Slurm: fluid particle-in-cell code for plasma modeling

V. Olshevsky\textsuperscript{a,b,*}, F. Bacchini\textsuperscript{a}, S. Poedts\textsuperscript{a}, G. Lapenta\textsuperscript{a}

\textsuperscript{a}Centre for mathematical Plasma Astrophysics (CmPA), KU Leuven. Celestijnenlaan 200B, bus 2400, B-3001 Leuven, Belgium
\textsuperscript{b}Main Astronomical Observatory, 27 Akademika Zabolotnoho St. 03143, Kyiv, Ukraine

Abstract

With the approach of exascale computing era, particle-based models are becoming the focus of research due to their excellent scalability. We present a new code, Slurm, which implements the classic particle-in-cell algorithm for modeling magnetized fluids and plasmas. It features particle volume evolution which damps the numerical finite grid instability, and allows modeling of key physical instabilities such as Kelvin-Helmholtz and Rayleigh-Taylor. The magnetic field in Slurm is handled via the electromagnetic vector potential carried by particles. Numerical diffusion of the magnetic flux is extremely low, and the solenoidality of the magnetic field is preserved to machine precision. A double-linked list is used to carry particles, thus implementation of open boundary conditions is simple and efficient. The code is written in C++ with OpenMP multi-threading, and has no external dependencies except for Boost. It is easy to install and use on multi-core desktop computers as well as on large shared-memory machines. Slurm is an ideal tool for its primary goal, modeling of space weather events in the heliosphere. This article walks the reader through the physical model, the algorithm, and all important details of implementation. Ideally, after finishing this paper, the reader should be able to either use Slurm for solving the desired problem, or create a new fluid PIC code.

Keywords: particle-in-cell, magnetohydrodynamics, plasma simulations, computational fluid dynamics

\textsuperscript{*}Corresponding author

\textit{Email addresses:} vyacheslav.olshevsky@kuleuven.be (V. Olshevsky), fabio.bacchini@kuleuven.be (F. Bacchini), stefaan.poedts@kuleuven.be (S. Poedts), giovanni.lapenta@kuleuven.be (G. Lapenta)

Preprint submitted to Computer Physics Communications June 26, 2018
1. Motivation

Particle-in-cell (PIC) method was first proposed by [Harlow 1964] for modeling compressed fluids. PIC combines particles which follow material motion and carry conserved quantities such as mass and momentum, with a grid on which the equations of motion are solved. Computational particles in PIC could be considered as a moving refined grid, hence the method is useful for modeling highly distorted flows and interface flows. Early implementations were cumbersome (especially in treating boundary conditions), memory-hungry, and unable to treat certain physical problems such as, e.g., stagnating flows, which made the method obsolete. However, in the last two decades the method got its second wind thanks to the Cloud-in-Cell (CIC) algorithm [Langdon and Birdsall 1970] for kinetic modeling of plasmas, and the Material Point Method (MPM) applied in solid mechanics [Sulsky et al. 1995]. It is common nowadays to treat PIC solely as a method for kinetic plasma modeling, therefore narrowing its scope only to CIC [Brackbill 2005]. Both MPM and CIC delivered impressive new results and are further developing; we refer the reader to the reviews of the applications of CIC in space plasma modeling [Lapenta 2012]; MPM in geophysics [Sulsky et al. 2007; Fatemizadeh and Moormann 2015], engineering [York et al. 2000], and 3D graphics [Stomakhin et al. 2013].

Motivated by the above success we proposed a new PIC algorithm for magnetized fluids [Bacchini et al. 2017] which clarified and simplified many aspects of the original FLIP-MHD algorithm by [Brackbill 1991]. This article discusses the realization of our algorithm, Slurm, implemented in C++ with OpenMP multi-threading. We will walk you through all steps of the typical computational cycle of Slurm, its initialization, and main solver loop, explaining, where appropriate, data structures, architecture and physical models used in the code.

2. Physical model

2.1. MHD equations

Slurm is designed (but not limited) to numerically advance in time a discretized set of conventional MHD equations in Lagrangian formulation...
which describe the dynamics of plasmas or magnetized fluids

\[
\frac{d\rho}{dt} = -\rho \frac{\partial u_i}{\partial x_i},
\]

\[
\rho \frac{du_j}{dt} = -\frac{\partial p}{\partial x_j} + \rho g_j + \frac{\partial}{\partial x_i} \left[ B_i B_j - \frac{1}{2} B^2 \delta_{ij} \right] + \frac{\partial}{\partial x_i} \left[ \mu \left( \frac{\partial u_j}{\partial x_i} + \frac{\partial u_i}{\partial x_j} \right) + \left( \mu_v - \frac{2}{3} \mu \right) \frac{\partial u_i}{\partial x_i} \delta_{ij} \right],
\]

\[
\rho \frac{de}{dt} = -p \frac{\partial u_i}{\partial x_i} + \mu \left( \frac{\partial u_j}{\partial x_i} + \frac{\partial u_i}{\partial x_j} - \frac{2}{3} \frac{\partial u_i}{\partial x_i} \delta_{ij} \right)^2 + \mu_v \left( \frac{\partial u_i}{\partial x_i} \right)^2 + \eta J^2,
\]

where summation is implied over the repeating indices \(i, j, l\), which denote vector components; \(d/dt = \partial/\partial t + u_j \partial/\partial x_j\) is the convective derivative. The plasma is described by density \(\rho\), velocity \(u\), pressure \(p\), internal energy per unit mass \(e\), and magnetic field \(B\); \(g\) represents external forces (gravity); \(\mu = \nu \rho\) is the dynamic shear viscosity and \(\mu_v = \nu_v \rho\) is the dynamic bulk viscosity; \(\eta\) is the resistivity, and \(\mathbf{J} = \nabla \times \mathbf{B}\) is electric current; \(\delta_{ij}\) is the Kronecker delta.

In most cases reported in this paper, the adiabatic equation of state is assumed

\[
e = \frac{1}{\rho} \frac{p}{\gamma - 1},
\]

where \(\gamma\) is the predefined gas constant.

