
ppiclF Documentation

Release v1.0.0

David Zwick

May 04, 2019

CONTENTS

1	Capabilities	3
2	Quickstart	5
2.1	Quickstart	5
2.1.1	Dependencies	5
2.1.2	Build	5
2.1.3	Link	6
3	Algorithms	7
3.1	Algorithms	7
3.1.1	Particle Storage	7
3.1.2	Overlap Mesh	9
3.1.3	Ghost Particles	11
3.1.4	Performance	12
4	User Interface	15
4.1	User Interface	15
4.1.1	The H-File	15
4.1.2	The F-File	17
4.1.3	External Interface	19
4.1.4	File Output	22
5	Examples	25
5.1	Examples	25
5.1.1	Stokes 2D	25
5.1.2	DEM Packing 3D	27
5.1.3	Nek5000 Case	30
6	Contributing	35
6.1	Contributing	35
6.1.1	Questions	35
6.1.2	Workflow	35
6.1.3	Issues	35
7	Acknowledgements	37
7.1	Acknowledgements	37
7.1.1	Funding	37
7.1.2	Contributions	37

ppiclF is a parallel particle-in-cell library written in Fortran.

Applications of ppilcF include element-based particle-in-cell simulations, such as Euler-Lagrange mutliphase flow simulation, immersed boundary methods, and even atomistic-scale modeling. At its essence, ppiclF's main purpose is to provide a unified and scalable interface for a user to solve the following system of differential equations

$$\frac{d\mathbf{Y}}{dt} = \dot{\mathbf{Y}}$$

which are found in all of the previously given particle-in-cell applications. The library can be downloaded from [ppiclF](#). On this documentation website, you will find more details, theory, examples, questions, etc.

CAPABILITIES

- On-the-fly load-balancing of the system of equations across MPI processing ranks based on the coordinates associated with each particle.
- Simple user input of an external overlapping mesh for interactions between particles and their nearby cells.
- Optional fast binned parallel nearest neighbor search between particles within a user specified distance so that more sophisticated user-implemented right-hand-side forcing models can easily be evaluated.
- Algorithms have demonstrated scalability to 100,000 processors, allowing billions of equations to be solved simultaneously.
- Links to both Fortran and C++ external code as a library.
- Overview of `ppic1F`.

QUICKSTART

2.1 Quickstart

In this section, you can find a guide to install dependencies, build, and link ppiclF.

2.1.1 Dependencies

A valid Fortran and C compiler is required.

Additionally, a valid Message Passing Interface (MPI) standard is required. On ubuntu, this can be installed using

```
sudo apt-get install -y libmpich-dev mpich
```

2.1.2 Build

To install ppiclF, checkout the master branch from the [official GitHub repository](https://github.com/dpzwick/ppiclF). First, create a local directory where the source code will be copied to. Here, we will call this LocalCodeDir:

```
cd LocalCodeDir
git clone https://github.com/dpzwick/ppiclF.git
cd ppiclF
```

Then, set the appropriate properties in the files source/ppiclF_user.f and source/PPICLF_USER.h (see [The H-File](#) and [The F-File](#)). For now, we will copy these files from an existing example:

```
cp examples/stokes_2d/user_routines/ppiclF_user.f source/
cp examples/stokes_2d/user_routines/PPICLF_USER.h source/
```

Now, set the appropriate parameters in the Makefile. Either mpif77 or mpif90 are supported Fortran compilers. Note that mpicc is also required for a 3rd party library internally.

If you are compiling for a C or C++ application, add the flag -DPPICLC to the FFLAGS variable in the Makefile for correct bindings. The code can then be built with:

```
make
```

Upon successful compilation, you should see the following output:

```
*****
***  LIBRARY SUCCESS  ***
*****
```

2.1.3 Link

In order to use the ppiclF library that was built in the last step (see [Build](#)), the library file `source/libppiclF.a` must be linked to your existing code. This will be specific to your existing code compilation. The process of linking will generally involve adding “`-I LocalCodeDir/ppiclF/source`” to the compiler flags and “`-L LocalCodeDir/ppiclF/source -lppiclF`” to the linking flags.

Alternatively, ppiclF may be used on its own as well. Both a Fortran and C++ example of this is given in the simple MPI driver program example [Stokes 2D](#). We will use the Fortran program to test the linking.

First, create a working directory to run the test case. We will call this `TestCaseDir`, and it should be located outside of the cloned GitHub repository ppiclF:

```
mkdir TestCaseDir
```

The example case can then be copied from the cloned GitHub code to `TestCaseDir`

```
cp -r LocalCodeDir/ppiclF/examples/stokes_2d/fortran/* TestCaseDir/  
cd TestCaseDir
```

Included in this test case is a Makefile and a driver program called `test.f`.

This case may be compiled and linked to ppiclF by setting the variable `PPICLF_LOCATION` in the Makefile to `LocalCodeDir/ppiclF` and using the command

```
make
```

The built executable `test.out` can then be run using your MPI wrapper. For example, the following may be used to run with 4 MPI ranks

```
mpirun -np 4 test.out
```

Note the driver program `test.f` is explained in further detail in the [Stokes 2D](#) example.

ALGORITHMS

3.1 Algorithms

In this section, the algorithms used in the ppicIF code are detailed.

3.1.1 Particle Storage

Based on the coordinates (**X**) of each individual particle, the particle domain may be decomposed into rectangular prisms in 3D (or simply rectangles in 2D). These rectangular prisms are called **bins**. The particles with coordinates inside each bin are stored together on the same processor in memory. Additionally, we make the following key assumption: *the total number of bins does not exceed the total number of processors*.

Bin Generation

In order to decompose the particle domain into bins, a recursive planar cutting is used. Given the bounds of the particle domain, which is the extrema coordinates of all particles in each dimension, the recursive planar cutting algorithm below is used.

```
c(1:d) = 1
f(1:d) = 0
while Condition 1 and Condition 2 do

    do i=1,d
        c(i) = c(i) + 1

        if Condition 1 or Condition 3 then
            Continue
        else
            c(i) = c(i) - 1
            f(i) = 1
```

In the previous algorithm, we have defined d to be the number of dimensions (either 2 or 3), $c(i)$ to be a counting array in the i dimension, and $f(i)$ to be a flagging array in the i dimension. The counting array specifies the number of bins in each dimension while the flagging array specifies if more bins can be added in the respective dimension.

Notice that there are three conditions that are checked upon each iteration. The conditions are:

- Condition 1 checks the key assumption: if the total number of bins is less than or equal to the total number of processors.
- Condition 2 deals with the size of the bins. Ideally, $c(1) * c(2) * c(3)$ would be as close to the total number of processors as possible. However, the user may specify a minimum interaction length between nearby particles,

called W . In the current algorithm, this requires the bin lengths in each dimension to be no smaller than W . Thus, condition 2 checks if the length of the bins in every dimension is the smallest that it can be without being smaller than W .

- Condition 3 also deals with the size of the bins. However, condition 3 only checks if the length of the bins in a single i dimension is the smallest that it can be without being smaller than W .

In 2D, an example of this is shown in the figure below.

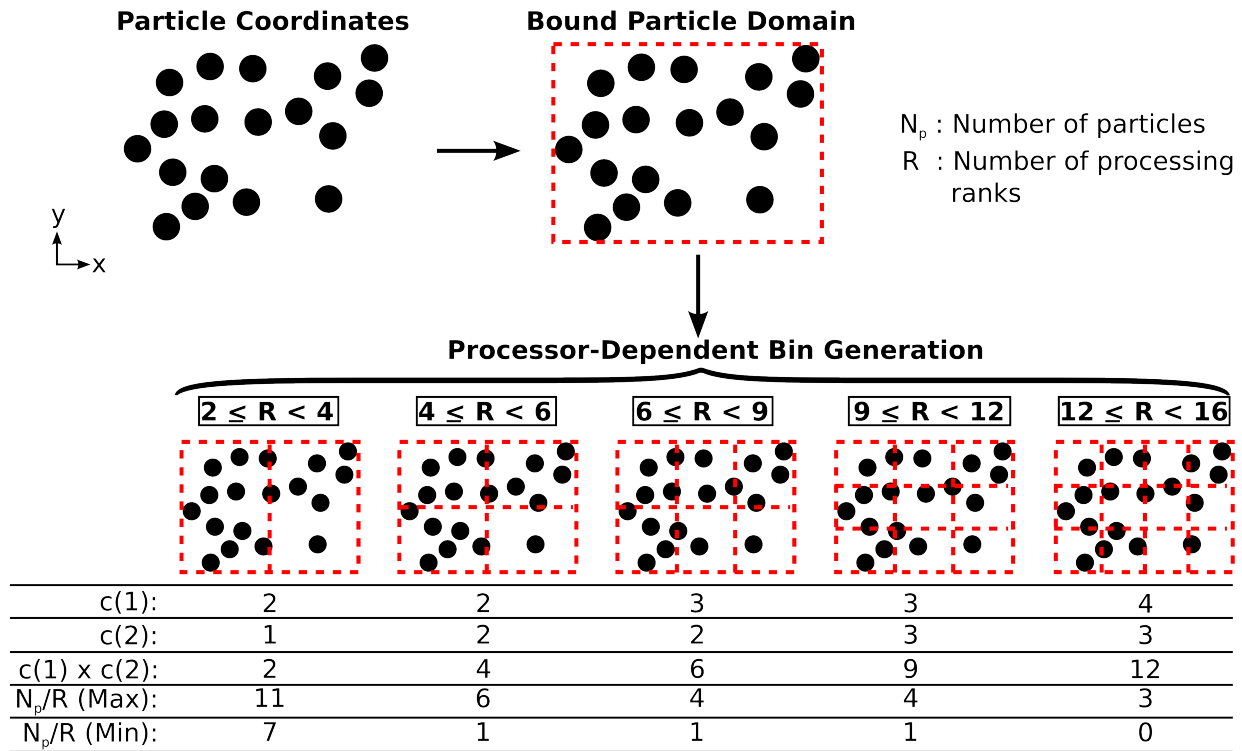


Fig. 1: Example of bin generation through recursive planar cutting.

