Development of an Unstructured Mesh Gyrokinetic Particle-in-cell Code for Exascale Fusion Plasma Simulations on GPUs

Chonglin Zhang[a,*], Gerrett Diamond[a], Cameron W. Smith[a], Mark S. Shephard[a]

[a]*Scientific Computation Research Center, Rensselaer Polytechnic Institute*
110 8th St, Troy, NY 12180

Abstract

This paper presents XGCm, a new unstructured mesh gyrokinetic Particle-in-Cell (PIC) code for modeling fusion plasma. The physical models and aspects of the numerical methods employed are the same as those used in the X-point gyrokinetic code, XGC. The core difference is that XGCm builds on an unstructured mesh-centric infrastructure that supports a distributed mesh, making it scalable in both the number of mesh elements and number of particles. A second advantage of an unstructured mesh infrastructure is its performance is not degraded when generally graded or anisotropic meshes are used. The switch from a particle-centric data infrastructure to a distributed mesh-centric infrastructure required the introduction of new methods to execute core PIC operations and substantial modifications to a number of key algorithms from those used in XGC. First, we present the methods and algorithms used in the development of XGCm, which performs all key computing steps on the GPU accelerators. GPU accelerators are providing the main computational power of current generation U.S. Department of Energy supercomputers. Secondly, we perform code validation and test using the circular geometry cyclone base case (case 5 of Burckel et al., Journal of Physics: Conference Series 260, 2010, 012006) and realistic DIII-D geometry case, respectively. The turbulence growth rate shows excellent agreement with existing XGC result in the first case, while ion temperature gradient turbulence growth is further demonstrated in the second case. Finally, we present weak scaling results, using up to full system (27,648 GPUs) of the Oak Ridge National Laboratory's Summit supercomputer.

Keywords: particle method, particle-in-cell method, unstructured mesh, fusion plasma physics, GPU

[*]Corresponding author

Email addresses: zhangc20@rpi.edu (Chonglin Zhang), diamog320@gmail.com (Gerrett Diamond), smithc11@rpi.edu (Cameron W. Smith), shephard@rpi.edu (Mark S. Shephard)

1. Introduction

To meet the aggressive plan to bring a prototype fusion power plant online by 2040 [1], fusion plasma modeling codes that provide the needed levels of simulation fidelity, while fully accounting for the geometric complexity of the reactor system, must be applied as part of the design and system evaluation processes. A number of codes with the potential to provide the required levels of physical modeling fidelity are based on the particle-in-cell (PIC) method. Ongoing advances in the development of exascale computing systems, modeling technologies, and numerical methods, hold the promise of supporting the PIC simulation workflows needed to carry out the desired high fidelity simulations.

Computational methods that execute on uniform grids are carried out most efficiently on GPU accelerated exascale computing systems. However uniform grid methods are not well suited to address many of the reactor design and operation questions in which the grid must directly account for complex geometric components (eg., divertor cassette assemblies, probe ports, etc.) and/or localized behaviors (e.g., pellet injection and ablation). On the other hand, unstructured mesh generators have evolved to the point that strongly graded, anisotropic meshes can be easily generated that can take full account of the geometric details and/or localized behaviors of interest. An increasing number of PIC codes for the modeling of plasma physics in fusion reactors, [2, 3, 4, 5, 6] are currently using or are planning to use unstructured mesh methods. To support the development of unstructured mesh PIC simulation codes that scale and are performant on GPU based exascale computers, a distributed mesh unstructured mesh for PIC infrastructure, PUMIPic [7], that is scalable both with respect to the numbers of particles and mesh elements, has been developed.

In this paper, we report the development of XGCm, a new distributed unstructured mesh gyrokinetic PIC code, short for x-point included gyrokinetic code, mesh-based. The code runs fully on current GPU hardware, and is aimed at exascale fusion plasma simulations. It adopts the physical algorithms from the well-established XGC code [2, 8]. The corresponding numerical algorithms within XGCm are presented, which are suitable for distributed unstructured mesh and achieve performance and scalability on GPU devices. XGCm heavily depends on the open-source Omega [9, 10] and PUMIPic [7] libraries for mesh and particle management, respectively. Both of these libraries use the Kokkos [11] programming model to achieve code portability.

Section 2 overviews the major operations of the gyrokinetic PIC method used in both XGC and XGCm. These include four major components: the gyroaveraging charge scatter process depositing particle charge onto the mesh vertices; the solution of the gyrokinetic Poisson equation on the mesh and associated mesh field operations; field gather operation to obtain electric and magnetic field information at particle location from the background mesh field; the ion and electron particle push operations. Section 3 overviews the PUMIPic unstructured mesh PIC infrastructure that XGCm builds upon. The four components of the gyrokinetic PIC method operate on different particle and mesh interaction levels, including particle-to-mesh operation, mesh field operation, mesh-to-particle operation, and particle operation, respectively. Building on Omega and PUMIPic, Section 4 discusses the algorithms used in XGCm to handle different operations in the gyrokinetic PIC method. Section 5 first presents a code validation study using the circular geometry cyclone base case [12, 13], where ion temperature gradient (ITG) turbulence growth is compared with the result of XGC. It then tests XGCm using ITG turbulence simulation with realistic DIII-D geometry. Section 6 provides weak scaling and computational performance results of XGCm. Section 7 summarizes the current work and provides directions for future work.

2. Gyrokinetic Particle-in-cell Method

The PIC method is usually used to study the micro turbulence, turbulence transport, and plasma instabilities in the tokamak devices, within the context of magnetic confinement fusion plasma modeling. Due to the strong magnetic field in the tokamak device, it is sufficient to track the guiding center motion of charged particle, while modeling the particle's gyration motion perpendicular to the magnetic field. This is called the gyrokinetic PIC method [14, 15, 16, 17, 18, 19, 20, 21]. In this method, the motion of charged particles are treated as a combination of particle guiding center motion and gyration motion surrounding the guiding center and perpendicular to the magnetic field. This allows the use of larger simulation time steps, significantly reducing the simulation cost.

A further simplification in the gyrokinetic PIC method is the delta-f treatment of the particle velocity distribution function (VDF) $f(t, \mathbf{x}, \mathbf{v})$ [15, 22, 19]. This takes advantage of the fact that the particle VDF is close to equilibrium Maxwellian-Boltzmann distribution in the core region of the tokamak device. It is thus sufficient to only simulate the perturbative part of the particle VDF, hence the name delta-f method. The delta-f gyrokinetic PIC method maintains a low statistical error while using relatively small numbers of simulation particles, further reducing the overall simulation time cost.