### 2.2. Magnetic field equation

To close the above system of MHD equations, a condition must be imposed on the evolution of \(B\). As a proxy for magnetic field, Slurm uses electromagnetic potentials \(\mathbf{A}\) and \(\phi\), where

\[
\mathbf{B} = \nabla \times \mathbf{A}.
\]

Using vector potential as a proxy for magnetic field, on a staggered grid (Yee lattice), the \(\nabla \cdot \mathbf{B} = 0\) condition is satisfied to machine precision \cite{Bacchini2017}. The staggered grid also makes our method second-order accurate in space. In Slurm, the magnetic field is specified on the grid’s cell centers, and electromagnetic vector potential is given on cell vertices (nodes).

\footnote{the notation and terminology follows the excellent book by Kundu et al. (2012)}
The evolution of $\mathbf{A}$ is governed by the equation

$$\frac{d\mathbf{A}}{dt} = \mathbf{u} \times \mathbf{B} + (\mathbf{u} \cdot \nabla) \mathbf{A} - \nabla \phi - \eta \mathbf{J},$$

(6)

where $\phi$ can be chosen freely to satisfy a certain gauge condition. Although it has no effect on the solenoidality of the magnetic field, appropriate measures have to be taken to preserve the gauge condition throughout the simulation. [Tricco and Price (2012); Tricco et al. (2016)] have introduced the constrained hyperbolic divergence cleaning of $\nabla \cdot \mathbf{B}$ in smoothed particle hydrodynamics, which was used later by [Stasyszyn and Elstner (2015)] to clean the divergence of $\mathbf{A}$. According to this strategy, the scalar potential $\phi$ is evolved as

$$\frac{d\phi}{dt} = -c_h^2 \nabla \cdot \mathbf{A} - \sigma c_h \phi,$$

(7)

where $c_h$ is the fast MHD wave speed, $h$ is the smoothing length (particle size), and $\sigma \approx 1$ is a constant. This way, the Coulomb gauge $\nabla \cdot \mathbf{A} = 0$ or any other initial gauge could be preserved throughout the simulation.

Interestingly, in two dimensions

$$B_x = \frac{\partial A_z}{\partial y},$$

(8)

$$B_y = -\frac{\partial A_z}{\partial x},$$

(9)

and by definition

$$\frac{dA_z}{dt} = 0,$$

(10)

hence $A_z$ defined on a moving grid (particles), does not change over time ([Bacchini et al., 2017]). Therefore in 2D (with no out-of-plane $B$ field) the gauge condition is preserved to roundoff, and no divergence cleaning is necessary.

2.3. Artificial viscosity

The PIC method has very low numerical dissipation, and in many problems related to shock handling a numerical bulk viscosity $\mu_v$ is useful to stabilize the solution. Among many possible options we had tested, we obtained satisfactory results with the Kuropatenko’s form of artificial viscosity.
This viscosity is non-zero only in grid cells for which \( \nabla \cdot \mathbf{u} > 0 \). As given by Caramana et al. (1998),

\[
\mu_v = \rho \left[ c_2 \frac{\gamma + 1}{4} |\Delta \mathbf{u}| + \sqrt{c_2^2 \left( \frac{\gamma + 1}{4} \right)^2 (\Delta \mathbf{u})^2 + c_1^2 c_s^2} \right] |\Delta \mathbf{u}|, \tag{11}
\]

where \( |\Delta \mathbf{u}| = |\Delta u_x + \Delta u_y + \Delta u_z| \) is the velocity jump across the grid cell, \( c_s = \sqrt{\gamma p/\rho} \) is the adiabatic sound speed, \( c_1 \) and \( c_2 \) are constants close to 1.

### 3. Main classes and data structures

Traditional implementations of PIC (or, rather, CIC) algorithms use conventional domain decomposition for parallelizing the computation which, obviously, can lead to strong processor load imbalance. Different strategies were proposed to overcome it, such as particle splitting/merging (Beck et al., 2016) or grid refinement and temporal sub-stepping (Innocenti et al., 2015). Particle methods, instead, provide excellent opportunity for task or event based parallel approach. The latter has been discussed by, e.g., Karimabadi et al. (2005), but no productive implementation has been reported yet, to our knowledge.

Slurm embraces a task based approach in which all particle operations are split between a user-defined number of OpenMP threads. Distributing tasks, not computational sub-domains, is advantageous in problems with strong density imbalances. Particles and computational grid exchange information several times during each computational cycle, but otherwise grid operations and particle operations are independent. Therefore two main classes, \texttt{Grid} and \texttt{ParticleManager}, were chosen to manage grid elements (cells and nodes) and particles, respectively.

#### 3.1. Basic players: cells, nodes, and particles.

We use a staggered grid, therefore three types of basic entities are represented by the \texttt{GridCell}, \texttt{GridNode}, and \texttt{Particle} classes, which share some common properties as illustrated in Fig. 1. Particles carry the following physical quantities: volume \( V \), mass \( m \), momentum \( m \mathbf{u} \), internal energy \( \epsilon = em \), and electromagnetic potentials \( \mathbf{A} \) and \( \phi \) (the latter is solely used to preserve the gauge of \( \mathbf{A} \)). Each grid element is assigned a mass \( m \) and a volume \( V \), hence the density is computed (upon interpolation from particles)
as \( \rho = m/V \), for both nodes and cells. Grid nodes keep the fluid velocity \( \mathbf{u} \) and the vector potential \( \mathbf{A} \), while grid cells keep the internal energy density \( \epsilon \), the magnetic field \( \mathbf{B} \) and the scalar potential \( \phi \). In addition, all grid elements carry connectivity information as explained below.

3.2. Grid

The \texttt{Grid} object is initialized by the solver first. Slurm supports rectilinear grids with \( n_{xc} \times n_{yc} \times n_{zc} \) cells in each dimension, and \( n_{xn} \times n_{yn} \times n_{zn} \) nodes in each dimension \( (n_{x,y,z,n} = n_{x,y,z,c} + 1) \). At each boundary, there is one layer of ghost cells, one layer of ghost nodes. Each cell and each node stores information about its role (‘general’, ‘boundary’, or ‘ghost’) and pointers to its neighbors. Besides geometrical neighbors, logical neighbors are also computed.