As is shown in the figure, the bin generation begins by bounding the particle domain, which is the region occupied by the rectangular prism whose planes are determined by the global maximum and minimum coordinates of the particles. Following this, the bins, which are demarcated by the dashed lines, are generated through the recursive planar cutting algorithm above. At the end of the process, different bin configurations are realized based on the number of processing ranks used. The resulting bin configurations are shown in the bottom of the figure above.

Additionally, below each bin distribution, the number of particles N_p per processing rank R is compared between the different cases. Note that since we have required there to only be one bin stored on each processor, N_p/R also refers to the number of particles per bin. Ideally, there would be exactly N_p/R particles on a processor. In the current method, the ideal number of particles per rank is relaxed to allow nearby particles to be locally grouped together in the same rank (or bin). Thus, in the current method we have a deviation from the ideal number of particles on each rank, with ranks having varying numbers of particles. In the figure, the maximum and minimum number of particles on each rank in ppicIF is shown.

Bin-to-Rank Mapping

Once the bins have been created, they are mapped to processing ranks. This includes mapping all the data that has coordinates which are found within the enclosing volume of each bin to the same processor. A single bin is described by the volume which it encloses. To reference a bin, the indices (i, j, k) may be used, where $i = 0, \dots, c(1) - 1$, $j = 0, \dots, c(2) - 1$, and $k = 0, \dots, c(3) - 1$. The volume which each bin encloses is

$$\begin{aligned} x &\in [L_{dx} + iL_{bx}, L_{dx} + (i + 1)L_{bx}], \\ y &\in [L_{dy} + jL_{by}, L_{dy} + (j + 1)L_{by}], \\ z &\in [L_{dz} + kL_{bz}, L_{dz} + (k + 1)L_{bz}], \end{aligned}$$

where L_{d*} is the particle domain width in respective dimension and L_{b*} is the bin width in respective dimension (i.e., $L_{b*} = L_{d*}/c(*)$).

The bin-to-rank mapping, which determines which processing rank stores the data within a given bin, is then performed by mapping each bins indices (i, j, k) to processing rank n_{ID} by

$$n_{ID} = i + c(1)j + c(1)c(2)k.$$

3.1.2 Overlap Mesh

Mapping

In particle-in-cell simulations, the particles are often required to interact with an Eulerian mesh. Since the particle are tracked individually at discrete locations that do not necessarily coincide with a mesh, interactions between the particles and mesh must be handled carefully. We define two key operations between the particles and the mesh:

1. **Interpolation** - field quantities on the mesh are evaluated at the particle's coordinates,
2. **Projection** - particle quantities are filtered from the particle's coordinates to the mesh.

Both of these operations require the user to initially specify the coordinates of the mesh. In the current version of ppicIF, only element based hexahedral (or quad in 2D) meshes are supported.

The elements of the mesh are stored in memory according to the external program which calls ppicIF. For this reason, the elements need to be mapped to bins where the particles are stored. An example of this mapping is shown in the figure below.

In the top left corner of the figure, a semicircular mesh is given. Particles handled by ppicIF are shown in the interior of the mesh. The four bins generated through the algorithm in the section [Particle Storage](#) are also shown surrounding the particles. The particles within the same bin are mapped to the same processor as illustrated in through the coloring of bins in the bottom left corner of the figure.

The elements in the mesh are mapped to processing ranks in the top right of the figure. Note that the ppicIF library is independent of this mesh-to-rank mapping and this is handled in the external calling program. As is shown in the bottom right corner of the figure, the mesh elements which are spatially found within each bin belong to processing ranks that are not the same as the ranks which store the bin data. For this reason, the mesh and corresponding fields found within bins must be communicated between the processors that store the bins themselves. In this example, this results in:

- communication between rank 0 (bin 0) and overlap ranks 0, 1, and 3,
- communication between rank 1 (bin 1) and overlap ranks 0, 2, and 3,
- communication between rank 2 (bin 2) and overlap ranks 0 and 1,
- communication between rank 3 (bin 3) and overlap ranks 0 and 2.

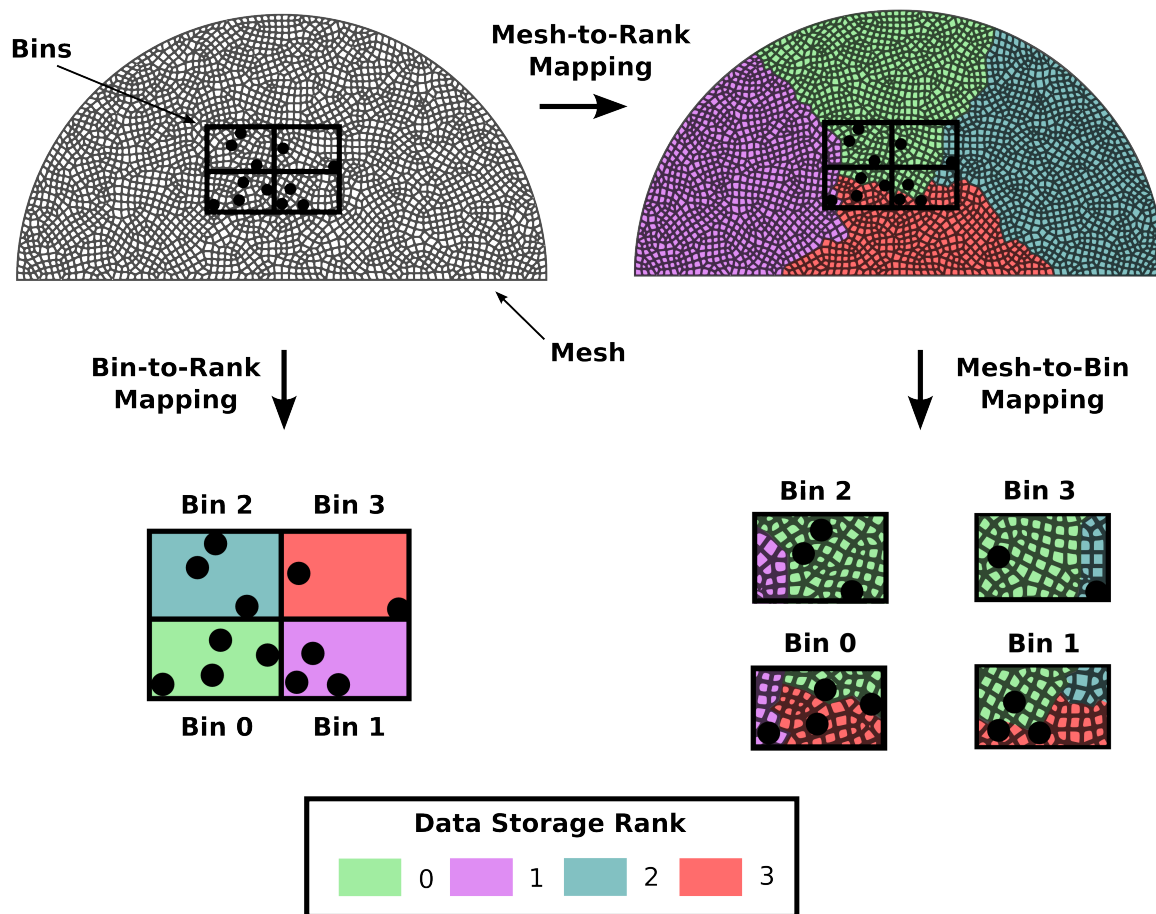


Fig. 2: Illustration of mapping overlapping mesh to bins.

Interpolation

In order to interpolate fields from the mesh to the particle locations, the mesh coordinates and interpolated fields found within a bin are communicated to the processor which stores that bins data. Then, interpolation is performed locally.