2.1. Delta-f treatment of particle VDF

The particle VDF, $f(t, \mathbf{x}, \mathbf{v})$, describes particles' collective behavior in the six-dimensional phase space $(\mathbf{x}, \mathbf{v})$, where $\mathbf{x}$ and $\mathbf{v}$ are the spatial and velocity vectors. It is the probability density of finding a particle with specific velocity $\mathbf{v}$ at specific location $\mathbf{x}$. It is governed by the

Boltzmann equation [23]. Under the gyrokinetic assumption, the particle property is described by the five-dimensional variable, $(\mathbf{X}, \mu, v_{\|})$, where $\mathbf{X}$ is particle guiding center position, μ is the magnetic moment, and $v_{\|}$ is the component of particle velocity parallel to the magnetic field vector $\mathbf{B}$. Magnetic moment is $\mu = \frac{mv_{\perp}^2}{2B}$, where m is the particle mass, $v_{\perp}$ is the component of particle velocity perpendicular to the magnetic field $\mathbf{B}$, and B is the magnitude of $\mathbf{B}$.

Under the gyrokinetic assumption, we solve the five-dimensional gyrokinetic Boltzmann equation [24, 25, 26]:

$$\frac{df}{dt} = S(f), \tag{1}$$

where $S(f)$ describe the change in particle VDF, $f(\mathbf{X}, \mu, v_{\|})$, due to particle collisions and external source or sink [23, 26]. For collisionless plasma in the core region of the tokamak device, we can ignore this and assume:

$$S(f) = 0. \tag{2}$$

In the delta-f gyrokinetic PIC method, the particle VDF, $f = f(\mathbf{X}, \mu, v_{\|})$, can be described using two separate contributions:

$$f = f_0 + \delta f. \tag{3}$$

where f_0 is the equilibrium Maxwellian-Boltzmann distribution function[19], while δf is the perturbative contribution to f. In the gyrokinetic PIC method, δ_f is described by the simulation particles (also called marker particles) and is denoted as f_p, to represent that it is from the contribution of simulation particles. Each simulation particle has its own properties, including spatial position $\mathbf{X}$, magnetic moment μ, and parallel velocity $v_{\|}$. By sampling all particles' information at a specific location $\mathbf{X}$, the particle VDF f_p is known.

2.2. Particle guiding center equations of motion

In the gyrokinetic PIC method, the property of a simulation particle is described by $(\mathbf{X}, \mu, v_{\|})$, its guiding center position, magnetic moment, and parallel velocity. The motion of each simulation particle is determined by the evolution of its guiding center position and parallel velocity. They are governed by the Lagrangian equations of motion [17, 26, 25],

$$\dot{\mathbf{X}} = \frac{d\mathbf{X}}{dt} = \frac{1}{D}\left[v_{\|}\mathbf{b} + \frac{v_{\|}^2 \nabla B \times \mathbf{b}}{B} + \frac{\mathbf{B} \times (\mu \nabla B - \mathbf{E})}{B^2}\right], \tag{4}$$

$$\dot{v}_{\|} = \frac{dv_{\|}}{dt} = -\frac{1}{D}(\mathbf{B} + v_{\|}\nabla B \times \mathbf{b}) \cdot (\mu \nabla B - \mathbf{E}), \tag{5}$$

where $\mathbf{B}$ is the equilibrium magnetic field, $B = |\mathbf{B}|$, $\mathbf{E}$ is the electric field vector, $\mathbf{b}$ is the unit vector along the magnetic field direction (along $\mathbf{B}$ direction), and,

$$D = 1 + \frac{v_{\|}}{B}\mathbf{b} \cdot (\nabla \times \mathbf{b}). \tag{6}$$

Equations 4 and 5 fully determine the particle trajectory and velocity. In the gyrokinetic PIC simulation, these first order ordinary differential equations are numerically discretized to determine the simulation particle properties at different time.

2.3. Gyrokinetic Poisson Equation

The gyrokinetic Poisson equation determines the electrostatic potential field given the charge field, which is generated by charged particles moving in the physical domain. In long wavelength limit, the electrostatic Gyrokinetic Poisson equation [14, 15, 27, 17, 18, 19, 20, 21, 28] is,

$$-\nabla_\perp \cdot \frac{n_0 m}{qB^2}\nabla_\perp \phi = \bar{n}_i - n_e, \tag{7}$$

where ϕ is the electrostatic potential, n_0 is the background charge density, $\bar{n}_i$ is the ion guiding center charge density, n_e is the electron charge density, $\nabla_\perp$ is the gradient operator, ρ_i is the thermal ion gyroradius, and q is the particle charge. $\bar{n}_i(\mathbf{x})$ at any position $\mathbf{x}$ is obtained from the gyroaverage charge scatter process and is defined as [17, 28, 29],

$$\bar{n}_i = \frac{1}{2\pi}\int f_i(\mathbf{X}, \mu, v_\parallel)\delta(\mathbf{X} + \vec{\rho}_i - \mathbf{x})d\mathbf{X}d\mu dv_\parallel d\alpha, \tag{8}$$

where $\mathbf{X}$ is the ion guiding center position vector, $\vec{\rho}_i$ is the ion gyroradius vector pointing from $\mathbf{X}$ to $\mathbf{x}$, α is the gyrophase angle of $\vec{\rho}_i$ relative to a reference vector, $\delta(\mathbf{X} + \vec{\rho}_i - \mathbf{x})$ is the Dirac delta function. The numerical calculation of the ion guiding center charge density $\bar{n}_i$ in the gyrokinetic PIC method is usually done using multi-point average. This process is shown in Figure 1, where an eight-point average is used. In XGCm, 32-point average is usually used for increased numerical accuracy.

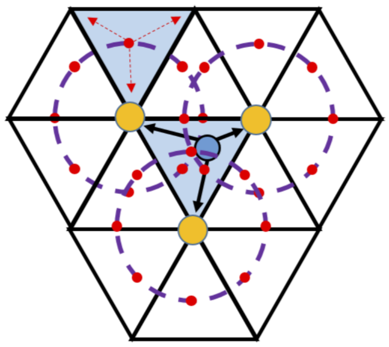


Figure 1: Schematics of the multi-point average process to perform the gyroaverage charge scatter.

