0.78125};
cfill = {0, 1, 0.5};
cfill = {1, 1, 0.5};
cfill = {1, 0, 0.5};
slopes = {0, 0, 0.45};
slopes = {0, 1, 0.45};
slopes = {1, 0, 0.45};
slopes = {1, 1, 0.45};
```

6.1 Simulation geometry, spatial and temporal discretization

```
##Geometrical dimensions of the simulation domain
DIMENSION = 2;
MESH_X = 100;
MESH_Y = 100;
```

```
MESH_Z = 1;
```

- DIMENSION: This can take values 2,3 for 2D and 3D simulations respectively. This is a required key in the solver and not mentioning this key might lead to unexpected results
- MESH_X,MESH_Y,MESH_Z: These are the number of grid points in the domain(and not the physical lengths) in the respective X, Y, Z directions. When DIMENSION is 2, the value of MESH_Z will redundant and will be taken as 1.

```
##Discretization, space and time
DELTA_X = 2.0;
DELTA_Y = 2.0;
DELTA_Z = 2.0;
DELTA_t = 0.08;
```

- The values DELTA_X, DELTA_Y, DELTA_Z correspond to the grid resolution in the X, Y, Z directions respectively. Similarly, DELTA_t corresponds to the temporal discretization(time-step).

6.2 Phases and Components information

```
##Number of phases and composition
NUMPHASES = 2;
NUMCOMPONENTS = 2;
```

- The keys are self-explanatory. NUMPHASES corresponds to the number of phases in the domain, while NUMCOMPONENTS corresponds to the number of components(2, for binary, 3 for ternary etc.)
- These keys are absolutely necessary, please fill carefully for successful running of your codes.

```
COMPONENTS = {Al, Cu};
PHASES = {alpha, beta};
```

- COMPONENTS refers to the tuple that consists of the names of the components. The names are separated by commas and the entire tuple needs to placed within {} followed by a semicolon.
- Similarly, PHASES refers to the names of the phases in the domain.

6.3 Boundary conditions

```
##0:FREE, 1: Neumann, 2: Dirichlet, 3: Periodic, 4: Complex
##BOUNDARY = {TYPE, X_LEFT, X_RIGHT, Y_FRONT, Y_BACK, Z_TOP,
    ↪ Z_BOTTOM}
BOUNDARY = {phi, 1, 1, 1, 1, 0, 0};
BOUNDARY = {mu, 1, 1, 1, 1, 0, 0};
BOUNDARY = {c, 1, 1, 1, 1, 0, 0};
BOUNDARY = {T, 1, 1, 1, 1, 0, 0};
```

- Any of the solvers in repository will consist of the following scalar default types. Type "phi" will represent the phase-field order parameters whose number is specified by the key, "NUMPHASES". Depending on the solver whether it is the grand-potential based solver, where "mu" will be the independent variable or if it is the Cahn-Hilliard or Kim-Kim-Suzuki based solvers, where "c" is the independent variable, Type "mu" or "c" will refer to the components in the key "COMPONENTS". Similarly, type "T" will refer to the temperature field. The BOUNDARY key will refer to the boundary condition for the respective Type of field. The following numbers in a given tuple refer to the boundary at the respective (X_LEFT, X_RIGHT, Y_FRONT, Y_BACK, Z_TOP, Z_BOTTOM) boundaries, that refer to the X, Y, Z boundaries in order. The boundary condition is specified by numbers, 0:FREE, 1: Neumann, 2: Dirichlet, 3: Periodic, 4: Complex, where for the DIRICHLET boundary condition the respective boundary can also take in a specified value which can be specified in the following key: BOUNDARY_VALUE. If the key is not present, the default remains the one that is initialized at the start of the simulation and is left unchanged. Further, if for any direction, for eg: X, one of the extremeties is specified as a PERIODIC boundary, then the other X-boundary will also be initialized as PERIODIC irrespective of the entry in the tuple.