The current supported interpolation method is element based, in that only the nodes of the mesh element which surround the particle is used to interpolate a field to the particle's position. In most cases, this results in trilinear (or bilinear in 2D) interpolation being performed. However, spectral polynomial interpolation is also supported in which mapped Gauss-Lobatto-Legendre points within each element are used to evaluate the barycentric Lagrange polynomials at the particles coordinates.

Projection

In order to project quantities from the particle locations to the mesh, each bin stores a copy of the mesh coordinates that are found within its volume. Then, particles are filtered to this mesh copy locally and then communicated back to the processing ranks in the original mesh-to-rank mapping so that the external program has access to the projected fields.

The projection operation is simply a filtering operation which is characterized by a kernel. The current supported projection kernels are the box and Gaussian kernels. The projection operation is performed for any i particle quantity $A^{(i)}$, resulting in the projected field $a(\mathbf{x})$. In this case, \mathbf{x} correspond to the mesh coordinates and $\mathbf{X}^{(i)}$ is the i particle's coordinates. Projection for all N_p particles is then computed by

$$a(\mathbf{x}) = \sum_{i=1}^{N_p} A^{(i)} g_{\mathcal{M}}(|\mathbf{x} - \mathbf{X}^{(i)}|).$$

Here, $g_{\mathcal{M}}(r)$ is the projection kernel. For the Gaussian kernel, we have

$$g_{\mathcal{M}}(r) = \left(\sqrt{\pi} \sqrt{\delta_f^2 / (4 \ln 2)} \right)^{-d} \exp \left(\frac{-r^2}{\delta_f^2 / (4 \ln 2)} \right),$$

where the kernel is parameterized by the filter half-width δ_f in d dimensions. Note that when $r = \delta_f/2$, the kernel has decayed to half of its $r = 0$ value.

The box kernel on the other hand is

$$g_{\mathcal{M}}(r) = \begin{cases} \frac{1}{(4/3)^{d-2} \pi (\delta_f/2)^d}, & r \leq \delta_f/2 \\ 0, & r > \delta_f/2 \end{cases}$$

Similarly, the filter half-width defines the maximum point where distances $r > \delta_f/2$ are outside the filtering region. Note that due to the normalization of each filter, when the field $a(\mathbf{x})$ is integrated over the entire volume \mathcal{V} (or area in 2D) spanned by the mesh, the following relationship is preserved

$$\int_{\mathcal{V}} a(\mathbf{x}) d\mathcal{V} = \sum_{i=1}^{N_p} A^{(i)}.$$

3.1.3 Ghost Particles

Particles in the interior of bins have access to locally stored particles through the mapping of bins to ranks (see [Particle Storage](#)) and the locally stored mesh through mapping the mesh to bins (see [Overlap Mesh](#)). However, particles near the boundary of bins may interact with data that is stored in the processing ranks associated with neighboring bins. This interaction may occur in two ways:

1. Particles near the edge of one bin need information from particles in a neighboring bin,
2. Projection of particles near the edge of one bin spills over into neighboring bins.

In either case, ppicIF handles these interactions through the use of ghost particles. Ghost particles are simply copies of computational particles which are used to communicate particle data to neighboring bins that are stored on different processors.

Recall that a user may specify a minimum interaction length, which we called w (see [Particle Storage](#)). This length specifies the minimum length of bins in each dimension, even though the actual length of bins is most often determined by the number of processors used. For this reason, a single ghost particle is created for each bin that touches the current bin that a particle is stored with when the distance from the particle to the closest point in the neighboring bin is less than w .

As a result, only the single layer of neighbor bins to the bin a particle is stored in may receive a ghost particle. Due to the Cartesian layout of bins, this results in at most 26 ghost particles in 3D and 8 ghost particles in 2D being created for each computational particle.

3.1.4 Performance

Description

In this section, the performance of the ppicIF implementation of these algorithms is tested. In the following simulations, ppicIF has been coupled with the incompressible fluid solver [Nek5000](#).

Here, the fluid discretization is composed of hexahedral spectral elements, each with Gauss-Lobatto-Legendre (GLL) quadrature points. Since the GLL points are collocated with the Eulerian field variables, particle-fluid interactions (i.e., [Interpolation](#) and [Projection](#)) occur between the overlap GLL mesh within each element and nearby particles.

In this example, 9 fields are interpolated and 7 fields are projected, which corresponds to typical multiphase flow simulations. For simplicity, each spectral element is a cube with side lengths L_e and $N = 6$ quadrature points in each dimension. Additionally, the entire domain is a cube with side lengths L_D .

Within the domain, N_p particles are also distributed with uniform random probability in a cube with side lengths of L_P , with this cube being centered inside the GLL elemental domain. Each particle is assumed to be spherical with diameter D_p and density ρ_p . For projection, a cut-off radius for the Gaussian filter $r_c/L_e = 0.35$ is used, which bounds the number of grid points each projected particle quantity will be spread over relative to the particle's coordinates. The soft-sphere collision model in the [DEM Packing 3D](#) example is used. In each case, $L_e = 11.25$ mm, $\rho_p = 2,500$ kg/m³, $k_c = 1,000$ kg/s², and $e_n = 0.9$. Each case was simulated with R MPI ranks of the [Vulcan](#) supercomputer at Lawrence Livermore National Laboratory.

In order to understand how the algorithms in ppicIF perform, we have looked at multiple configurations of the test problem under strong and weak processor scaling. The exact parameters for each case are found in the table below.

Table 1: Scaling case parameters.

Scaling	Case	R	N_p	D_p (μm)	L_p/L_D (m)
Strong	1	1,024	25,165,824	511	0.54/0.54
	2	4,096	“	“	“
	3	8,192	“	“	“
	4	16,384	“	“	“
	5	32,768	“	“	“
	6	98,304	“	“	“
Weak	7	1,024	3,145,728	232	0.12/0.54
	8	4,096	12,582,912	“	0.19/0.54
	9	8,192	25,165,824	“	0.24/0.54
	10	16,384	50,331,648	“	0.30/0.54
	11	32,768	100,663,296	“	0.37/0.54
	12	98,304	301,989,888	“	0.54/0.54

Results

For each of these cases, the maximum time on a processor was measured per time step for different algorithms of ppicIF. The strong and weak scaling of these cases is shown in the two figures below on a $\log_{10} - \log_{10}$ scale. Note that slopes of -1 and -2 are shown in the dashed lines in the strong scaling plot and a slope of 0 is shown on the weak scaling plot.

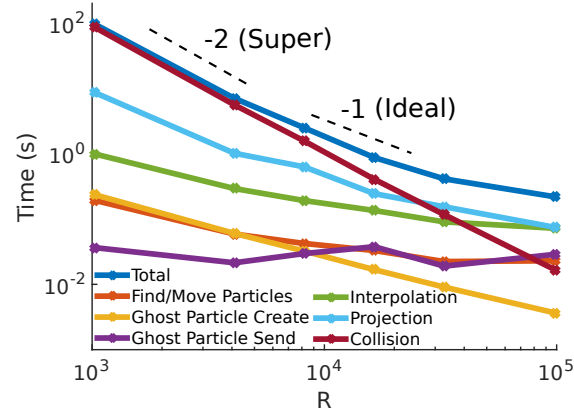


Fig. 3: Strong scaling results (cases 1-6).

Additionally, the log-log slopes b of each of these lines have been computed for a functional scaling of the form $t = aR^b$ in the table below.

Table 2: Log-log slopes b for $t = aR^b$ for different ppicIF algorithms.

	Strong Scaling	Weak Scaling
Total	-1.36	-0.38
Find/Move	-0.47	-0.15
Ghost Particles	-0.48	0.32
Interpolation	-0.58	-0.64
Projection	-1.03	-0.04
Collision	-1.87	-0.07

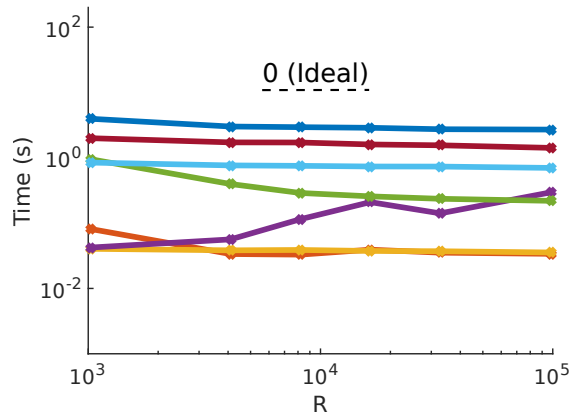


Fig. 4: Weak scaling results (cases 7-12).