Following Equation 3, the delta-f treatment of VDF, the electrostatic potential Φ, ion guiding center charge density $\bar{n}_i$, and electron charge density n_e can be similarly expressed as two separate components: the equilibrium component and the perturbative (turbulent) component. For instance, the electrostatic potential ϕ can be written as,

$$\phi = \phi_0 + \delta\phi, \tag{9}$$

where ϕ_0 is the equilibrium component, while $\delta\phi$ is the turbulent component. Equation 7 can then be solved by separating it into two components: the axisymmetric or equilibrium component with toroidal mode number $n = 0$; and the non-axisymmetric or turbulent component with toroidal mode number $n \neq 0$ [17, 18, 30].

2.4. Major steps of the PIC method

The four major steps of the PIC method are:

- **Charge Scatter:** The "charge" information associated with the particles is related to the domain definition such that the forcing function driving the field evolution, and, potentially, the domain properties can be updated. In the gyrokinetic PIC method, the charge of the simulation particle is projected onto the background mesh field [29].

- **Field Solve:** The domain physics equation (the gyrokinetic Poisson equation in section 2.3) governing the electrostatic potential field is updated based on the updated forcing function, the domain properties are updated when changed, and the associated discrete system is constructed and solved.

- **Field Gather:** The values of the fields that drive the particles are associated with each particle through an appropriate interpolation procedure.

- **Particle Push:** The particles are moved to a new location as a function of the fields and time step.

The manner in which the domain fields are defined over the spatial domain of a PIC calculation has a substantial influence on the design of the code data structures and domain related operations. In the case of the XGC gyrokinetic code [28, 2, 8] and XGCm code, the spatial discretization consists of a poloidal plane mesh that is repeated on a number of poloidal planes in the toroidal direction, which is typically 32 to 128. Within the poloidal plane, the ion and electron charge density, electrostatic potential, and electromagnetic potential fields are defined as C^0 continuous polynomials (currently defined as linear polynomials) over triangular finite elements. For a location within two adjacent poloidal planes, a linear interpolation over a nearly field-following coordinate is used.

To effectively model the fact that the particles move much more quickly in the toroidal direction than in the poloidal plane, the unstructured poloidal plane mesh is defined to be quasi field following [31] where the primary mesh vertices are placed on magnetic flux curves with a spacing that follows the motion of a particle on the flux curve as it moves between poloidal planes. The mesh in the core region uses a one element deep construction of elements between flux curves. Outside the core region around the X-Point, in open flux curves, and between flux curves and the wall, a set of ah-hoc meshing operators and general unstructured mesh methods are applied, which introduce additional mesh vertices between flux curves, and between flux curves and the wall, to produce meshes with the desired element shape quality. An example unstructured mesh on one poloidal is shown in Figure 2.

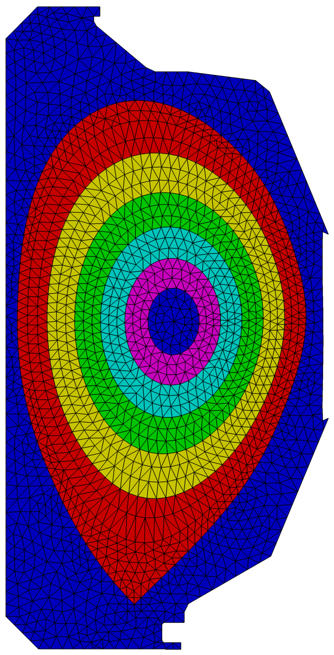


Figure 2: An example of a coarse mesh generated using the XGC meshing procedure. Meshes for physics studies have at least one to two orders of magnitude more elements.

3. Parallel Unstructured Mesh Infrastructure for PIC Calculations

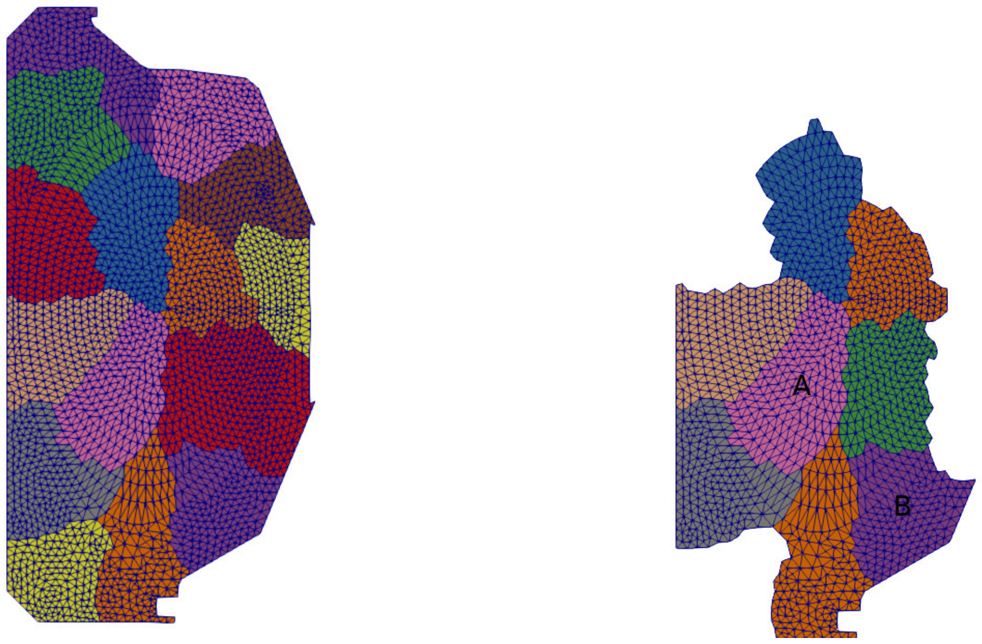
The typical approach to the development of an unstructured mesh PIC code is for the particle data structure to be the core data structure, where the data structure of the particles contain a pointer to the mesh element in which the particles are contained. The mesh is stored in an independent data structure that is typically copied into the memory of each process. Although the mesh data is substantially smaller than the particle data, maintaining a copy of the mesh in each memory space does limit the scalability with respect to mesh size. PIC codes using particle-centric data structures also include a spatial-based structure, either a uniform grid or a tree-based structure, to support the process of determining which unstructured mesh element a particle is in after a push operation.

The unstructured mesh infrastructure used in XGCm, PUMIPic [7], takes an alternative approach in which the core data structure is a distributed mesh, where the particles are tied to the mesh elements in which they currently reside. This approach provides an effective opportunity to distribute the mesh over the nodes of the compute system, thus supporting scalability with respect to the mesh, while maintaining ready access to the mesh fields of mesh elements where the particles will move to during the next push step. The mesh data structure used in PUMIPic, Omega [10, 9] stores a complete mesh topology and has been designed to support the effective execution of unstructured mesh operations on distributed meshes on massively parallel computers employing GPU accelerators. The mesh-based functions supported range from adjacency-based point location operations using the mesh adjacency and geometry information, the integration of physical field parameters over the mesh, and the implementation of effective linear system matrix assembly and solution operations.