```
#BOUNDARY_VALUE = {Type, Value X+, Value X-, Value Y+, Value Y-,
    ↪ Value Z+, Value Z-}
BOUNDARY_VALUE = {phi, 0, 0, 0, 0, 0, 0};
BOUNDARY_VALUE = {mu, 0, 0, 0, 0, 0, 0};
BOUNDARY_VALUE = {c, 0, 0, 0, 0, 0, 0};
BOUNDARY_VALUE = {T, 0, 0, 0, 0, 0, 0};
```

- This key corresponds to a specific value that needs to be specified on a boundary which will be held constant during the duration of the simulation. The definition of this key should follow the BOUNDARY specification and follows the same type of definition, except here the tuple Value X_LEFT, Value X_RIGHT, Value Y_FRONT, Value Y_BACK, Value Z_TOP, Value Z_BOTTOM, refer to values on the respective boundaries. The values in this tuple will only be utilized if the respective boundary condition on that boundary is DIRICHLET. By default, the value will be treated as the same as the one that is initialised at the start of the simulation, which will be utilized if this key is not present.

6.4 Number of iterations, smoothing time-steps and writing interval

```
#Running and saving information
NTIMESTEPS = 10000;
NSMOOTH = 10;
SAVET = 1000;
```

- NTIMESTEPS: Total number of iterations that you wish the solver to run. This is not the total time
- NSMOOTH: The number of pre-conditioning steps for smoothening sharp phase-field profiles that are initialised at the start of the simulation
- SAVET: Writing interval, i.e, frequency of writing files of the respective fields

6.5 Material parameters

```
##Gas constant and molar volume
R = 1.0;
V = 1.0;
```

- The values "R" and "V" will refer to the gas constant and the molar volume respectively

```
#DIFFUSIVITY={Diagonal:0/1, phase, 11,22,33..(K-1) diagonal
    ↪ elements, 12, 13, 23...(rest of the elements; rowwise)}
DIFFUSIVITY = {1, 0, 1};
```

- The DIFFUSIVITY key refers to the inter-diffusivity matrix which has the tuple in the following form. The first element can taken in values 0/1, "1" refers to as a diagonal matrix and "0" is a full matrix.
- The following element refers to the phase number referring to the phases in the list PHASES. This can take values from 0 to NUMPHASES-1.
- The following elements are the values in the inter-diffusivity matrix. The first elements are the diagonal terms in the matrix, while the following elements correspond rowwise to the off-diagonal terms in the diffusivity matrix.
- If the first element in the tuple is "1", i.e. the matrix is diagonal irrespective of the number of entries in the tuple only the entries corresponding to the diagonal elements in the matrix will be read in.

```
##GAMMA = {12, 13, 14, 23, 24...}
GAMMA = {1.0};
```

- The GAMMA key refers to the interfacial energy $\gamma_{\alpha\beta}$ between the phases $\alpha\beta$. The elements in tuple correspond to all combination of phases forming the interfaces from the list of phases in PHASES numbered from 0 to NUMPHASES-1.
- The elements are numbered in the order $12, 13, 14, 23, 24 \ldots N(N-1)$, $N = NUMPHASES$ where "12" corresponds to the value of the interfacial energy between phase 1 and 2; γ_{12}.
- In the tuple only combinations $\alpha\beta$ are included where $\alpha < \beta$ as the value of the interfacial energy of the $\alpha\beta$ interface is $\gamma_{\alpha\beta}$ which is the same as $\gamma_{\beta\alpha}$.
- Therefore, the total number of elements in the tuple is $\frac{N(N-1)}{2}$.

```
#EIGEN_STRAIN = {phase, exx, eyy, ezz, eyz, exz, exy};
EIGEN_STRAIN = {0, 0.01, 0.01, 0.0, 0.0, 0.0, 0.0};
EIGEN_STRAIN = {1, 0.01, 0.01, 0.0, 0.0, 0.0, 0.0};
```

- The EIGEN_STRAIN key refers to the eigen-strain tensor in a given phase. The information about the elements of the eigen-strain are derived from the elements in the tuple.
- The first element refers to the phase number in the list PHASES and can take in values from 0 to NUMPHASES-1.
- The following elements in tuple are mapped to the eigen-strain matrix in the following order $exx, eyy, ezz, eyz, exz, exy$.
- This is an important key required for problems where there are coherent interfaces and the phase transformation is coupled with the stress distribution arising due to coherency strains at the interface.

```
#VOIGT_ISOTROPIC = {phase, c11, c12, c44};
VOIGT_ISOTROPIC = {0, 270, 187.5, 125.0};
VOIGT_ISOTROPIC = {1, 270, 187.5, 125.0};
```

- VOIGT_ISOTROPIC is the key that refers to the elastic stiffness properties in the Voigt notation for an isotropic material.
- The first element in the tuple refers to the phase which will be a number from 0 to NUMPHASES-1.
- The following elements are the values $C11, C12, C44$ in order.