In the strong scaling, the problem size is fixed with nearly 25 million particles while the number of processors are increased from approximately 10^3 to 10^5 ranks. It is observed that collisions dominate the total time taken per time step when N_p/R is large in case 1. This is expected since the collision algorithm scales as $O((N_p/R)^2)$ whereas other portions of the algorithm only scale as $O(N_p/R)$. Despite this, when the number of ranks are increased, the cost of collisions is driven down at a rate of $R^{-1.87}$ (refer to log-log slope table above) even when 10^5 processors are used. Note that -1.87 is close to the algorithmically expected value of -2. Since ideal strong scaling would have a slope of -1, we refer to the collision algorithm as having a “super” scaling. Also, projection and interpolation are the next most costly portions of ppiclF. Projection is more costly than interpolation even though in case 6 they are of comparable cost. Overall, projection has been found to scale as $R^{-1.03}$ and interpolation as $R^{-0.58}$. Both projection and interpolation have begun to level off in cases 5-6 even though they perform quite well at first. This is due to the inherent communication associated with the overlap mesh and nearby particles not residing on the same processor. Thus, for greater numbers of particles on a processor (case 1) the communication costs in sending elemental data to the overlapping bins is much less than the computational costs of particle-element interactions. However, when fewer particles per rank are used, the communication costs begin to dominate which is observed in cases 5-6.

In the weak scaling cases, the problem size per processor is fixed at $N_p/R = 3,072$ as the number of ranks are scaled from approximately 10^3 to 10^5 . This results in over 300 million total particles in case 12. We observe similar trends compared to the strong scaling. Note that an ideal weak scaling slope would be 0 since the problem size per rank doesn’t change. For the most part, the weak scaling of ppiclF is exceptional with most of the algorithmic slopes being less than 0. The reason that most algorithms perform better than ideal is that the bin generation (see [Particle Storage](#)) equally partitions the particle domain into a near-ideal distribution with increasing number of processors. The only algorithmic portion which increases under weak scaling is the sending of ghost particles. This routine relies upon the open source [GSLIB](#) tool for efficient parallel data transfer under the hood. GSLIB adheres to a logarithmic many-to-many scaling of $\log_2 R$ and is the back bone of ppiclF’s parallel communication. Thus, we expect to see this increase in sending ghost particles since the total number of processing ranks is also increasing. While sending ghost particles becomes non-trivial at 10^5 processors, if we extrapolate the performance to 10^6 ranks we presume that even at such extreme scales ppiclF is not consumed by communication costs.

USER INTERFACE

4.1 User Interface

In this section, the user interface is detailed. This consists of the user providing two required files to the source code, which are the H-File and the F-File. In order to drive the library, the external interface provides possible variables and subroutines that can be called from an outside program.

Note that a user can copy an existing H-File, F-File, and external calls from a previous example case to fit their setup. Alternatively, there is a simple python script `usergen.py` which will walk the user through generating these files for a specific case from scratch. This file can be used by issuing the command `python usergen.py`.

4.1.1 The H-File

The H-File refers to the file `PPICLF_USER.h`. This file is required for each case and specifies the maximum size of different arrays used in the code. These sizes are defined by C pre-processor directives, which use the `#define NAME N` syntax, where `NAME` is the variable name in capital letters and `N` is the integer number that `NAME` is replaced by everywhere in the code at compilation.

The pre-processor directives, along with the actual arrays and other `ppiclf` variables can be included for use in any Fortran subroutine by simply including the following directive at the top of a routine

```
#include "PPICLF.h"
```

Pre-Defined Directives

A list of pre-defined names are found in the table below with their descriptions.

Table 1: Pre-processor pre-defined names in `PPICLF_USER.h`.

NAME	Required?	Description
<code>PPICLF_LRS</code>	Yes	The number of equations solved per particle.
<code>PPICLF_LRP</code>	No	The number of additional properties per particle.
<code>PPICLF_LPART</code>	No	The maximum number of particles per rank.
<code>PPICLF_LRP_INT</code>	No	The number of interpolated fields.
<code>PPICLF_LRP_PRO</code>	No	The number of projected fields.
<code>PPICLF_LEE</code>	No	The maximum number of overlap mesh elements per rank.
<code>PPICLF_LEX</code>	No	The number of x coordinates per overlap mesh element.
<code>PPICLF_LEY</code>	No	The number of y coordinates per overlap mesh element.
<code>PPICLF_LEZ</code>	No	The number of z coordinates per overlap mesh element.
<code>PPICLF_LWALL</code>	No	The maximum number of triangular patch boundaries.

User-Only Directives

A list of user-only suggested names are found in the table below with their descriptions. Each of these names can be chosen by the user to reflect the equations being solved.

Table 2: Pre-processor suggested names in PPICLF_USER.h.

Suggested NAME	Description
PPICLF_J*	Index in solution array.
PPICLF_R_J*	Index in property array.
PPICLF_P_J*	Index in projected field array.

Here, the above suggested names may be used for easy naming with user-edited coded. For example, consider the system of equations

$$\frac{d\mathbf{Y}}{dt} = \dot{\mathbf{Y}},$$
$$\mathbf{Y} = \begin{bmatrix} X \\ Y \\ V_x \\ V_y \end{bmatrix}, \quad \dot{\mathbf{Y}} = \begin{bmatrix} V_x \\ V_y \\ -aV_x \\ -aV_y + b \end{bmatrix}.$$

In this case, we can refer to the equations as solving for the variables X , Y , V_x , and V_y . As a result, the user may want to refer to the equations as

```
#define PPICLF_JX 1
#define PPICLF_JY 2
#define PPICLF_JVX 3
#define PPICLF_JVY 4
```

Then, the user can use these directive names in place of the index j for the i particle in the solution arrays `ppiclf_y(j,i)`, `ppiclf_ydot(j,i)`, and `ppiclf_ydotc(j,i)`.

Caution: User ordering of solution array indices.

1. The indices are ordered from 1 to PPICLF_LRS.
 2. The first two (2D) or three (3D) indices must always be the X , Y , and Z (3D only) coordinates of each particle.
-

Similarly, if a and b are properties of each particle that are not being solved for, the user may want to refer to the properties as

```
#define PPICLF_R_JA 1
#define PPICLF_R_JB 2
```

Then, the user can use these directive names in place of the index j for the i particle in the property array `ppiclf_rprop(j,i)`.

Caution: User ordering of property array indices.

1. The indices are ordered from 1 to PPICLF_LRP.
-

Similarly, if the user is projecting any properties, each mapped (see [The F-File](#)) projected field can also be referenced in this way. Consider three projected fields $f(\mathbf{x})$, $g(\mathbf{x})$, and $h(\mathbf{x})$. The user may want to refer to these fields as

```
#define PPICLF_P_JF 1
#define PPICLF_P_JG 2
#define PPICLF_P_JH 3
```

Then, the user can use these directive names in place of the index m for the (i,j,k) coordinate of the e element on the overlap mesh in the projected field array `ppiclf_pro fld(i,j,k,e,m)`.

Caution: User ordering of projection array indices.

1. The indices are ordered from 1 to `PPICLF_LRP_PRO`.
-

4.1.2 The F-File

The F-File refers to the file `ppiclf_user.f`. This file is required for each case and specifies three key subroutines in the code. Each of these subroutines are described below. In what follows, the terms *int* and *real* refer to the Fortran integer*4 and real*8 types, which are also the types `int` and `double` in C.

ppiclf_user_SetYdot

The main purpose of this subroutine is described below.

ppiclf_user_SetYdot()

This routine must set $\dot{\mathbf{Y}}$ for the system of equations

$$\frac{d\mathbf{Y}}{dt} = \dot{\mathbf{Y}}$$

This is done by specifying the j indices (corresponding to the j equations) of the array `ppiclf_ydot(j,i)` for the i particle.

Typically, this subroutine amounts to a do-loop over the index i which loops through the total number of particles on each processor (`ppiclf_npart`). The user is then required to specify $\dot{\mathbf{Y}}^{(i)}$ by setting the j indices ranging from 1 to `PPICLF_LRS` of the array `ppiclf_ydot(j,i)`. Since $\dot{\mathbf{Y}}^{(i)}$ may be a function of \mathbf{Y} and other properties associated with each particle, the user-defined pre-processor directives can be useful in organizing this evaluation.

Note that additionally, the routine `ppiclf_solve_NearestNeighbor` can optionally be called for the i particle within the do-loop. When this occurs, the nearby neighbor searching routine `ppiclf_user_EvalNearestNeighbor` is activated (see below).

After `ppiclf_ydot(j,i)` is set, a user may also choose to store some computed values in the array `ppiclf_ydotc(j,i)` which is ordered the same as `ppiclf_ydot(j,i)`. Values stored in this array are then passed to the projection mapping routine `ppiclf_user_MapProjPart` (see below).

ppiclf_user_MapProjPart

The main purpose of this subroutine is to specify which particle properties A are projected to which fields $a(\mathbf{x})$. For more details on the projection operation, see [Projection](#). Note that projection can only be performed when there is an overlap mesh specified. A description of this routine is given below.

ppiclf_user_MapProjPart(real map, real y, real ydot, real ydotc, real rprop)

This routine specifies which particle properties are projected to which fields.