To store particles based on mesh elements, an additional structure is maintained that groups particles in memory based on the mesh element they are in. PUMIPic provides Sell-C-σ [32, 33] and Cabana AoSoA particle data structure [34] that supports the irregular memory storage of the mesh element to particles relationship. Sell-C-σ is derived from the work of Kreutzer et al. [32]. The detailed implementations within PUMIPic are further discussed in the work of Diamond et al. [7], to support key particle operations such as particle push, particle migration, and the association of particle with the mesh element. An AoSoA, array-of-structures-of-arrays, is the principle particle data structure used in Cabana to provide performance portoability across hardware [34]. Cabana AoSoA is extended in PUMIPic to support grouping of structure-of-arrays, which contains particle data and is associated with mesh element. The advantage of Sell-C-σ or Cabana AoSoA particle data structure may depend on the details of the particle operations being performed, and could be case-specific. As such, this needs to be discussed separately.

To maintain the particle structure as particles move through the domain, three operations are required. After every particle push, we need to determine whether each particle has moved to a new mesh element and if so which element it has moved to. To achieve this, an adjacency-based search is executed on each particle using ray tracing and barycentric coordinates methods. After the new mesh element of each particle is determined, the particle structure must be updated to reflect these changes. Since the unstructured mesh is distributed, we need to check that if each particle needs to be migrated to a new process based on its new mesh element. Once all the particles are migrated to the correct processes, the particle structure is rebuilt in order to add or remove particles, and reorder particles based on their new mesh elements and processes.

A key component of PUMIPic is the manner in which the mesh is distributed, as a set of so-called PICparts, to the MPI processes employed on today's parallel computers. A PICpart is an overlapping domain decomposition with halo cells that support local computations on the particles in a manner that reduces interprocess communications. As discussed in the work of Diamond et al. [7], the definition of the PICpart begins by partitioning the mesh into a non-overlapping set of parts. Each of those parts defines the core of a PICpart, or simply called PICpart core. The remainder of the PICpart, called PICpart buffer, is the set of other parts either bound the PICpart core or have elements within a given number of buffer elements of the PICpart core. In PUMIPic, there is sufficient buffer such that any particle that is in an element in the PICpart core at the start of a push operation, will end up in an element on that PICpart. Figure 3a shows a 2D tokamak cross section with a very coarse mesh partitioned into 15 non-overlapping mesh parts. Figure 3b shows the PICpart defined for mesh part labeled A. Mesh part labeled A is the PICpart core, all other mesh parts including mesh part labeled B are the PICpart buffer.



(a) (b)

Figure 3: (a) Two-dimensional unstructured mesh partitioned using multi-level graph partitioner. (b) PICpart generated for mesh part A.

Since the particle-based gather and scatter operations in a PIC calculation will require communications in each time step, PUMIPic deems it is satisfactory to move particles between PICparts during these steps. With the PICpart defined in PUMIPic, all required information is local to the

PICpart and no communications are required during the particle push step. Care is required in the definition of the PICpart buffer, to ensure it does not produce large increases in memory usage or effort required to maintain the mesh distribution information. The size of the PICpart buffer is determined by the fact that, particles contained in the mesh elements of a PICpart core do not move outside of the PICpart buffer in a single push operation. After a particle push operation, particles can enter the buffer and be close enough to the boundary of the PICpart that they may move off the PICpart in the next particle push. As such, it becomes necessary to do the communication to move the particle onto a PICpart for which that element is sufficiently far from that PICpart's boundary. The communications required to move those particles can be coordinated and carried out in the charge deposition step which always requires communications.

With the current PICpart definition, a substantial percentage of the particles that move to elements in the PICpart buffer are far enough from the PICpart boundary. In general, they would not exit the current PICpart on the next push. In these cases particles are only migrated for the purpose of improving load balance for the next push operation. As particles move through the domain during the simulation, the regular application of dynamic load balancing [7], another core operation, is needed.

4. Particle-in-cell Methods and Numerical Algorithms Suitable for GPU

Section 2 overviewed the delta-f gyrokinetic PIC method and the major operations in both XGC and XGCm. These include four major components:

1. the charge scatter process depositing particle charge onto vertices of the background mesh;
2. solving the gyrokinetic Poisson equation and calculation of the electric field, both of which is performed on the mesh;
3. field gather operation to obtain the electric field information at particle location from the background mesh field;
4. and the ion and electron particle push operations.

Taking a mesh-centric approach, we categorize the four major components of the delta-f gyrokinetic PIC method according to the level of interactions between the mesh and the particle. This categorization allows us to better analyze each component and improve their computational performance in future work. Corresponding to the four major components of the gyrokinetic PIC method, there are four types of particle and mesh interactions, including:

- particle-to-mesh operation, corresponding to charge scatter;

- mesh field operation, corresponding to solving the Poisson equation and calculation of the electric field;
- mesh-to-particle operation, corresponding to electric field gather;
- particle operation, corresponding to ion and electron particle push.

In general, each type of interaction corresponds to either an Eulerian or a Lagrangian representation, or an interaction between the Eulerian and Lagrangian representation. Building on Omega [9, 10], PUMIPic [7], and PETSc [35, 36, 37] libraries, this section discusses the numerical algorithms used in XGCm to handle each of these interactions. In XGCm, all four components of the gyrokinetic PIC method are executed on the GPU.

4.1. Flux surface aligned and field-following mesh

In XGCm, we use a cylindrical coordinate system $(\hat{r}, \hat{\varphi}, \hat{z})$ as shown in Figure 4 to describe the tokamak geometry. O is the origin of the coordinate system, while $\hat{r}$, $\hat{\varphi}$, and $\hat{z}$ are the three basis vectors. The coordinate of any point P is (r, φ, z), where r represents the distance from point P to the center $\hat{z}$ axis, while z is the distance from point P to the origin O along the $\hat{z}$ axis. Unit vectors $\hat{r}$, $\hat{z}$ correspond to the (r, z) coordinates and form a plane called the poloidal plane. Unit vector $\hat{\varphi}$ corresponds to the φ coordinate or the toroidal direction. φ represents the rotation angle of point P relative to a reference poloidal plane with respect to the $\hat{z}$ rotation axis. A poloidal plane corresponds to points with constant φ angle.