Logical neighbors denote the grid elements which must be used for interpolating to/from this grid element. For instance, a ‘logical neighbor node’ of a boundary cell on a periodic boundary is the node on the opposite boundary. The number of particles is much larger than the number of grid elements,
hence connectivity information doesn’t lead to excessive memory use. However, it saves a substantial amount of resources and provides capabilities for implementing irregular grids.

3.3. Particle Manager

Class \texttt{ParticleManager} stores a double-linked list of \texttt{Particle} objects, which makes adding and removing particles rather efficient and trivial to implement. The list could easily be processed in parallel using OpenMP directives, e.g.,

```c
#pragma omp parallel
{
    int thread_num = omp_get_thread_num();

    // Make sure particles were divided between threads according to the current number of threads
    if (omp_get_num_threads() != IteratorsNumberOfThreads) {
        if (thread_num == 0)
            IteratorsNumberOfThreads = omp_get_num_threads();

        #pragma omp barrier
        computeOMPThreadRange();
    }

    // Process particles which belong to this thread.
    for(ParticleIterator pi = IteratorsBegin[thread_num]; pi != IteratorsEnd[thread_num]; ++pi)
    {
        // Do something
    }
}
```

where \texttt{IteratorsNumberOfThreads} is the number of threads for which the particle ranges \texttt{IteratorsBegin} and \texttt{IteratorsEnd} were computed. All expensive procedures: interpolation from particles to grid, interpolation from grid to particles, and particle push, are handled by the \texttt{ParticleManager} object, and are parallelized in the above manner.

Strong scaling tests performed on 36-core Intel Xeon processor E5-2600 v4 (Broadwell) have shown that parallel performance improves with the number
of particles per cell (Figure 2). To compute the parallel speedup, we ran simulations on the same $32^3$ grid, increasing the number of threads from 1 to 36. The parallel speedup equals to the ratio of the average time of one computational cycle on one thread to the time on multiple threads $s(N) = \frac{\langle T(1) \rangle}{\langle T(N) \rangle}$. On 36 threads the code executes 18 times faster than on a single thread, i.e., parallel efficiency reaches 50%. Note, the code has neither been optimized, nor tuned for performance. We used GNU C++ compiler 6.1 with the following options: `-std=c++11 -O3 -fopenmp -lpthread`.

4. Main solver loop

The main solver loop of an explicit PIC algorithm consists of four steps: (1) interpolation particles→grid, (2) grid advancement, (3) interpolation grid→particles, and (4) particle mover, as illustrated in Figure 3. In a typical PIC simulation the number of computational particles far exceeds the number of grid cells, and traversing all particles is the most time-consuming operation in the cycle. In Slurm, particles are only traversed twice per cycle: first time in step (1) to interpolate to the grid. The second loop invokes both grid→particle interpolation and particle push, therefore steps (3) and (4) are combined. In the same loop particles are confronted against the boundary.

![Figure 2: Strong scaling on Intel Xeon E5-2600 v4 (Broadwell). The black line denotes ideal scaling; each color line corresponds to the number of particles per cell: from 27 to 216. The more particles per cell, the better the scaling is.](image-url)
conditions, and are marked for deletion if needed.

4.1. Step 1. Interpolation from particles to grid.

Projection of data from particles to grid begins with setting the corresponding quantities \((m, V, e, \phi)\) for cells and \(m, V, u, A\) for nodes to zero. Interpolation is implemented in the ParticleManager class in one loop over all particles. It fetches the required information from each particle and adds the corresponding contribution to the surrounding grid nodes and grid cells. When all particles have been processed, the accumulated values of the physical quantities on each grid element are normalized to the total interpolated weight at that grid element.

Information from each particle \((\epsilon, m, V, V\phi)\) is projected onto 27 cells:
the cell that encloses this particle, and 26 cells that share a face or a vertex with it. Interpolation weights are given by the second order b-spline

\[
w_{c,x} = \begin{cases} 
\frac{1}{2} x^2 - \frac{3}{2} \frac{|x|}{\Delta x} + \frac{9}{8} & \frac{1}{2} \leq |x| < \frac{3}{2} \\
-x^2 + \frac{3}{4} & 0 \leq |x| < \frac{1}{2}
\end{cases},
\]

where \( x = (x_c - x_p)/\Delta x \) is the difference between the cell center coordinate \( x_c \) and the particle’s coordinate \( x_p \) normalized to the cell’s extent \( \Delta x \). The total weight is the product of the three weights in each dimension

\[
w_c = w_{c,x} \cdot w_{c,y} \cdot w_{c,z}
\]

Conserved quantities to be projected on the nodes include \( \mathbf{u}, m, V \). These are interpolated onto 8 vertices (nodes) of the cell which encloses this particle. To compute interpolation weights, the conventional first order b-spline is used as described in Bacchini et al. (2017)

\[
w_n = (1 - |x_n - x_p|) \cdot (1 - |y_n - y_p|) \cdot (1 - |z_n - z_p|),
\]

where \( x_n, y_n, z_n \) are the coordinates of this vertex (node). A different procedure used to interpolate \( \mathbf{A} \) from particles to grid.

Electromagnetic vector potential is not an additive conserved quantity, but rather a smooth vector function of three coordinates. We interpolate \( \mathbf{A} \) to each grid node from an arbitrary hexahedron formed by the node’s closest particles. Each grid node keeps information about the eight closest particles, hence the interpolation is realized in the \texttt{Grid} class. It is called when all conserved quantities have been projected. Interpolation from irregular grids is quite straightforward in 2D, but in three dimensions it is more sophisticated, and requires the iterative solution of a nonlinear system. For the sake of completeness, we provide the detailed description of the interpolation procedure adopted from NASA’s b4wind User’s Guide in Appendix A. Note, the same interpolation approach may be used for projecting the scalar potential \( \phi \) on the cells, however none of our tests indicated that this is beneficial.

\[\text{https://www.particleincell.com/2012/quad-interpolation}\]
\[\text{https://www.grc.nasa.gov/WWW/winddocs/utilities/b4wind_guide/trilinear.html}\]
4.2. Step 2. Advance the grid