- **map(m)** is a vector input of length `PPICLF_LRP_PRO` which must be set, corresponding to the particle property *A*. The *m* index corresponds to which field is being projected in the `ppiclf_pro_fld(i,j,k,e,m)` array. The user-defined pre-processor directives can be useful in organizing this evaluation.
 - **y(j)**, **ydot(j)**, and **ydotc(j)** are vector inputs of length `PPICLF_LRS`. They correspond to the values in the arrays of the *i* particle `ppiclf_y(j,i)`, `ppiclf_ydot(j,i)`, and `ppiclf_ydotc(j,i)`. Each of these can be used in evaluating **map(m)**.
 - **rprop(j)** is a vector input of length `PPICLF_LRP`. It corresponds to the values in the array of the *i* particle `ppiclf_rprop(j,i)`. Its values can be used in evaluating **map(m)**.
-

ppiclf_user_EvalNearestNeighbor

The main purpose of this subroutine is to allow the user to have access to neighboring particle's information. For the *i* particle, the *j* nearest points are passed in within a distance *W* (see [Particle Storage](#)). This routine is only activated when the routine `ppiclf_user_NearestNeighbor` is called for the *i* particle in the do-loop in `ppiclf_user_SetYdot` (see [DEM Packing 3D](#) for an example of this). The routine `ppiclf_user_NearestNeighbor` is described below.

ppiclf_user_NearestNeighbor(int i)

This routine specifies that the `ppiclf_user_EvalNearestNeighbor` routine should be called.

- **i** is the *i* particle for which the nearest neighbors are found.
-

Additionally, the routine `ppiclf_user_EvalNearestNeighbor` is described below.

ppiclf_user_EvalNearestNeighbor(int i, int j, real yi, real rpropi, real yj, real rpropj*)

This routine evaluates the nearest neighbors of the *i* particle. The neighbors are the *j* points which are either nearby particles or nearby boundaries. This routine is only activated when `ppiclf_user_NearestNeighbor` is called for the *i* particle in `ppiclf_user_SetYdot`.

- **i** is the *i* particle for which the nearest neighbors are found for. The index *i* may also be used to access the internal arrays (i.e., `ppiclf_y(:,i)`, `ppiclf_ydot(:,i)`, `ppiclf_ydotc(:,i)`, and `ppiclf_rprop(:,i)`).
 - **j** is the *j* point for which the nearest neighbors of particle *i* are found. **DO NOT USE j TO ACCESS THE INTERNAL ARRAYS.** The index *j* may be positive, negative, or zero. A positive value means that the neighbor particle is on the same processor. A negative value means that the neighbor particle is on a different processor. A value of *j* = 0 means that the neighbor point is a boundary point.
 - **yi** is a vector of length `PPICLF_LRS` which passes in the solution variables of the *i* particle.
 - **rpropi** is a vector of length `PPICLF_LRP` which passes in the properties of the *i* particle.
 - **yj** is a vector of length `PPICLF_LRS` which passes in the solution variables of the *j* particle. Note that when a boundary point is passed in as point *j* (*j* == 0), then only the first two (2D) or three (3D) values of **yj** can be used, which store the nearest point on the boundary to the *i* particle.
 - **rpropj** is a vector of length `PPICLF_LRP` which passes in the properties of the *j* particle. Note that when a boundary point is passed in as point *j* (*j* == 0), the values of **rpropj** are meaningless.
-

4.1.3 External Interface

The following subroutines and variables can be called from an external program's pre-compilation source code. The source code may be found in an application that you are linking ppiclF to (see *Nek5000 Case*) or a simple driver program that links to ppiclF (see *Stokes 2D*).

In what follows, the terms *int* and *real* refer to the Fortran integer*4 and real*8 types, which are also the types int and double in C.

Common Variables

The following common variables are stored in memory and may be included by adding the following directive to the top of a routine.

```
#include "PPICLF.h"
```

Table 3: Common variables used and their descriptions.

Name	Type	Description
ppiclf_y(j,i)	real	Array of j equations solution for i particle ($\mathbf{Y}^{(i)}$).
ppiclf_ydot(j,i)	real	Array of j equations forcing for i particle ($\dot{\mathbf{Y}}^{(i)}$).
ppiclf_ydotc(j,i)	real	Array of j equations extra storage for i particle.
ppiclf_rprop(j,i)	real	Array of j properties for i particle.
ppiclf_npart	int	The number of local particles on the current processor.
ppiclf_pro_fld(i,j,k,e,m)	real	Projected field m at overlap mesh point (i,j,k) on element e.

Initialization Subroutines

The following subroutines are only to be called once at the beginning of a simulation. Note that every routine does not need to be called. The actual subroutines called at setup are problem dependent. However, the subroutines **ppiclf_comm_InitMPI** and **ppiclf_solve_InitParticle** must be called at least once at initialization for every case.

ppiclf_comm_InitMPI(int comm, int id, int np)

The user inputs the current MPI communicator **comm** where the current MPI rank is **id** and there are **np** total ranks.

ppiclf_comm_InitOverlapMesh(int ncell, int lx, int ly, int lz, real xgrid, real ygrid, real zgrid)

The user inputs the element based mesh. The arrays **xgrid**, **ygrid**, and **zgrid** are of size **(lx,ly,lz,ncell)** and stores the mesh coordinates. Here, **ncell** is the number of local hexahedral elements on rank **id**, with **lx,ly,lz** points in each dimension.

- **ncell** ≤ PPICLF_LEE.
 - **lx** == PPICLF_LEX.
 - **ly** == PPICLF_LEY.
 - **lz** == PPICLF_LEZ.
-

ppiclf_io_ReadParticleVTU(char*1 filein(132))

The user specifies **filein** with .vtu extension to use as the initial conditions for particles. When this routine is called, the arrays `ppiclf_y(j,i)` and `ppiclf_rprop(j,i)` will automatically be filled.

- Note that `ppiclf_solve_InitParticle` should still be called following this, but with a dummy argument in place of **y** and **rprop**.
-

ppiclf_io_ReadWallVTK(char*1 filein(132))

The user specifies **filein** with .vtk extension to use as triangular plane boundaries.

The format of the file is a minimized ASCII VTK format as shown below.

In 3D:

```
POINTS npoints
p1x p1y p1z
p2x p2y p2z
...
CELLS ncells
n1ip1 n1ip2 n1ip3
n2ip1 n2ip2 n2ip3
...
```

and in 2D:

```
POINTS npoints
p1x p1y p1z
p2x p2y p2z
...
CELLS ncells
n1ip1 n1ip2
n2ip1 n2ip2
...
```

where:

- a tri-element surface mesh is required in 3D problems,
- a line-element surface mesh is required in 2D problems,
- `npoints` is total number of points that follow,
- `pab` is point `a` coordinate in `b` dimension,
- `nwalls` is total number of walls that follow,
- `ncipd` is the index of the points from 0 to `npoints-1` that make up the wall with `c` being the wall number and `d` being the arbitrary ordering of points.

This format can actually be output using the free finite element mesh generator [Gmsh](#). The process is to create the appropriate mesh, export as a VTK file, and then remove everything except the format as specified above. **Please make sure there are no blank lines in file.**

ppiclf_solve_InitWall(real xp1, real xp2, real xp3)

The user manually sets a boundary. This is similar to `ppiclf_io_ReadWallVTK`, but the user may only want to manually set a single or few walls manually without a VTK file.

- The inputs are all vectors of length two (2D) or three (3D).

- **xp1** stores coordinates (p1x, p1y, p1z) as in the VTK file.
 - **xp2** stores coordinates (p2x, p2y, p2z) as in the VTK file.
 - **xp3** stores coordinates (p3x, p3y, p3z) as in the VTK file.
-

ppiclf_solve_InitNeighborBin(real W)

The user specifies **W** as the minimum interaction distance to resolve. See [Particle Storage](#).

ppiclf_solve_InitSuggestedDir(char*1 dir)

The user inputs **dir** dimension which lets the bin generation algorithm attempt to create more bins in chosen dimension.

- **dir** = 'x', 'y', or 'z'.
-

ppiclf_solve_InitPeriodicX(real a, real b)

- The user sets periodicity in the y-z planes at $x = \mathbf{a}$ and $x = \mathbf{b}$.
 - Note: $\mathbf{a} < \mathbf{b}$.
-

ppiclf_solve_InitPeriodicY(real a, real b)

The user sets periodicity in the x-z planes at $y = \mathbf{a}$ and $y = \mathbf{b}$.

- $\mathbf{a} < \mathbf{b}$.
-

ppiclf_solve_InitPeriodicZ(real a, real b)

The user sets periodicity in the x-y planes at $z = \mathbf{a}$ and $z = \mathbf{b}$.

- $\mathbf{a} < \mathbf{b}$.
-

ppiclf_solve_InitGaussianFilter(real f, real a, int iflg)

The Gaussian filter half-width for projection δ_f is **f** (see [Overlap Mesh](#)). The value **a** is the percent of the $r = 0$ value that the Gaussian filter is computationally allowed to decay to. The flag **iflg** sets if particles should be mirrored and then projected across boundaries or not.