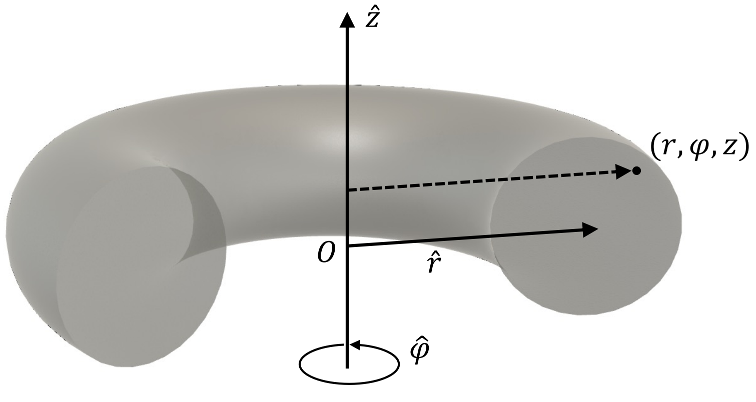


Figure 4: Cylindrical coordinate system used in XGCm.

Under this coordinate system, the equilibrium magnetic field in the tokamak, $\mathbf{B}$, is represented as summation of the poloidal and toroidal components,

$$\mathbf{B} = \mathbf{B}_p + \mathbf{B}_\varphi, \quad (10)$$

where $\mathbf{B}$ is axisymmetric. $\mathbf{B}_p$ is the poloidal component and is represented as,

$$\mathbf{B}_p = \mathbf{B}_r + \mathbf{B}_z = B_r\hat{\mathbf{r}} + B_z\hat{\mathbf{z}} = \nabla\Psi \times \nabla\varphi, \tag{11}$$

where $\Psi = \Psi(r, z)$ is called the flux function and a function of (r, z). Specifically,

$$\mathbf{B}_r = -\frac{1}{r}\frac{\partial\Psi}{\partial z}\hat{\mathbf{r}}, \tag{12}$$

$$\mathbf{B}_z = -\frac{1}{r}\frac{\partial\Psi}{\partial r}\hat{\mathbf{z}}. \tag{13}$$

$\mathbf{B}_\varphi$ is the toroidal component and is a function of (r, z), $\mathbf{B}_\varphi = B_\varphi(r, z)\hat{\varphi}$. It can also be written as,

$$\mathbf{B}_\varphi = g(r, z)\nabla\varphi, \tag{14}$$

where $g(r, z)$ is defined as $g(r, z) \equiv rB_\varphi(r, z)$. Functions $\Psi(r, z)$, $g(r, z)$, and the equilibrium magnetic field $\mathbf{B}$ are known a prior for each simulation condition. They are represented using piecewise cubic spline interpolation in both XGC and XGCm. $\mathbf{B}$ can thus be determined at any point in the simulation domain.

In XGC and XGCm, the three dimensional simulation domain is discretized into a number of uniformly spaced two dimensional poloidal planes along the toroidal direction. The angle between two neighboring poloidal planes is $\Delta\varphi = 2\pi/N_\varphi$, where N_φ is the number of poloidal planes. Each poloidal plane is then discretized using unstructured triangular meshes as shown in Fig 2. A unique feature of XGC and XGCm mesh is that, the equilibrium magnetic field information is built into the mesh [17, 31], achieving high numerical accuracy.

In a Tokamak, points with constant value of $\Psi(r, z)$ form the magnetic flux surface. Its projection on each poloidal plane is the magnetic flux curve. Each poloidal plane is discretized using flux surface aligned unstructured triangular meshes in XGC and XGCm. In the closed flux curve region [38], mesh vertices are arranged such that they are on a flux curve. Flux surface aligned mesh allows us to resolve perturbations with low parallel wavenumber and high toroidal mode number, while using relatively low toroidal resolution N_φ [26].

The meshes used in XGC and XGCm also exhibit the field-following feature. We pick a mesh vertex, $P_i = (r_i, \varphi_j, z_i)$, in the closed flux surface region on a poloidal plane j with toroidal angle $\varphi_j = \Delta\varphi$, where poloidal plane is indexed as $j = 0, 1, 2, ..., (N_\varphi - 1)$. Staring from P_i and traveling along the magnetic field line $\mathbf{B}$ in the same direction as $\hat{\varphi}$), the magnetic field line crosses another point $P_k = (r_k, \varphi_{j+1}, z_k)$ in the neighboring poloidal plane $(j + 1)$ with toroidal angle $\varphi_{j+1} = (j + 1)\Delta\varphi$. P_k lies on the same flux surface as P_i, and in general is also a mesh vertex or very close to a mesh vertex in poloidal plane $(j + 1)$. Traveling in the opposite direction along the magnetic field line $\mathbf{B}$, we have another intersection point P_l in the neighboring poloidal plane $(j-1)$ with toroidal angle $\varphi_{j-1} = (j-1)\Delta\varphi$. In general, P_l also corresponds to a mesh vertex or is very close to a mesh vertex on the same flux surface. This is discussed further in the work of Zhang et al. [31].

Taking advantage of the field-following feature of the unstructured mesh used in XGC and XGCm, major operations in the gyrokinetic PIC method are performed along the magnetic field line, rather than along the toroidal direction. For example, the deposition of a particle's charge to two neighboring poloidal planes in the charge scatter operation, is carried out along the magnetic field line direction [39].

4.2. XGCm mesh partitions

In XGCm, we have three levels of mesh partitions: the first level partition is the *toroidal partition*, which divides the simulation domain into N_φ sub-domains along the toroidal direction, where each sub-domain is bounded by two neighboring poloidal planes; the second level partition is the *poloidal partition*, which divides the poloidal plane mesh into multiple radial mesh regions or PICparts according to the flux curves, each of which is generally bounded by two flux curves; the third level partition is the *group partition*, where each radial mesh region or PICpart can correspond to one or multiple processes.

In our mesh-centric approach, particles are associated with triangular mesh elements. The *toroidal partition* allows simulation particles to be grouped into sub-domains along the toroidal direction, resulting in better data locality. In addition, mesh fields are distributed to multiple processes along the toroidal direction, achieving better memory usage.

The *poloidal partition* is justified by the fact that the motion of the particles with respect to their projection onto the poloidal plane mesh is such that they tend to move slowly in the radial direction. This is used in the specification of radial mesh regions or PICparts to distribute the mesh across processes. Figure 5 shows a coarse mesh in which two PICparts are highlighted. In this example, the core of the PIC part is the mesh between two flux curves and the remainder of the PICpart is the mesh between three sets of flux curves on each side of the core part. Similarly, the *poloidal partition* allows for better memory usage and data locality, achieving better computing performance.