4.2.1. Boundary conditions

Boundary conditions should be applied to all quantities interpolated from particles, and to all derived quantities such as \( B \) and \( J \). All ghost and boundary grid cells and nodes carry the references to the applicable \texttt{BoundaryCondition} objects. They implement methods which modify the given cell or node quantities according to the specific rule. For instance, in reflective boundary

```cpp
// class BoundaryConditionReflective : public BoundaryCondition
void BoundaryConditionReflective::imposeOnNode(GridNode * n, SlurmDouble t)
{
    for (SlurmInt i = 0; i < 3; i++)
        if (i == normal)
            n->set_u(normal, 0);
        else
            n->set_u(i, 2 * n->get_u(i));

    n->set_m(2 * n->get_m());
}
```

4.2.2. Directional derivatives

To evolve the equations of MHD in time, and to compute certain derived quantities such as \( B, J \), etc., the gradients or directional derivatives of the discretized quantities should be computed. In Slurm, a directional derivative of a node quantity \( q \) in the direction \( x \) is cell-centered. It is computed as the average over the four corresponding edges of that cell. Using three indices \( i, j, k \), corresponding to \( x, y, z \) coordinate dimensions, we designated the node-centered value as \( q_{ijk} \), where each index is either 0 for the ‘left’ (bottom, front) or 1 for the ‘right’ (top, back) corner of the given cell. Then the discrete directional derivatives are equal to

\[
\frac{\partial q}{\partial x} = \frac{q_{100} - q_{000} + q_{110} - q_{101} + q_{001} + q_{111} - q_{011}}{4\Delta x}, \tag{15}
\]

\[
\frac{\partial q}{\partial y} = \frac{q_{010} - q_{000} + q_{110} - q_{100} + q_{001} + q_{111} - q_{011}}{4\Delta y}, \tag{16}
\]

\[
\frac{\partial q}{\partial z} = \frac{q_{001} - q_{000} + q_{101} - q_{100} + q_{011} + q_{111} - q_{101}}{4\Delta z}. \tag{17}
\]
where \( \Delta x, \Delta y, \) and \( \Delta z \) are the cell’s extents in the corresponding dimension. The gradients of cell-centered quantities are node-centered. They are computed in exactly the same way, except that \( \Delta x, \Delta y, \) and \( \Delta z \) should represent the distances between the corresponding neighbor cell centers. This way, the second derivative of a node quantity is node-centered, and the second derivative of a cell quantity is cell-centered.

4.2.3. Solution of MHD equations

A semi-implicit scheme is used to march the MHD equations on the grid in time. First, from the cell-centered properties at the previous time step \( n \), the new velocity is computed on each node.

\[
\mathbf{u}^{n+1} = \mathbf{u}^n + \Delta t Q^n,
\]

where \( Q^n \) is the right-hand side of the discretized momentum equation 2 computed with the ‘old’ grid values. Other grid properties are advanced using the new velocity according to discretized equations 1, 3, 6, 7 after the boundary conditions are imposed on \( \mathbf{u}^{n+1} \)

\[
q^{n+1} = q^n + \Delta t Q \left( \mathbf{u}^{n+1}, \rho^n, p^n, \mathbf{A}^n, \phi^n \right),
\]

where \( q \) is one of \( \rho, e, \mathbf{A}, \) or \( \phi \), and \( Q \) is the right-hand side of the corresponding MHD equation. Note, boundary conditions need to be taken care of before interpolating the updated information to the particles.

The numerical scheme described above is a special case of the implicit scheme analyzed by [Brackbill and Ruppel 1986]. It is stable when the Courant condition is met, and offers better accuracy than a simple forward Euler method.

4.3. Steps 3 and 4. Interpolate on, and push the particles

The weights used to interpolate from grid to particles are the same as those used for particles to grid projection (Section 4.1). From grid cells to particles we interpolate the change of internal energy \( \Delta \epsilon = \epsilon^{n+1} - \epsilon^n \), the cell-centered velocity gradient \( \nabla \mathbf{u}^{n+1} \), and the change of scalar potential \( \Delta \phi = \phi^{n+1} - \phi^n \). From grid nodes, the following quantities are interpolated: \( \mathbf{u}^{n+1}, \Delta \mathbf{u} = \mathbf{u}^{n+1} - \mathbf{u}^n, \) and \( \Delta \mathbf{A} = \mathbf{A}^{n+1} - \mathbf{A}^n \).
Physical quantities carried by the \( p \)–th particle are updated as

\[
\epsilon_p^{n+1} = \epsilon_p^n + m_p \Delta \epsilon, \quad (20)
\]
\[
u_p^{n+1} = u_p^n + \Delta u, \quad (21)
\]
\[
A_p^{n+1} = A_p^n + \Delta A, \quad (22)
\]
\[
\phi_p^{n+1} = \phi_p^n + \Delta \phi. \quad (23)
\]

The velocity gradient is used to advance the particle’s volume according to either the strategy described by [Bacchini et al. (2017)], or in a simpler way:

\[
V_p^{n+1} = V_p^n \left( 1 + \Delta t \text{Tr} \nabla \mathbf{u}^{n+1} \right), \quad (24)
\]

where

\[
\text{Tr} \nabla \mathbf{u}^{n+1} \equiv \nabla \cdot \mathbf{u}^{n+1} = \nabla u_{11}^{n+1} + \nabla u_{22}^{n+1} + \nabla u_{33}^{n+1} \quad (25)
\]

is the trace of the interpolated velocity gradient tensor. Note that particle quantities are updated using the interpolated changes in the grid quantities \( (\Delta q) \), which ensures very low numerical diffusivity of the method. The interpolated velocity \( \mathbf{u}^{n+1} \) is used only to advance the particle’s position

\[
r^{n+1} = r^n + u^{n+1} \Delta t. \quad (26)
\]

Once the new position is found, it is checked against boundary conditions. If necessary, the particle is either reflected from the wall, flipped around the periodic boundary, or marked for deletion if it crosses an open boundary.

When all particles have been updated, the list of particles is walked through one more time, and all particles marked for deletion are removed. Finally, if necessary, new particles are injected and the pre-computed particle ranges for each thread are updated.