- $\mathbf{a} < 1.0$.
 - **iflg** = 0 (no mirror) or 1 (mirror).
-

ppiclf_solve_InitBoxFilter(real f, int iflg)

The box filter half-width for projection δ_f is **f** (see [Overlap Mesh](#)). The flag **iflg** sets if particles should be mirrored and then projected across boundaries or not.

- **iflg** = 0 (no mirror) or 1 (mirror).
-

ppiclf_solve_InitParticle(int imethod, int ndim, int iendian, int npart, real y, real rprop)

The user also initializes Y_0 , which is the initial condition of Y for the system of equations. The user also sets the integration method **imethod**, the problem dimension **ndim**, the byte ordering **iendian**, and the number of particles being initialized on the current rank **npart**.

- **imethod** = +/-1 (RK3). When **imethod** is negative, the user is in charge of looping through both steps **AND** stages.
 - **ndim** = 2 or 3.
 - **iendian** = 0 (little endian) or 1 (big endian).
 - **npart** ≥ 0 .
-

Solve Subroutines

The following subroutines can be called at every time step. Note that every routine does not need to be called. The actual subroutines called at each time step are problem dependent. However, the subroutine **ppiclf_solve_IntegrateParticle** must be called at least once every time step for any operations to be performed.

ppiclf_solve_InterpFieldUser(int ifld, real fld)

The user sets which field array **fld** of same dimensions set in initialization call to **ppiclf_comm_InitOverlapMesh** routine is used to interpolate to index in property array **ppiclf_rprop(ifld,i)** for the i particle. This routine may be called multiple times to interpolate multiple fields.

- **ifld** \leq PPICLF_LRP.
-

ppiclf_solve_IntegrateParticle(int istep, int iostep, real dt, real time)

This routine must be called every time step and uses the initial integration method to advance Y in time. The inputs are the current **time**, the current time step **dt** to advance by, the current time step number **istep**, and will output solution files every **iostep** number of time steps. Note that the user must call this routine every time step for the system to be integrated. When **imethod** is initialized to be a negative number, this routine must also be called every stage as well.

- **istep** ≥ 0 .
 - **iostep** ≥ 0 .
 - **dt** ≥ 0.0 .
 - **time** ≥ 0.0 .
-

4.1.4 File Output

Two different files are output for viewing the solution. The default names for these files are:

1. parXXXXXX*.vtu,
2. binXXXXXX*.vtu,

where XXXXX refers to a particular file output number ordered from 00001 to the number of output files in a simulation. Each file is in a combined binary/ASCII VTU format.

The par-prefixed file stores the solution **Y** and property arrays of each particle according to the order in *The H-File*. Additionally, a three part tag is given to every particle and also stored in the par files so that any particle can be uniquely identified throughout a simulation. These files can be used to restart from a particular output step by calling the external interface routine `ppiclf_io_ReadParticleVTU` at initialization.

The bin file stores the volumes enclosed by the bins and how many particles are stored within that bin (see *Particle Storage*). This can be useful for analyzing the distribution of particles-to-ranks throughout a simulation.

These files may be viewed and/or post-processed in outside programs, such as *VisIt* or *ParaView*.

EXAMPLES

5.1 Examples

In this section, example cases are given demonstrating the use of ppicIF in different applications.

5.1.1 Stokes 2D

Background

This is a simple example that illustrates Stokes drag on solid spherical particles. This can be illustrated by the following system of equations. For each particle, we have

$$\begin{aligned}\frac{d\mathbf{X}}{dt} &= \mathbf{V}, \\ M_p \frac{d\mathbf{V}}{dt} &= \mathbf{F}_{qs} + \mathbf{F}_b,\end{aligned}$$

where, for each particle we have the position vector \mathbf{X} , the velocity vector \mathbf{V} , the Stokes drag force \mathbf{F}_{qs} , and the weight force \mathbf{F}_b , defined by

$$\mathbf{X} = \begin{bmatrix} X \\ Y \end{bmatrix}, \quad \mathbf{V} = \begin{bmatrix} V_x \\ V_y \end{bmatrix}, \quad \mathbf{F}_{qs} = \frac{M_p}{\tau_p} \begin{bmatrix} V_x \\ V_y \end{bmatrix}, \quad \mathbf{F}_b = M_p \begin{bmatrix} 0 \\ g \end{bmatrix}.$$

Here, each particle has the same mass M_p , time scale τ_p , and gravitational acceleration g . Thus, for each particle we can write the following system of equations

$$\frac{d\mathbf{Y}}{dt} = \dot{\mathbf{Y}},$$

where

$$\mathbf{Y} = \begin{bmatrix} X \\ Y \\ V_x \\ V_y \end{bmatrix}, \quad \dot{\mathbf{Y}} = \begin{bmatrix} V_x \\ V_y \\ -V_x/\tau_p + 0 \\ -V_y/\tau_p - g \end{bmatrix}.$$

User Interface

The H-File for this case ([Stokes 2D H-File](#)) is given below and corresponds to the equations being solved and the property being stored for each particle. Note that since g is constant, we do not included it in the list of properties.

```
#define PPICLF_LRS 4
#define PPICLF_LRP 1

#define PPICLF_JX 1
#define PPICLF_JY 2
#define PPICLF_JVX 3
#define PPICLF_JVY 4
#define PPICLF_R_JTAUP 1
```

The two blocks of lines denote the pre-defined and user-only directives. The pre-defined directives are in the top block and are the number of equations and the number of properties. The user-only directives are in the bottom block.

The *F-File* for this case (Stokes 2D F-File) only has meaningful information in `ppiclf_user_SetYdot`. The other two routines `ppiclf_user_MapProjPart` and `ppiclf_user_EvalNearestNeighbor` are defined only.

```
subroutine ppiclf_user_SetYdot
!
  implicit none
!
#include "PPICLF.h"
!
! Internal:
!
  real*8 fqsx, fqsy, fbx, fby
  integer*4 i
!
! evaluate ydot
  do i=1,ppiclf_npart
    ! Stokes drag
    fqsx = -ppiclf_y(PPICLF_JVX,i)/ppiclf_rprop(PPICLF_R_JTAUP,i)
    fqsy = -ppiclf_y(PPICLF_JVY,i)/ppiclf_rprop(PPICLF_R_JTAUP,i)

    ! Gravity
    fbx = 0.0d0
    fby = -9.8d0

    ! set ydot for all PPICLF_LRS number of equations
    ppiclf_ydot(PPICLF_JX ,i) = ppiclf_y(PPICLF_JVX,i)
    ppiclf_ydot(PPICLF_JY ,i) = ppiclf_y(PPICLF_JVY,i)
    ppiclf_ydot(PPICLF_JVX,i) = fqsx+fbx
    ppiclf_ydot(PPICLF_JVY,i) = fqsy+fby
  enddo
! evaluate ydot

  return
end
```

In this example, the do-loop loops through the total number of particles on each processor. The user computes the stokes drag force and weight in each direction for each particle. Then, the 4 equations are specified according to the system of equations defined in this case.

The *External Interface* calls for this example occur in a simple driver program in the file `test.f` with the minimum number of initialization and solve subroutines called. In this case:

- `ppiclf_comm_InitMPI` is called to initialize the communication,
- `ppiclf_comm_InitParticle` is called with initial properites and conditions for the particles,
- `ppiclf_solve_IntegrateParticle` is called in a simple time step loop.

Compiling and Running

This example can be tested by issuing the following commands:

```
cd ~
git clone https://github.com/dpzwick/ppiclF.git      # clone ppiclF
mkdir TestCase                                     # make test directory
cd TestCase
cp ../ppiclF/examples/stokes_2d/fortran/* .         # copy example files to test case
cp -r ../ppiclF/examples/stokes_2d/user_routines/ . # copy example files to test case
cd ../ppiclF                                        # go to ppiclF code
cp ../TestCase/user_routines/* source/             # copy ppiclF_user.f and PPICLF_
↳USER.h to source
make                                                 # build ppiclF
cd ../TestCase
make                                                 # build test case and link with_
↳ppiclF
mpirun -np 4 test.out                             # run case with 4 processors
```

Simulation Output

The system of equations can analytically be solved subject to the previously given initial conditions. The solution is

$$\mathbf{Y} = \begin{bmatrix} X \\ Y \\ V_x \\ V_y \end{bmatrix} = \begin{bmatrix} X_0 \\ Y_0 - t - e^{-tg}/g \\ 0 \\ e^{-tg} - 1 \end{bmatrix}.$$

As a result, it is clear that the velocity V_y will increase exponentially in time at a rate of $\tau_p = g^{-1}$, eventually reaching $V_y(t \rightarrow \infty) \approx -1$.

In the user code above, $g = 9.8$ and the driver program runs for a total time of $t = 0.1 \approx \tau_p$. The analytical velocity at this time is $V_y = -0.62468890114$. The simulation output in this case can be confirmed to be $V_y = -0.62468886375$.