```
#VOIGT_CUBIC = {phase, c11, c12, c44};
VOIGT_CUBIC = {0, 270, 187.5, 125.0};
VOIGT_CUBIC = {1, 270, 187.5, 125.0};
```

- VOIGT_CUBIC is the key that refers to the elastic stiffness properties in the Voigt notation for a cubic material.
- The first element in the tuple refers to the phase which will be a number from 0 to NUMPHASES-1
- The following elements are the values $C11, C12, C44$ in order

```
#VOIGT_TETRAGONAL = {phase, c11, c12, c13, c33, c44, c66};
```

- VOIGT_TETRAGONAL is the key that refers to the elastic stiffness properties in the Voigt notation for a tetragonal material.
- The first element in the tuple refers to the phase which will be a number from 0 to NUMPHASES-1.
- The following elements are the values $C11, C12, C13, C33, C44, C66$ in order.

```
BINARY = 1;
DILUTE = 0;
```

- The keys "BINARY", "TERNARY", "DILUTE" are special flags to the solver allowing for simpler routines in the update of the chemical potential or composition fields.

```
##FILEWRITING and OUTPUTTING TO SCREEN
## WRITEFORMAT ASCII/BINARY
##TRACK_PROGRESS: interval of writing out the progress of the
    ↪ simulation to stdout.
WRITEFORMAT = ASCII;
WRITEHDF5 = 0/1;
TRACK_PROGRESS = 10;
```

- The key WRITEFORMAT mentions the type of the output files to be written. The possible values are ASCII or BINARY that are self-explanatory(BINARY is not activated for MPI because of incompatibility with the h5pcc compiler). When running in parallelized mode with MPI, there is a third possibility for writing files using the key WRITEHDF5 key. For MPI runs, if the WRITEHDF5 key is non-zero, then the value of the WRITEFORMAT key is ignored and files in hdf5 format are written for each time-step (files with extension .h5). Since, the hdf5 file library is linked to the solver by default, the solver needs to be compiled with h5pcc for parallel runs instead of mpicc. This is irrespective of the value of the key WRITEHDF5.
- For the case when the type chosen is BINARY, the format is BIG-ENDIAN such that it matches the BINARY type required for PARAVIEW. BINARY is not activated for MPI solvers.

- For MPI runs, in the event WRITEHDF5=0, the WRITEFORMAT key decides the format of the output file, where each processor writes its own file for each time given by the key saveT. In order to view the consolidated file, the tool (./reconstruct) needs to executed in the folder just outside the /DATA folder that is created. The following command needs to be executed for reconstructing,

```
./reconstruct name_of_infile name_of_output_file
    ↪ number_of_workers start_time end_time
```

- For the case when WRITEHDF5=1, all processors write collectively into a single .h5 file. For viewing the file in paraview, it needs to be put in .xml format, which can be performed using the following command, that needs to be executed from just outside the /DATA folder that is created using the run,

```
./write_xdmf name_of_infile name_of_output_file
    ↪ number_of_workers start_time end_time
```

- TRACK_PROGRESS is a key that will inform the solver about the frequency with which the progress of the simulation will be written to stdout.
- The number can be anything other than 0.

6.6 Model specific parameters

The rest of the parameters in Infile are model specific parameters that are different for the separate solvers and their description will be found in the manuals contained in the separate solver folders.

7 Filling

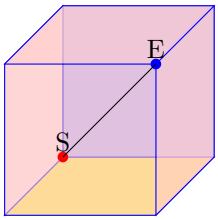
The instructions about filling the domain with phases present in the file "filling.in", which is detailed in

```
#FILLCUBE = {0, 10, 10, 0, 20, 20, 0};
#FILLCUBE = {0, 10, 10, 0, 20, 20, 0};
##FILLCYLINDER = {phase, x_centre, y_centre, z_start, z_end,
    ↪ radius}
FILLCYLINDER = {0, 25, 50, 0, 0, 20};
#FILLCUBE = {0, 0, 0, 0, 16, 10, 0};
#FILLCUBE = {1, 16, 0, 0, 33, 10,0};
#FILLCUBE = {2, 33, 0, 0, 50, 10,0};
###FILLELLIPSE = {phase, x_center, y_center, major_axis,
    ↪ eccentricity, rotation_angle_deg}
#FILLELLIPSE = {0, 50, 50, 0, 10, 0.1, 10};
#FILLSPHERE = {0, 50, 50, 0, 10};
```

. In the following, the possible filling routines are described. The filling instructions may be repeated, that allows for the filling of complicated shapes with overlapping.