5. Examples

Most testcases considered in this section could be directly compared with an excellent paper on Athena code validation by [Stone et al. (2008)]. Where appropriate, we provide references to the corresponding figures in this paper.
5.1. Two interacting blast waves

A simple one-dimensional 1D shock tube proposed by Woodward and Colella (1984) is widely used to study the shock-capturing and stability properties of the codes. In this problem, the initial state possesses a uniform density $\rho = 1$, specific heats ratio $\gamma = 1.4$, zero velocity, and three different pressures. In the leftmost tenth of the domain $p = 1000$, in the rightmost tenth $p = 100$, and in between $p = 0.01$. The shock tube evolves quickly and produces multiple shock waves at high Mach numbers which are reflected from the walls and interact with each other. This test is very hard to handle for Eulerian codes, however Slurm deals with it rather easy. Step-by-step description of the complex evolution could be found in Woodward and Colella (1984), we only present the final result at time $t = 0.038$.

We performed two runs, with 2400 and 200 cells; both with 40 particle/cell, and the same $\Delta t = 4 \cdot 10^{-6}$. The CFL condition computed on the fine grid using the largest initial value of the sound speed $c_s = \sqrt{\gamma p/\rho} = 37.4$ suggests $\Delta t < 1.1 \cdot 10^{-5}$. The density and velocity profiles at $t = 0.038$ in both simulations are shown in Figure 4. These plots can be compared to the reference solutions in Figure 2h of the original paper (Woodward and Colella, 1984). Even at low grid resolution (200 cells), the contact discontinuity at $x = 0.6$ (Fig. 4b) is very sharp. However, even at high resolution there is an artificial spike at density discontinuity at $x = 0.8$. This spike neither grows, nor is a source of fictitious oscillations, even when the simulation is run.
Figure 5: Simulations of Brio & Wu shock tube at two resolutions, with 3200 cells (solid line), and with 400 cells (crosses), at $t = 0.1$.

Further. More sophisticated forms of hyperviscosity might be needed to get rid of it in our simple explicit first-order numerical scheme.

5.2. Brio & Wu shock tube

Brio and Wu (1988) proposed an extension of the Sod shock tube to MHD. There are two initial states in the initial domain with $L_x = 1$: the left state has $\rho = 1$, $p = 1$, and $B_y = 1$; the right state has $\rho = 0.125$, $p = 0.1$, and $B_y = -1$. In the whole domain, $B_x = 0.75$, $\gamma = 2$, and all other quantities are zero. In terms of vector potential, the initial field is given by

$$A_z = \begin{cases} 
-x & 0 \leq x < 0.5 \\
 x - 1 & 0.5 \leq x \leq 1 
\end{cases}$$

At each computational cycle, when $B$ is computed on grid cells from the interpolated $A$ according to Eq. 5, $B_x = 0.75$ is added in each cell.

The aim of this test is to check how well the code distinguishes and handles different MHD shocks. We present here the results of two runs: with
Figure 6: Orszag-Tang vortex simulation at $t = 0.5$. Left: pressure along two cuts, at $y = 0.3125$ (a) and at $y = 0.4277$ (b) in the simulations with $256 \times 256$ cells (solid) and $128 \times 128$ cells (crosses). Right: magnetic field amplitude. This figure may be compared with Figures 5 and 6 in Stasyszyn et al. (2013), and Figure 23 in Stone et al. (2008).

3200 cells, 9 particles/cell, $\Delta t = 10^{-5}$, and with 400 cells, 36 particles/cell, $\Delta t = 10^{-4}$. In Figure 5, the plots of $\rho$, $B_y$, $u_x$, and $u_y$ are shown at $t = 0.1$. These plots can be compared with Figure 2 of the original paper (Brio and Wu, 1988). Slurm accurately reproduces two fast rarefaction waves (FR), a slow compound wave (SM), a contact discontinuity (C), and the slow shock (SS). There is still some numerical noise around the slow shock boundary, similar to a density spike in the interacting blast waves simulation. This noise is efficiently damped by the Kurotatenko’s hyper-viscosity, and is not affecting other cells of the grid. It neither grows in amplitude, nor propagates further when the simulation runs well beyond the $t = 0.1$ time. In this simulation, high spatial resolution is essential to correctly handle the shocks.

5.3. Orszag-Tang

Orszag-Tang vortex represents a complex interaction of different MHD shocks in 2D, with a translation to MHD turbulence, and is often used as a reference test case for MHD code validation. Initial pressure and density in a periodic domain with $L_x = L_y = 1$ are uniform, $\rho = 25/\left(36\pi\right)$, and $p = 5/\left(12\pi\right)$. Initial velocity components are $u_x = -\sin \left(2\pi y\right)$ and $u_y = \sin \left(2\pi x\right)$. The magnetic field is given by $A_z = (B_0/4\pi) \cos \left(4\pi x\right) +$
Figure 7: Snapshots of Rayleigh-Taylor instability simulation at time moments $t = 5$, $t = 10$, $t = 12.5$, and $t = 15$. Each little cube corresponds to a computational particle (only 1 out of 27 particles are saved). Purple particles represent the heavy fluid, initially on top of the lighter fluid (green particles), and completely mixed at the end of the simulation. The $3/4$ of the box in front of each snapshot have been clipped to visualize the motion inside the domain.

$$(B_0/2\pi) \cos (2\pi y)$$

Simulations at two grid resolutions, with $256 \times 256$ cells, $\Delta t = 10^{-4}$, and with $128 \times 128$ cells, $\Delta t = 5 \cdot 10^{-4}$, both using 25 particles/cell, are compared in Figure 6. Panels on the left represent gas pressure along two cuts, $y = 0.3125$ (a) and at $y = 0.4277$ (b), and may be used for quantitative comparison with reference results provided in Figure 23 of [Stone et al., 2008], and in Figure 6 of [Stasyszyn et al., 2013]. Slurm simulation with $256 \times 256$ grid cells reproduces all major features of the reference solutions, except a few sharpest shock interfaces. This is also confirmed by a snapshot of the magnetic field amplitude $B$ shown in the right panel of Fig. 6. This image could be compared with Fig. 5 in [Stasyszyn et al., 2013].