In this case, third order Runge-Kutta time integration was used with a time step of 10^{-4} , resulting in error of the order 10^{-8} when compounded over 1000 time steps. Since ppiclF outputs only 4-byte precision on the real numbers which is accurate to 7 digits, we expect the precision to be more important than the third order truncation error. To test this, the difference between the simulation output and analytic solution is 0.00000003739, reflecting the larger byte precision.

If instead we change the time step to 10^{-2} and the number of time steps to 10, we expect the new truncation error of order 10^{-5} to be larger than byte precision. To confirm this, the simulation with these parameters give $V_y = -0.62470448017$. The difference between the simulation output and analytic solution is 0.00001557903, reflecting the larger truncation error.

5.1.2 DEM Packing 3D

Background

This is an example that illustrates a particle packing in a realistic geometry. To accomplish this, an efficient neighbor search is performed along with reading in external boundary conditions. The following system of equations is solved

for the i particle

$$\begin{aligned}\frac{d\mathbf{X}^{(i)}}{dt} &= \mathbf{V}^{(i)}, \\ M_p^{(i)} \frac{d\mathbf{V}^{(i)}}{dt} &= \mathbf{F}_c^{(i)} + \mathbf{F}_b^{(i)},\end{aligned}$$

where, for each particle we have the position vector \mathbf{X} , the velocity vector \mathbf{V} , the collision force \mathbf{F}_c , and the weight force \mathbf{F}_b , defined by

$$\begin{aligned}\mathbf{X}^{(i)} &= \begin{bmatrix} X^{(i)} \\ Y^{(i)} \\ Z^{(i)} \end{bmatrix}, \quad \mathbf{V}^{(i)} = \begin{bmatrix} V_x^{(i)} \\ V_y^{(i)} \\ V_z^{(i)} \end{bmatrix}, \\ \mathbf{F}_c^{(i)} = \begin{bmatrix} F_{cx}^{(i)} \\ F_{cy}^{(i)} \\ F_{cz}^{(i)} \end{bmatrix} &= \sum_{j=1}^{N_p^{(i)}} -k_c \delta^{(i,j)} \mathbf{n}^{(i,j)} - \eta \mathbf{V}^{(i,j)}, \quad \mathbf{F}_b = M_p^{(i)} \begin{bmatrix} 0 \\ g \\ 0 \end{bmatrix}.\end{aligned}$$

Here, we have used the mass of each particle, M_p and the gravitational acceleration g . Additionally, the collision force is a soft-sphere normal force which only acts on physically overlapping particles. Due to this, the particle must add the forces from all j overlapping nearby particles. The variable $N_p^{(i)}$ is the number of nearby overlapping particles, which is local to the i particle itself.

For the collision model, the user must specify the two numerical constants, which are the spring stiffness k_c and the coefficient of restitution e_n . The distance between two particle's center locations is

$$d^{(i,j)} = |\mathbf{X}^{(j)} - \mathbf{X}^{(i)}|.$$

The unit normal vector between nearby particles is computed by

$$\mathbf{n}^{(i,j)} = (\mathbf{X}^{(j)} - \mathbf{X}^{(i)}) / d^{(i,j)}.$$

Two particles are said to be overlapping, and thus have a non-zero collision force, when the overlap $\delta^{(i,j)} > 0$. The overlap is computed by

$$\delta^{(i,j)} = \frac{1}{2}(D_p^{(i)} + D_p^{(j)}) - d^{(i,j)},$$

where D_p is the particle's diameter.

Additionally, the particles dissipate energy when they collide. Thus, the collision force also includes the damping coefficient η , computed by

$$\eta = -2\sqrt{M^{(i,j)}k_c} \frac{\ln e_n}{\sqrt{\ln^2 e_n + \pi^2}},$$

where $M^{(i,j)} = (1/M_p^{(i)} + 1/M_p^{(j)})^{-1}$. Additionally, we have the normal relative velocity which is

$$\mathbf{V}^{(i,j)} = ((\mathbf{V}^{(i)} - \mathbf{V}^{(j)}) \cdot \mathbf{n}^{(i,j)}) \mathbf{n}^{(i,j)}.$$

Note that the time in collision can be computed by

$$t_c = \sqrt{\frac{M^{(i,j)}}{k_c} (\ln^2 e_n + \pi^2)}.$$

This time scale must be resolved in the simulation. As can be seen, this sets practical limitations on the simulation time step.

Thus, for each particle we can write the following system of equations

$$\frac{d\mathbf{Y}^{(i)}}{dt} = \dot{\mathbf{Y}}^{(i)},$$

where

$$\mathbf{Y}^{(i)} = \begin{bmatrix} X^{(i)} \\ Y^{(i)} \\ Z^{(i)} \\ V_x^{(i)} \\ V_y^{(i)} \\ V_z^{(i)} \end{bmatrix}, \quad \dot{\mathbf{Y}}^{(i)} = \begin{bmatrix} V_x \\ V_y \\ V_z \\ (F_{cx}^{(i)} + 0)/M_p^{(i)} \\ (F_{cy}^{(i)} + M_p^{(i)}g)/M_p^{(i)} \\ (F_{cz}^{(i)} + 0)/M_p^{(i)} \end{bmatrix}.$$

User Interface

The *H-File* for this case (DEM Packing 3D H-File) is given below and corresponds to the equations being solved and the property being stored for each particle. Note that since g is constant, we do not included in in the list of properties.

```
#define PPICLF_LRS 6
#define PPICLF_LRP 3
#define PPICLF_LWALL 800

#define PPICLF_JX 1
#define PPICLF_JY 2
#define PPICLF_JZ 3
#define PPICLF_JVX 4
#define PPICLF_JVY 5
#define PPICLF_JVZ 6
#define PPICLF_R_JRHOP 1
#define PPICLF_R_JDP 2
#define PPICLF_R_JVOLP 3
```

The two blocks of lines denote the pre-defined and user-only directives. The pre-defined directives are in the top block and are the number of equations, the number of properties, and the maximum number of boundaries. The user-only directives are in the bottom block.

The *F-File* for this case (DEM Packing 3D F-File) is similar to the *Stokes 2D* exmaple. In `ppiclf_user_SetYdot`, the forces are evaluated. Note that the routine `ppiclf_solve_NearestNeighbor` is invoked which activates the routine `ppiclf_user_EvalNearestNeighbor`. In `ppiclf_user_EvalNearestNeighbor`, the collision force model is applied between the j nearby particles as well as the j nearby boundaries. The collision force is stored in the extra storage array `ppiclf_ydotc`. The other routine `ppiclf_user_MapProjPart` is defined only.

The *External Interface* calls for this example occur in a simple driver program in the file `test.f` with the minimum number of initialization and solve subroutines called. In this case:

- `ppiclf_comm_InitMPI` is called to initialize the communication,
- `ppiclf_comm_InitParticle` is called with initial properites and conditions for the particles,
- `ppiclf_solve_InitNeighborBin` is called with minimum interaction distance of the largest particle size,
- `ppiclf_io_ReadWallVTK` is called which reads the minimal ASCII triangular patch boundary file,
- `ppiclf_solve_IntegrateParticle` is called in a simple time step loop.

Compiling and Running

This example can be tested by issuing the following commands:

```
cd ~
git clone https://github.com/dpzwick/ppiclF.git      # clone ppiclF
mkdir TestCase                                     # make test directory
cd TestCase
cp ../ppiclF/examples/dem_pack_3d/fortran/* .      # copy example files to test_
↪ case
cp -r ../ppiclF/examples/dem_pack_3d/user_routines/ . # copy example files to test_
↪ case
cp -r ../ppiclF/examples/dem_pack_3d/geometry/*.vtk . # copy example files to test_
↪ case
cd ../ppiclF                                       # go to ppiclF code
cp ../TestCase/user_routines/* source/            # copy ppiclf_user.f and PPICLF_
↪ USER.h to source
make                                                # build ppiclF
cd ../TestCase
make                                                # build test case and link with_
↪ ppiclF
mpirun -np 4 test.out                             # run case with 4 processors
```

5.1.3 Nek5000 Case

Background

This is an example that illustrates a two-dimensional fluidized bed. This example also illustrates linking ppiclF to the incompressible, spectral element, fluid solver [Nek5000](#).