```
#FILLCUBE = {phase, x_start, y_start, z_start, x_end, y_end,
    ↪ z_end}
FILLCUBE = {0, 10, 10, 0, 20, 20, 0};
```

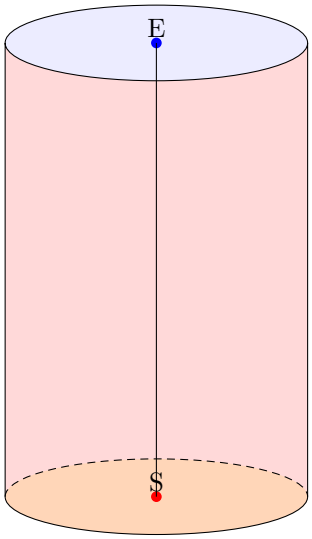
- The "FILLCUBE" key can be utilized for initializing a cube.
- The input about the dimensions of the cube are given in the form of a tuple FILLCUBE = {phase, x_start, y_start, z_start, x_end, y_end, z_end}, where the first element is a number between 0 and NUMPHASES-1, which indicates a phase in the list PHASES.
- The following elements are the co-ordinates(x,y,z) of the end-points of the diagonal in the cube. In the following figure, (x_start, y_start, z_start) corresponds to point S and (x_end, y_end, z_end) corresponds to point E in the cube.
- This operation leads to the filling of the phase given by the number "phase" in the shape of a cube. All other phases are initialised as zero in this region while the rest of the domain is filled with the last phase NUMPHASES-1.
- For 2D object, the z dimensions should be initialised as 0. This would initialize a rectangle.




```
##FILLCYLINDER = {phase, x_centre, y_centre, z_start, z_end,
    ↪ radius}
FILLCYLINDER = {0, 50, 50, 0, 0, 20};
```

- The "FILLCYLINDER" key allows one to initialize a phase in the shape of a cylinder.
- The dimensions are provided in the form of a tuple {phase, x_centre, y_centre, z_start, z_end, radius}.
- The first element "phase" is a number between 0 and NUMPHASES-1, which indicates a phase in the list PHASES.
- The next elements (x_centre, y_centre, z_start) refer to the point S in the following figure.
- The next dimension is z_end which sets the length of the cylinder, i.e the co-ordinate of point E in the following figure is (x_end, y_end, z_end).

- Finally, the radius of the cylinder is specified by the final element in the tuple.
- This operation leads to the filling of the phase given by the number "phase" in the shape of a cylinder. All other phases are initialised as zero in this region while the rest of the domain is filled with the last phase NUMPHASES-1.
- For 2D object, the z dimensions should be initialised as 0. This would initialize a circle.
- Presently, the FILLCYLINDER is designed to fill a cylinder oriented in the Z-direction.




```
#FILLSPHERE = {phase, x_center, y_center, z_center, radius};
FILLSPHERE = {0, 50, 50, 0, 10};
```

- The "FILLSPHERE" key allows the filling of a phase in the form of a sphere.
- The dimensions are provided in the form of a tuple {phase, x_center, y_center, z_center, radius} where the first element "phase" is a number from 0 to NUMPHASES-1, which corresponds to a phase in the list PHASES.
- The center of the sphere is provided in the next three elements of the tuple (x_center, y_center, z_center).
- The last element is the radius of the sphere.
- This operation leads to the filling of the phase given by the number "phase" in the shape of a sphere. All other phases are initialised as zero in this region while the rest of the domain is filled with the last phase NUMPHASES-1.
- This object can only be used when the DIMENSION = 3 in the infile.

8 Developers and Credits

Following is the list of the developers

- Tanmay Dutta (IISc, Bangalore) (OpenFoam solvers and documentation)
- Ajay Sagar(IISc) (Python wrapper and infile generator)
- Dasari Mohan, M.P. Gururajan, Gandham Phanikumar (IIT Bombay, Madras) (KKS OpenCl and FFTW codes)
- Saurav Shenoy, Pankaj and Saswata Bhattacharyya (IIT Hyderabad) (KKS Nvidia-CUDA)
- Abhik Choudhury (IISc, Bangalore) (Grand-potential based solvers)

Acknowledgments for contributions from PhD/Master/UG students

- Sumeet Rajesh Khanna (PhD, IISc)
- Tushar Jogi (PhD, IIT Hyderabad)
- Gerald Tennyson (PhD, IIT Madras)
- Ravi Kumar Singh (M.Tech, IISc)
- Umate Kartik (BS, IISc)