5.4. Rayleigh-Taylor

A Rayleigh-Taylor instability, together with Kelvin-Helmholtz instability are the key ingredients in fluid modeling of many astrophysical phenomena, from solar convection to the solar wind-magnetosphere interaction. As shown by [Bacchini et al., 2017], particle volume evolution enables Slurm to successfully model the Kelvin-Helmholtz instability, even with explicit time-stepping. Rayleigh-Taylor instability simulated by different numerical codes was compared by [Liska and Wendroff, 2003]. There is no unique so-
olution to the problem in the nonlinear regime, therefore we do not attempt to reproduce results of other codes. Instead, we use the three-dimensional Rayleigh-Taylor problem to test the reflecting boundaries of our code, the symmetry of the solution, and demonstrate its unique capability to study fluid mixing. No pure grid-based code is able to track individual fluid particles and thus be self-consistently used to investigate mixing of different fluids or plasmas.

The computational domain is a $0.5 \times 0.5 \times 1.5$ box with periodic X and Y boundaries, and reflective Z boundaries. In the top half of the box the fluid is heavier with $\rho = 2$, while in the bottom half $\rho = 1$. Gravity acceleration $g_z = -0.1$ and specific heats ratio $\gamma = 1.4$ are constant throughout the domain. The initial pressure is given by the hydrostatic equilibrium $p(z) = 2.5 + g_z \rho z$. The instability is excited by a single-mode velocity perturbation of the form

$$u_z = u_0 \left( 1 + \cos \left( 4\pi \left( x - 0.25 \right) \right) \right) \left( 1 + \cos \left( 4\pi \left( y - 0.25 \right) \right) \right) \left( 1 + \cos \left( 3\pi \left( z - 0.75 \right) \right) \right),$$

where $u_0 = 0.01$.

Figure 7 shows the nonlinear evolution of the instability simulated in a $32 \times 32 \times 96$ grid with 27 particles/cell and $\Delta t = 10^{-3}$. The flow exhibits classical features of the Rayleigh-Taylor instability with secondary Kelvin-Helmholtz instabilities and growing turbulence on the edges of the flow. Thanks to the computational particles we can catch these features even at rather low grid resolution, and also track how initially different fluids mix together.

5.5. 3D magnetic loop

As mentioned above, in 2D simulations Slurm preserves magnetic topology exactly (Eq. 10), and two-dimensional magnetic loop test is only useful to track possible bugs. Here we show that Slurm deals with a 3D magnetic loop as easy. In this test, the electromagnetic vector potential inside a magnetic loop, inclined by $45^\circ$ to the vertical axis, is given by $A_x = A_z = A_0 (r - R)$, and is zero outside the loop, $r > R$, where $r$ is the distance to the loop’s axis, $R = 0.3$ is the loop radius, and $A_0 = 0.001/\sqrt{2}$. The domain’s extent is 1 in all dimensions, and the loop is advected in all three dimensions with speed $u_x = u_y = u_z = \sqrt{3}$. Loop’s magnetic field strength is very small to ensure high plasma beta and the absence of pressure imbalance effects.

The results of the simulation in a $100^3$ cells grid with 27 particle/cell are shown in Fig. 8. The loop is formed by two current sheets: one infinitely thin
Figure 8: Current density magnitude $J = |\nabla \times \mathbf{B}|$ in the beginning (left), during boundary crossing (in the middle), and after 2 crossings.

(as thin as the numerical scheme allows) on the axis, and one cylindrical on the outer boundary of the loop. The color in the figure depicts the current density $J$ in the loop. The right snapshot, taken after two crossings, is indistinguishable from the left snapshot which is taken in the beginning of the simulation, hence Slurm perfectly preserves magnetic topology, electric current and magnetic energy of the advected loop. This figure could be compared with Figure 34 of Stone et al. (2008).

6. Summary

We have successfully implemented and tested a particle-in-cell MHD model, Slurm. Several features distinguish it from all previous (known) implementations of fluid PIC:

- Magnetic field evolution strategy. Fluid particles carry electromagnetic potential $\mathbf{A}$ which is projected on grid nodes from the closest particles. This way, solenoidality of magnetic field is ensured to machine precision, there is no diffusion of magnetic energy and magnetic topology is preserved accurately.

- Particle volume evolution is taken into account. Volume is used to normalize the quantities which are interpolated between the grid and the particles. Volume evolution damps numerical ringing instability, and allows us to model key hydrodynamic instabilities such as Kelvin-Helmholtz and Rayleigh-Taylor.
Efficient bulk hyper-viscosity allows to resolve shocks and accurately model reference problems.

Particles are carried as a double-linked list of objects. Particles can efficiently be deleted and inserted on-the-fly without reducing the code’s performance. This is particularly important for space weather simulations, where open boundaries are crucial.

The code uses task-based parallel approach in which particles are split between multiple OpenMP threads operating in shared memory.

Future developments in Slurm include open boundary conditions in complex physical models such as solar wind; optimizations of parallel performance and additional parallelization for non-shared memory systems. Slurm is fully functional and tested, and is available online at bitbucket. Although Slurm’s primary application is envisioned in space weather modeling, the range of problems for which it could be suitable include interface flows, surface flows, fluid mixing, etc.

Acknowledgements

This work is conducted under the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0375. V.O. and F.B. are thankful to Craig DeForest for useful discussions during the 2017 AFOSR $B_z$ meeting.

Appendix A. Interpolation from irregular grid in 3D

The algorithm of interpolation from 8 vertices of a hexahedron (formed by the node’s closest particles) onto one enclosed point (this grid node) is adopted from NASA’s b4wind User’s Guide. The real coordinates of the vertices of an arbitrary hexahedron should be translated into ‘logical’ coordinates where they form a cube. Then interpolation weights are trivial to compute.