In the *group partition*, a group of processes is assigned to each PICpart, where particles associated with the same PICpart are distributed between multiple processes. This allows for a large number of simulation particles to be associated with each mesh element and each PICpart. A large number of simulation particles is usually needed to achieve low numerical simulation noise.

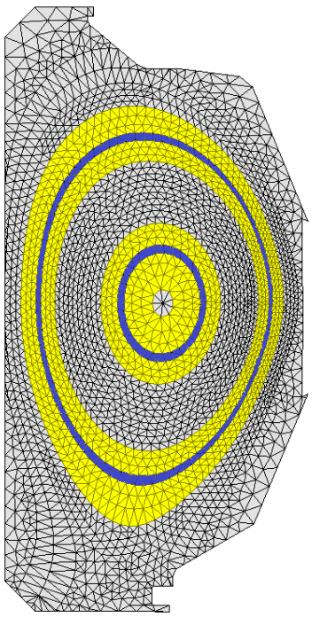


Figure 5: Example of PICpart mesh partition used in XGCm. Here two PICparts on a coarse mesh are shown, where a PICpart includes a core region with triangular elements between a pair of flux curves buffered by triangular elements between three pairs of flux curves on each side. Regions with blue color are the PICpart core and regions with yellow color are the PICpart buffer.

4.3. Particle charge scatter

4.3.1. Numerical algorithm

The charge of a simulation particle is scattered onto mesh vertices of two neighboring poloidal planes bounding that particle [17, 28]. This consists of three steps for ion species and two steps for electron species. Supposing we have a simulation particle P_i with coordinate (r_i, φ_i, z_i), it is bounded by two neighboring poloidal planes indexed as j and $(j+1)$ with toroidal angles φ_j and φ_{j+1}, $\varphi_j \leq \varphi_i < \varphi_{j+1}$. The poloidal plane in the middle of the two poloidal planes j and $(j+1)$ is indexed as $(j+\frac{1}{2})$, where its toroidal angle is $\varphi_{j+\frac{1}{2}} = (j+\frac{1}{2})\Delta\varphi$.

The first step of charge scatter is to perform the field-following projection. In this step, particle P_i is projected to a point P_i' in the poloidal plane $(j+\frac{1}{2})$ along the magnetic field line direction. The end result of this step is a charge field defined on each mesh vertex in the poloidal plane $(j+\frac{1}{2})$. Point P_i' lies in a triangular element indexed as t with three vertices ordered as $P_i^{t_1}$, $P_i^{t_2}$, and $P_i^{t_3}$. Charge of particle P_i is correspondingly deposited to the three mesh vertices $P_i^{t_1}$, $P_i^{t_2}$, and $P_i^{t_3}$ through linear interpolation of value at point P_i'. Linear weights are used in the charge deposition and depends on the barycentric coordinate of P_i' in triangular element t. Supposing the barycentric coordinate of P_i' is (η_1, η_2, η_3), the linear weights are η_1, η_2, and η_3.

Essentially, the field-following operation discussed above projects a point, $\mathbf{x}_1 = (r_1, \varphi_1, z_1)$, to a point, $\mathbf{x}_2 = (r_2, \varphi_2, z_2)$, on a poloidal plane with toroidal angle φ_2. Point $\mathbf{x}_1$ represents the simulation particle at position (r_1, φ_1, z_1). The operation projects point $\mathbf{x}_1$ to point $\mathbf{x}_2$ along the magnetic field line direction, and is denoted as $P_{\text{ff}}(r_1, z_1; \varphi_1 \mapsto \varphi_2)$,

$$\begin{aligned} P_{\text{ff}}(r_1, z_1; \varphi_1 \mapsto \varphi_2) &= \int_{(r_1,\varphi_1,z_1),\Psi(r,z)=\text{constant}}^{(r_2,\varphi_2,z_2)} \frac{\mathbf{B}_p(r,\varphi,z)}{B_\varphi} \hat{\varphi} \cdot d\overrightarrow{s} \\ &= \int_{\varphi_1,\Psi(r,z)=\text{constant}}^{\varphi_2} \frac{\mathbf{B}_p(r,\varphi,z)}{B_\varphi} r d\varphi \end{aligned}, \tag{15}$$

where $\mathbf{B}_p(r, z)$ is the poloidal component of the magnetic field vector at point (r, z), while $B_\varphi = B_\varphi(r, z) = |\mathbf{B}_{0,\varphi}(r, z)|$ is the magnitude of the toroidal component of the magnetic field vector at point (r, φ, z) and $\Psi(r, z)$ is the flux function. We have,

$$(r_2, z_2) = (r_1, z_1) + P_{\text{ff}}(r_1, z_1; \varphi_1 \mapsto \varphi_2). \tag{16}$$

The second step of charge scatter is to perform the gyroaverage operation for the ion species as outlined in Equation 8, and is ignored for the electron species. This step operates on mesh vertices in the poloidal plane $(j + \frac{1}{2})$. The end result is a charge field defined on each mesh vertex in the poloidal plane $(j + \frac{1}{2})$. The numerical implementation of gyroaverage in a gyrokinetic PIC code is usually done using multi-point average as shown in Figure 1. In general, an n_g-point average is used where n_g is the number of discrete gyro points representing the ion gyration orbit. In XGCm, usually $n_g = 32$ is used. Denoting the perpendicular velocity of an ion as $v_\perp$, its magnetic moment as μ, and the magnitude of magnetic vector at its guiding center position as $B = |\mathbf{B}|$, the ion gyroradius, ρ, is calculated as,

$$\rho = \frac{mv_\perp}{qB} = \sqrt{\frac{2m\mu}{q^2 B}}. \tag{17}$$

We assume there is an imaginary ion particle located on vertex $P_i^{t_k}$ of triangular element t, where k is indexed as $k = 1, 2, 3$. The imaginary ion particles has same properties as ion particle P_i except charge, which is linearly weighted from particle P_i to vertex $P_i^{t_k}$ in the first step. The gyroradius of the imaginary ion particle is $\rho_i^{t_k} = \sqrt{\frac{2m\mu_i}{q^2 B_i^{t_k}}}$, where μ_i is the magnetic moment of P_i and $B_i^{t_k}$ is the magnitude of the magnetic vector at vertex position $P_i^{t_k}$. The gyration orbit of the imaginary ion particle is a circle centered on vertex $P_i^{t_k}$ with radius $\rho_i^{t_k}$. The gyration orbit is represented by n_g uniformly spaced points, $P_i^{t_k,1}, P_i^{k_1,2}, ..., P_i^{t_k,n_g}$ on the circle. We consider an arbitrary point $P_i^{t_k,l}$ and assume that it is inside a triangular element t_k^l with three mesh vertices $t_k^{l,1}$, $t_k^{l,2}$, and $t_k^{l,3}$, where l is indexed as $l = 1, 2, 3, ..., n_g$. Charge of the imaginary ion particle located on vertex $P_i^{t_k}$ is linearly deposited to the three vertices $t_k^{l,1}$, $t_k^{l,2}$, and $t_k^{l,3}$. Similar to step one, the linear weights depend on the barycentric coordinate of $P_i^{t_k,l}$ in triangular element t_k^l. This process can be repeated for an imaginary ion particle located on any one of the three vertcies in triangular element t.