The core particle equations being solved in this case are

$$\begin{aligned}\frac{d\mathbf{X}}{dt} &= \mathbf{V}, \\ M_p \frac{d\mathbf{V}}{dt} &= \mathbf{F}_{qs} + \mathbf{F}_c + \mathbf{F}_b,\end{aligned}$$

where, for each particle we have the position vector \mathbf{X} , the velocity vector \mathbf{V} , the drag force \mathbf{F}_{qs} , the collision force \mathbf{F}_c , and the weight force \mathbf{F}_b , defined by

$$\mathbf{X} = \begin{bmatrix} X \\ Y \end{bmatrix}, \quad \mathbf{V} = \begin{bmatrix} V_x \\ V_y \end{bmatrix},$$

and the same weight and collision forces as in the [DEM Packing 3D](#) example are used. For the drag force, the Gidaspow drag model has been used, which is

$$\mathbf{F}_{qs} = \beta V_p (\mathbf{U} - \mathbf{V}) = \beta V_p \begin{bmatrix} U_x - V_x \\ U_y - V_y \end{bmatrix},$$

where \mathbf{U} is the fluid velocity vector interpolated to the particle's coordinates, V_p is each particle's volume, and β is a drag coefficient, computed by

$$\beta = \begin{cases} 150 \frac{\phi_p}{\phi_f} \frac{\mu_f}{D_p^2} + 1.75 \frac{\rho_f}{D_p} |\mathbf{U} - \mathbf{V}| & \phi_p > 0.2, \\ 0.75 C_D^* \frac{\phi_f}{D_p} \rho_f |\mathbf{U} - \mathbf{V}| \phi_f^{-2.65} & \phi_p \leq 0.2. \end{cases}$$

where C_D^* is another drag coefficient, $Re_p^* = \phi_f |\mathbf{U} - \mathbf{V}| D_p / \nu_f$ is the volume weighted Reynolds number, and $\nu_f = \mu_f / \rho_f$ is the fluid kinematic viscosity. The equation for C_D^* is

$$C_D^* = \begin{cases} \frac{24}{Re_p^*} (1 + 0.15 (Re_p^*)^{0.687}) & Re_p^* \leq 10^3, \\ 0.44 & Re_p^* > 10^3. \end{cases}$$

The fluid equations that Nek5000 solves in this example are

$$\begin{aligned} \nabla \cdot \mathbf{u} &= -\frac{1}{\phi_f} \frac{D\phi_f}{Dt}, \\ \rho_f \frac{D\mathbf{u}}{Dt} &= \nabla \cdot \sigma_f + \frac{\mathbf{f}_{pf}}{\phi_f}, \end{aligned}$$

where \mathbf{u} is the fluid velocity, ρ_f is the fluid density, σ_f is the Navier-Stokes fluid stress tensor, ϕ_f is the fluid volume fraction, and ϕ_p is the particle volume fraction ($\phi_f + \phi_p = 1$), and \mathbf{f}_{pf} is a particle-fluid coupling force.

The solution to these equations reside on a mesh within Nek5000. Since the particles are solved in the Lagrangian reference frame, their contributions on the mesh must be accounted for. Most notably, the explicit particle contributions from ppiclF to Nek5000 are ϕ_p (and as a result ϕ_f) and \mathbf{f}_{pf} . The method by which these fields are obtained is *Projection*.

User Interface

The *H-File* for this case (Nek5000 H-File) is given below and corresponds to the equations being solved and the property being stored for each particle. Note that since g is constant, we do not included in in the list of properties.

```
#define PPICLF_LRS 4
#define PPICLF_LRP 6
#define PPICLF_LEE 1000
#define PPICLF_LEX 6
#define PPICLF_LEY 6
#define PPICLF_LRP_INT 3
#define PPICLF_LRP_PRO 3

#define PPICLF_JX 1
#define PPICLF_JY 2
#define PPICLF_JVX 3
#define PPICLF_JVY 4
#define PPICLF_R_JRHOP 1
#define PPICLF_R_JDP 2
#define PPICLF_R_JVLP 3
#define PPICLF_R_JPHIP 4
#define PPICLF_R_JUX 5
#define PPICLF_R_JUY 6
#define PPICLF_P_JPHIP 1
#define PPICLF_P_JFX 2
#define PPICLF_P_JFY 3
```

The two blocks of lines denote the pre-defined and user-only directives. The pre-defined directives are in the top block and are the number of equations, the number of properties, the sizes of the overlap mesh, the number of interpolated fields, and the number of projected fields. The user-only directives are in the bottom block.

The *F-File* for this case (Nek5000 F-File) has meaningful information in every routine. The routine ppiclf_user_SetYdot is nearly the same as the *DEM Packing 3D* example but with an added drag model evaluation that is slightly more complicated than the *Stokes 2D* example. Also, the ppiclf_user_EvalNearestNeighbor routine is

similar to the *DEM Packing 3D* example. The new addition is the mapping of particle properties to be projected in ppiclf_user_MapProjPart. With some study, it can be found that the three fields being projected in 2D are:

Table 1: Projection mapping in ppiclf_user_MapProjPart.

Projected Field ($a(\mathbf{x})$)	Particle Property ($A^{(i)}$)
$\phi_p(\mathbf{x})$	V_p/D_p
$f_{pf,x}(\mathbf{x})$	$-F_{qs,x}/D_p$
$f_{pf,y}(\mathbf{x})$	$-F_{qs,y}/D_p$

where D_p has been used to normalize the values in 2D. Note that the negative signs of the components of \mathbf{F}_{qs} were added when the forces were stored in the storage array ppiclf_ydotc at the end of the routine ppiclf_user_SetYdot.

The *External Interface* calls for this example occur within the user initialization Nek5000 routine usrdat2 in the file `uniform usr` with the minimum number of initialization and solve subroutines called. In this case:

- ppiclf_comm_InitMPI is called to initialize the communication,
- ppiclf_comm_InitParticle is called with initial properites and conditions for the particles,
- ppiclf_solve_InitGaussianFilter is called to initialize the fitler for projection to the overlap mesh,
- ppiclf_comm_InitOverlapMesh is called to initialize the overlap mesh from Nek5000,
- ppiclf_solve_InitNeighborBin is called with minimum interaction distance of the largest particle size,
- ppiclf_solve_InitWall is called which sets a wall for the particles at the bottom of the domain,
- ppiclf_solve_InitPeriodicX is called which sets periodicity in the x dimension along the domain.

Additionally, the solve routines are called every time step in the same file in the Nek5000 routine userchk. In this routine,

- ppiclf_solve_InterpFieldUser is called three times to interpolate the fields ϕ_p , u_x , and u_y into the property array,
- ppiclf_solve_IntegrateParticle is called to integrate the system at the current time step.

Note that the projected fields in the array ppiclf_pro_fld are used to apply the projected forces to the fluid in the Nek5000 routine userf in the same file.

Also, note that ppiclF has been linked with Nek5000 in the Nek5000 makenek compilation file through the following lines:

```
SOURCE_ROOT_PPICLF=$HOME/libraries/ppiclF/source
FFLAGS=" -I$SOURCE_ROOT_PPICLF"
USR_LFLAGS+=" -L$SOURCE_ROOT_PPICLF -lppiclF"
```

Compiling and Running

This example can be tested with Nek5000 by issuing the following commands:

```
cd ~
git clone https://github.com/dpzwick/ppiclF.git           # clone ppiclF
git clone https://github.com/Nek5000/Nek5000.git         # clone Nek5000
mkdir TestCase                                           # make test directory
cd TestCase
cp -r ../ppiclF/examples/Nek5000/* .                   # copy example files to_
↪test case
cd ../ppiclF                                             # go to ppiclF code
cp ../TestCase/user_routines/* source/                 # copy ppiclf_user.f and_
↪PPICLF_USER.h to source
```

(continues on next page)

(continued from previous page)

```
make                                     # build ppiclF
cd ../TestCase
./makenek uniform                       # build Nek5000 and link_
↪with ppiclF
echo uniform > SESSION.NAME && echo `pwd`/ >> SESSION.NAME # create Nek5000 necessary_
↪file
mpirun -np 4 nek5000                    # run case with 4_
↪processors
```


CONTRIBUTING

6.1 Contributing

6.1.1 Questions

6.1.2 Workflow

In order to contribute to ppiclF, a fork/branch workflow is used. First, [ppiclF](#) must be forked to your personal github account.

When additions are made to ppiclF, a pull request can be issued. [Travis](#) will then run through unit tests to make sure that your additions have not broken the core code functionality. An administrator of ppiclF will then only approve your pull request if the unit tests succeed.

6.1.3 Issues

Issues can be submitted through the [issues tab](#) on the main github page.

ACKNOWLEDGEMENTS

7.1 Acknowledgements

7.1.1 Funding

Disclaimer

The views and opinions expressed in this work are those of the authors only and do not necessarily reflect the official policy or position of any funding agency.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1315138.

This work was also supported in part by the US Department of Energy, National Nuclear Security Administration, Advanced Simulation and Computing Program, as a Cooperative Agreement under the Predictive Science Academic Alliance Program, under Contract No. DE-NA0002378.

7.1.2 Contributions

There have been numerous suggestions and reworking of ppicIF over the past few years, including valuable input from the [Nek5000](#) and the [UF-CCMT](#) teams.

Recommended reading

Zwick, D. *Scalable highly-resolved Euler-Lagrange multiphase flow simulation with applications to shock tubes*. Dissertation. University of Florida. 2019.