First, number the eight vertices of the hexahedron,

\[ x_1 = x(i, j, k) \]

[4]https://bitbucket.org/volshevsky/slurm: The code will be made completely open, but at the moment send us an E-mail to get access to the repository.
and define the following variables

\[ f_0 = 0.125 \times (x_8 + x_7 + x_6 + x_5 + x_4 + x_3 + x_2 + x_1) - x_0 \]
\[ f_1 = 0.125 \times (x_8 - x_7 + x_6 - x_5 + x_4 - x_3 + x_2 - x_1) \]
\[ f_2 = 0.125 \times (x_8 + x_7 - x_6 - x_5 + x_4 + x_3 - x_2 + x_1) \]
\[ f_3 = 0.125 \times (x_8 - x_7 - x_6 + x_5 + x_4 - x_3 + x_2 + x_1) \]
\[ f_4 = 0.125 \times (x_8 - x_7 - x_6 + x_5 - x_4 + x_3 - x_2 - x_1) \]
\[ f_5 = 0.125 \times (x_8 + x_7 - x_6 - x_5 - x_4 - x_3 + x_2 + x_1) \]
\[ f_6 = 0.125 \times (x_8 - x_7 - x_6 + x_5 - x_4 + x_3 - x_2 + x_1) \]
\[ f_7 = 0.125 \times (x_8 - x_7 - x_6 - x_5 - x_4 + x_3 + x_2 - x_1) \]

Solve the linear system for three unknowns \( a, b, c \) using Newton's method

\[ f = f_0 + f_1a + f_2b + f_3c + f_4ab + f_5c + f_6b + f_7abc \]
\[ g = g_0 + g_1a + g_2b + g_3c + g_4ab + g_5c + g_6b + g_7abc \]
\[ h = h_0 + h_1a + h_2b + h_3c + h_4ab + h_5c + h_6b + h_7abc \]

where \( g_0 \ldots g_7 \) and \( h_0 \ldots h_7 \) are defined analogous to \( f_0 \ldots f_7 \) in two other coordinates, \( y \) and \( z \). We use initial guess \( a = b = c = 0 \), hence \( df = -f, dg = -g, dh = -h \) and

\[ -f_0 = f_1 \times da + f_2 \times db + f_3 \times dc \]
\[ -g_0 = g_1 \times da + g_2 \times db + g_3 \times dc \]
\[ -h_0 = h_1 \times da + h_2 \times db + h_3 \times dc \]

When the coefficients are zero, it means the hexahedron in physical space is degenerate. Using Kramer’s method this linear system is solved for \( da, db, dc \).

---

5In the original document, the rightmost term (the coordinate of the point to interpolate on) in the expression for \( f_0 \) was just “x”. Here, we denote it as “x0” for clarity.
6In the original document, \( c \) is sometimes confused with \( g \).
After the new values of a, b, c have been computed, they are used to compute the new coefficients f0...f7, g0...g7, and h0...h7, and proceed with the next iteration of Newton's solver. Iterations continue until a da, db, and dc are small enough, or until the absolute value of any of a, b, or c exceeds the pre-defined threshold $\gtrsim 1$. In the latter case, the method fails, and a simple non-weighted average is used to obtain the interpolated value.

Finally, the interpolation weights for eight vertices are given by

\[
\begin{align*}
  w_1 &= 0.125 \times (1-c) \times (1-b) \times (1-a) \\
  w_2 &= 0.125 \times (1-c) \times (1-b) \times (1+a) \\
  w_3 &= 0.125 \times (1-c) \times (1+b) \times (1-a) \\
  w_4 &= 0.125 \times (1-c) \times (1+b) \times (1+a) \\
  w_5 &= 0.125 \times (1+c) \times (1-b) \times (1-a) \\
  w_6 &= 0.125 \times (1+c) \times (1-b) \times (1+a) \\
  w_7 &= 0.125 \times (1+c) \times (1+b) \times (1-a) \\
  w_8 &= 0.125 \times (1+c) \times (1+b) \times (1+a)
\end{align*}
\]

i.e., a, b, c are the coordinates of the interpolation point in the ‘logical’ space where eight vertices form a cube.

It is unnecessary to provide the full proof of the method here, however the following helps to understand the basic idea. If a, b, c are the unknown ‘logical’ coordinates, the interpolated value of any quantity $v$ given on the eight vertices of the hexahedron is given by

\[
v = \left( (1-c) \times (1-b) \times (1-a) \times v_1 + (1-c) \times (1-b) \times (1+a) \times v_2 + (1-c) \times (1+b) \times (1-a) \times v_3 + (1-c) \times (1+b) \times (1+a) \times v_4 + (1+c) \times (1-b) \times (1-a) \times v_5 + (1+c) \times (1-b) \times (1+a) \times v_6 + (1+c) \times (1+b) \times (1-a) \times v_7 + (1+c) \times (1+b) \times (1+a) \times v_8 \right) / 8
\]

Define

\[
\begin{align*}
  xa &= \left( (1-a) \times x_1 + (1+a) \times x_2 \right) / 2 \\
  xb &= \left( (1-a) \times x_3 + (1+a) \times x_4 \right) / 2 \\
  xc &= \left( (1-a) \times x_5 + (1+a) \times x_6 \right) / 2 \\
  xd &= \left( (1-a) \times x_7 + (1+a) \times x_8 \right) / 2 \\
  xe &= \left( (1-b) \times xa + (1+b) \times xb \right) / 2 \\
  xf &= \left( (1-b) \times xc + (1+b) \times xd \right) / 2
\end{align*}
\]

and obtain the following identity

\[
x = \left( (1-c) \times xe + (1+c) \times xf \right) / 2
\]
Here, make the following substitutions

\[
x = \frac{[(1-c) * ((1-b) * xa + (1+b) * xb) +
(1+c) * ((1-b) * xc + (1+b) * xd)]}{4}
\]

\[
x = \frac{[(1-c) * ((1-b) * ((1-a) * x1 + (1+a) * x2) +
(1+b) * ((1-a) * x3 + (1+a) * x4)) +
(1+c) * ((1-b) * ((1-a) * x5 + (1+a) * x6) +
(1+b) * ((1-a) * x7 + (1+a) * x8))]}{8}
\]

\[
x = \frac{[(1-c)(1-b)(1-a) * x1 + (1-c)(1-b)(1+a) * x2 +
(1-c)(1+b)(1-a) * x3 + (1-c)(1+b)(1+a) * x4 +
(1+c)(1-b)(1-a) * x5 + (1+c)(1-b)(1+a) * x6 +
(1+c)(1+b)(1-a) * x7 + (1+c)(1+b)(1+a) * x8]}{8}
\]

After regrouping, we obtain a nonlinear equation for the three unknowns, \(a\), \(b\), and \(c\),

\[
x = \left[ (x8 + x7 + x6 + x5 + x4 + x3 + x2 + x1) + // f0
a*(x8 - x7 + x6 - x5 + x4 - x3 - x2 - x1) + // f1
b*(x8 + x7 - x6 - x5 + x4 + x3 - x2 - x1) + // f2
g*(x8 + x7 + x6 + x5 - x4 + x3 - x2 - x1) + // f3
a*b*(x8 - x7 - x6 + x5 + x4 - x3 - x2 + x1) + // f4
a*c*(x8 - x7 + x6 - x5 - x4 + x3 - x2 + x1) + // f5
b*c*(x8 + x7 - x6 - x5 - x4 - x3 + x2 + x1) + // f6
a*b*c*(x8 - x7 - x6 - x5 + x4 + x3 + x2 - x1)] + // f7
\]

where the coefficients represent the above defined \(f0\)…\(f7\).