The ion gyroaverage process discussed above depends on the physical properties of each ion particle including its position, and the numerical calculation is unique to each ion particle. This poses challenges in the numerical implementation due to the complexity and time cost of the gyroaverage process. Specifically, we need to perform multiple search operations to determine the triangular elements in which each point $P_i^{t_k,l}$ on the gyration orbit lies. The number of search operations to be performed is $9n_g$ for each ion particle. To reduce the time cost associated with the gyroaverage process, we introduce a *gyroaverage mapping* P_{ga} and apply this mapping to each imaginary ion particle located on vertex $P_i^{t_k}$ directly. In essence, P_{ga} is a four dimensional mapping between mesh vertices in the poloidal plane mesh.

We briefly describe the calculation of the *gyroaverage mapping* P_{ga} here. For a mesh vertex p located on poloidal plane $j + \frac{1}{2}$, we create $(n_{gr} + 1)$ circles surrounding it, where n_{gr} is chosen as a constant in each simulation. All the circles are centered on vertex p with uniformly spaced radius $R_q = q\frac{R_0}{n_{gr}}$, where q is indexed as $q = 0, 1, 2, 3, ..., n_{gr}$. With the chosen radius, the first circle is a point. R_0 is chosen as a value larger than the maximum gyroradius in the simulation. n_g equally spaced points are placed on each circle q, and are denoted as $P_q^1, P_q^2, ..., P_q^{n_g}$. These points discretely represent an ion gyration orbit with radius R_q. With properly chosen n_{gr} and R_0, we can resolve all possible gyration orbits of ion particles in a simulation. We assume point P_q^s is inside a triangular element t_q^s, with three mesh vertices indexed as $t_q^{s,1}$, $t_q^{s,2}$, and $t_q^{s,3}$, where s is indexed as $s = 1, 2, 3, ..., n_g$. The barycentric coordinates of the three mesh vertices $t_q^{s,1}$, $t_q^{s,2}$, and $t_q^{s,3}$ are denoted as $\chi_q^{s,1}$, $\chi_q^{s,2}$, and $\chi_q^{s,3}$, respectively. Supposing the charge on vertex p is q_p, it is interpolated to a mesh vertex $t_q^{s,k}$ as $\frac{q_p\chi_q^{s,k}}{n_g n_{gr}}$, where $k = 1, 2, 3$. Since vertex $t_q^{s,k}$ is a mesh vertex located on the same poloidal plane $j + \frac{1}{2}$, we denote its index as p'. We thus obtain the (p, q, s, p') component of the *gyroaverage mapping* P_{ga},

$$P_{ga}(p, q, s, p') = \frac{q_p\chi_q^{s,k}}{n_g n_{gr}}. \tag{18}$$

The dimensions of the *gyroaverage mapping* P_{ga} is $(N_v, (n_{gr} + 1), n_g, N_v)$, where N_v is the number of mesh vertices in a poloidal plane mesh. P_{ga} is completely determined through geometric relationships between mesh vertices and triangular elements. As a result, it can be computed once at the beginning of the simulation, stored, and then used later as needed. Moreover, many components of $P_{ga}(p, q, s, p')$ will be 0. It can therefore be stored in a data structure similar to those used for a sparse matrix to reduce memory usage.

Using the *gyroaverage mapping*, P_{ga}, we can significantly simplify the ion gyroaverage operation, which is step two of the ion charge scatter process. For an imaginary ion particle located on a mesh vertex indexed as t_k with gyroradius ρ_{t_k}, we assume the following relationship holds:

$$R_{u-1} \leq \rho_{t_k} < R_u, \tag{19}$$

where integer index $u \geq 1$ and $u \leq n_{gr}$. The charge deposited from this imaginary ion particle with charge q^{t_k} to any vertex indexed as v is denoted as $q^{t_k,v}$. With linear interpolation, it is determined that,

$$q^{t_k,v} = \frac{q^{t_k}}{n_g} \sum_{s=1}^{n_g} [w_1 P_{ga}(t_k, u-1, s, v) + w_2 P_{ga}(t_k, u, s, v)]. \tag{20}$$

The linear weights are defined as,

$$w_2 = \frac{n_{gr}\rho_{t_k}}{R_0} - \lfloor \frac{n_{gr}\rho_{t_k}}{R_0} \rfloor, \tag{21}$$

and,

$$w_1 = 1 - w_2. \tag{22}$$

Here $\lfloor \cdot \rfloor$ is the floor function. With Equation 20, the gyroaverage operation now only involves multiplication and summation operations and does not require the expensive search operation. This process is thus greatly simplified numerically, and is also suitable for GPU calculations where the gyroaverage operation can be performed simultaneously for different ion particles.

The third step of ion charge scatter (or the second step of electron charge scatter) is the interpolation of charge at any mesh vertex p in poloidal plane $(j + \frac{1}{2})$ to mesh vertices in the two neighboring poloidal planes j and $(j + 1)$. The end result of this step is a charge field defined on each mesh vertex in each poloidal plane j. First, a field-following projection is performed from mesh vertex p to two points p_1 and p_2 in the neighboring poloidal planes j and $(j + 1)$ along the magnetic field line. The same field-following projection as discussed in the first step of charge scatter is used. Linear interpolation is used in XGCm, where the interpolation weight depends on the arc distance between vertex p and the projected points p_1 and p_2 along the magnetic field line. The charge is then interpolated from points p_1 and p_2 to mesh vertices of the triangular element in which each of the two points lies. Linear weights are used in the interpolation and are depending on the barycentric coordinate of p_1 and p_2 in the two triangular element. Finally, the contributions of all particles belonging to different toroidal sections are summed to the appropriate poloidal planes.

When discussing the charge scatter operation in the above three steps, we only consider the deposition of charge and ignored the discussion of charge density calculation for simplicity. The calculation of charge density enters the calculations at both step two and three for ion charge scatter or step two for electron charge scatter. Simply speaking, to calculate the charge density defined on each mesh vertex, the charge defined on the same mesh vertex is divided by the volume surrounding that vertex.