References

Bacchini, F., Olshevsky, V., Poedts, S., Lapenta, G., 2017. A new particle-in-cell method for modeling magnetized fluids. Computer Physics Communications 210, 79–91.

Beck, A., Frederiksen, J. T., Dérouillat, J., Sep. 2016. Load management strategy for Particle-In-Cell simulations in high energy particle acceleration. Nuclear Instruments and Methods in Physics Research A 829, 418–421.
Brackbill, J. U., Sep. 1991. FLIP MHD - A particle-in-cell method for magnetohydrodynamics. Journal of Computational Physics 96, 163–192.

Brackbill, J. U., Mar. 2005. Particle methods. International Journal for Numerical Methods in Fluids 47, 693–705.

Brackbill, J. U., Ruppel, H. M., Aug. 1986. FLIP - A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. Journal of Computational Physics 65, 314–343.

Brio, M., Wu, C. C., Apr. 1988. An upwind differencing scheme for the equations of ideal magnetohydrodynamics. Journal of Computational Physics 75, 400–422.

Caramana, E. J., Shashkov, M. J., Whalen, P. P., Jul. 1998. Formulations of Artificial Viscosity for Multi-dimensional Shock Wave Computations. Journal of Computational Physics 144, 70–97.

Chandrasekhar, S., 1961. Hydrodynamic and hydromagnetic stability.

Fatemizadeh, F., Moormann, C., 2015. Investigation of the slope stability problem using the material point method. IOP Conference Series: Earth and Environmental Science 26 (1), 012019. URL http://stacks.iop.org/1755-1315/26/i=1/a=012019

Harlow, F., 1964. The particle-in-cell computing method for fluid dynamics. Methods Comput. Phys. 3, 319–343.

Innocenti, M. E., Beck, A., Ponweiser, T., Markidis, S., Lapenta, G., 2015. Introduction of temporal sub-stepping in the Multi-Level Multi-Domain semi-implicit Particle-In-Cell code Parsek2D-MLMD. Computer Physics Communications 189, 47–59.

Karimabadi, H., Driscoll, J., Omelchenko, Y. A., Omidi, N., May 2005. A new asynchronous methodology for modeling of physical systems: breaking the curse of courant condition. Journal of Computational Physics 205, 755–775.

Kundu, P., Cohen, I., Dowling, D., 2012. Fluid Mechanics. Academic Press. URL https://books.google.be/books?id=iUo_4tsHQYUC
Kuropatenko, V. F., 1966. Difference methods for hydrodynamics equations. In: Difference methods for solutions of problems of mathematical physics. Part 1, Moscow: Nauka, 1966. Vol. 74. pp. 107–137.

Langdon, A. B., Birdsall, C. K., Aug. 1970. Theory of Plasma Simulation Using Finite-Size Particles. Physics of Fluids 13, 2115–2122.

Lapenta, G., Feb. 2012. Particle simulations of space weather. Journal of Computational Physics 231, 795–821.

Liska, R., Wendroff, B., Mar. 2003. Comparison of several difference schemes on 1d and 2d test problems for the euler equations. SIAM J. Sci. Comput. 25 (3), 995–1017. URL http://dx.doi.org/10.1137/S1064827502402120

Stasyszyn, F. A., Dolag, K., Beck, A. M., Jan. 2013. A divergence-cleaning scheme for cosmological SPMHD simulations. Monthly Notices of the Royal Astronomical Society 428, 13–27.

Stasyszyn, F. A., Elstner, D., Feb. 2015. A vector potential implementation for smoothed particle magnetohydrodynamics. Journal of Computational Physics 282, 148–156.

Stomakhin, A., Schroeder, C., Chai, L., Teran, J., Selle, A., Jul. 2013. A material point method for snow simulation. ACM Trans. Graph. 32 (4), 102:1–102:10. URL http://doi.acm.org/10.1145/2461912.2461948

Stone, J. M., Gardiner, T. A., Teuben, P., Hawley, J. F., Simon, J. B., Sep. 2008. Athena: A New Code for Astrophysical MHD. The Astrophysical Journal Supplement 178, 137–177.

Sulsky, D., Schreyer, H., Peterson, K., Kwok, R., Coon, M., Feb. 2007. Using the material-point method to model sea ice dynamics. Journal of Geophysical Research (Oceans) 112, C02S90.

Sulsky, D., Zhou, S.-J., Schreyer, H. L., May 1995. Application of a particle-in-cell method to solid mechanics. Computer Physics Communications 87, 236–252.
Tricco, T. S., Price, D. J., Aug. 2012. Constrained hyperbolic divergence cleaning for smoothed particle magnetohydrodynamics. Journal of Computational Physics 231, 7214–7236.

Tricco, T. S., Price, D. J., Bate, M. R., Oct. 2016. Constrained hyperbolic divergence cleaning in smoothed particle magnetohydrodynamics with variable cleaning speeds. Journal of Computational Physics 322, 326–344.

Woodward, P., Colella, P., Apr. 1984. The numerical simulation of two-dimensional fluid flow with strong shocks. Journal of Computational Physics 54, 115–173.

York, A. R., Sulsky, D., Schreyer, H. L., 2000. Fluid-membrane interaction based on the material point method. International Journal for Numerical Methods in Engineering 48 (6), 901–924. URL http://dx.doi.org/10.1002/(SICI)1097-0207(20000630)48:6<901::AID-NME910>3.0.CO;2-T