4.3.2. Implementation on the GPU

As seen from the discussions in Section 4.3.1, the charge scatter operation for each simulation particle can be done independently. This allows us to perform the charge scatter process on the GPU efficiently, where different hardware threads of the GPU can perform the calculations simultaneously for different mesh elements and particles in that mesh element. In XGCm, particles are stored in the PUMIPic particle structure. The particle structure contains the information of all particles including the mesh element each particle belongs to, and is stored on the GPU memory. The building block for performing charge scatter on the GPU is a Kokkos “parallel_for” loop operation, which is analogous to the “for” loop on the CPU. The “parallel_for” loop is performed on different hardware threads simultaneously for mesh elements and particles with different index, while the “for” loop is performed sequentially on the same CPU thread for each index in the loop.

Listing 1 provides a simplified code, describing how the charge scatter operation is performed on the GPU. “ps” stands for the PUMIPic particle structure object, while “parallel_for” is performed on the GPU and is the equivalent of “for” operation on the CPU. “ptcls” is the variable name of particle structure object, containing all particle information. “scatter_to_vertices” is the name of a user defined C++ language Lambda expression and corresponds to the first step of the charge scatter operation. With “parallel_for”, calculations can be performed simultaneously for different particles belongs to different triangular elements, where each GPU hardware thread operates on one particle at a time. In Listing 1, “e” is the triangular element index, “pid” is the particle index, “mask” is an integer denoting whether the current memory location of “ptcls” object has an actual particle. “x_c” object stores the (r, φ, z) coordinates of all simulation particles, while “x_p” store the coordinates of a single particle with index “pid”. For simplicity, the actual code segment corresponding to the first step of the charge scatter operation is not shown here. The code segment first performs the field-following projection of a particle with index “pid”, then performs the charge deposition where the charge is deposited to the three mesh vertices bounding the projection point.

```
auto x_c = ptcls->get<PTCL_COORDS>();
auto scatter_to_vertices = PS_LAMBDA(const int& e, const int& pid, const int
& mask) {
  if (mask) {
      // particle position
      x_p[0] = x_c(pid, 0);
      x_p[1] = x_c(pid, 1);
      x_p[2] = x_c(pid, 2);

      // perform charge scatter operation for this particle
      ......
      // code segment deposit charge from particle to mesh vertices
      ......
    }
  };
  ps::parallel_for(ptcls, scatter_to_vertices, "scatterToVertices");
```

Listing 1: Simplified code showing how the first step of the charge scatter operation is performed on the GPU.

4.4. Solving the gyrokinetic Poisson equation

Due to the two dimensional nature, the gyrokinetic Poisson equation can be solved on each poloidal plane separately. Currently, the gyrokinetic Poisson equation, Equation 7, can be solved either on the CPU or the GPU using the PETSc library [37, 35, 36, 40]. In solving this equation, the electrostatic potential is defined on each mesh vertex, and its value is assumed as 0 on the domain boundary.

4.4.1. Linear equations and mesh partition

Equation 7 is linearized and separated into two equations corresponding to two components: the axisymmetric or equilibrium component, and the non-axisymmetric or turbulent component.

The axisymmetric component equation being solved in XGCm is [17, 18, 30],

$$-\nabla_\perp \cdot \frac{n_0 m}{eB^2}\nabla_\perp \phi_0 + \frac{n_0}{T_{e,0}}\phi_0 = \overline{(\bar{n}_i - n_{e,NA})} + \frac{n_0}{T_{e,0}}\langle \phi_0 \rangle, \tag{23}$$

where ϕ_0 is the axisymmetric electrostatic potential, e is the elementary charge, n_0 and $T_{e,0}$ are the background number density and electron temperature, $\bar{n}_i$ is the ion guiding center charge density, $n_{e,NA}$ is the non-adiabatic electron charge density, $\langle \cdot \rangle$ is the flux-surface average operation, and $\overline{\cdots}$ is the toroidal average operation.

The non-axisymmetric component equation being solved in XGCm is,

$$-\nabla_\perp \cdot \frac{n_0 m}{eB^2}\nabla_\perp \delta\phi + \frac{n_0}{T_{e,0}}\delta\phi = \delta\bar{n}_i - \delta n_{e,NA}, \tag{24}$$

where $\delta\phi$ is the turbulent electrostatic potential.

Equations 23 and 24 are discretized with linear finite elements and are solved iteratively. Since the two equations have the same linear matrix, they are solved by supplying different right-hand-side vector in the PETSc iterative solver. On each poloidal plane, multiple MPI ranks can be used to solve the two linear equations using the PETSc library.

A general *solver partition* can be created to solve the linear equation, for example the mesh partition shown in Figure 3a. Another type of mesh partition is created according to the *poloidal partition* discussed in Section 4.2 and is shown in Figure 6, which is a special case of the PICpart partition shown in Figure 3a. In Figure 6, only 4 mesh partitions are created on the poloidal plane mesh for better visualization. As a first step in solving Equations 23 and 24 using a distributed mesh, we use the flux-surface aligned *poloidal partition* as shown in Figure 6 to divide the poloidal mesh domain. Usually, at least 10 to 20 *poloidal partitions* are created in a realistic simulation.

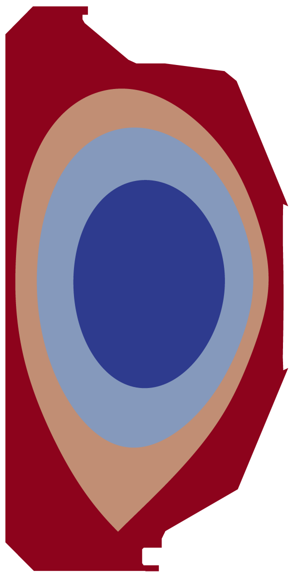


Figure 6: Example of a flux-surface aligned poloidal mesh partition used to solve the gyrokinetic Poisson equation. Here the mesh has four *poloidal partitions*.

4.4.2. Gyrokinetic Poisson equation solver work flow and numerical implementation

The linear equations, Eq.23 and Eq.24 are solved using the Krylov subspace (KSP) method [35, 37] in the PETSc linear system solver library. Due to the positive-definite nature of the linear matrix, the conjugate gradient (CG) method and the geometric algebraic multigrid (GAMG) preconditioner are used. The *DMPlex* object in PETSc is used to handle the unstructured distributed mesh.

Key aspects of the solution process are:

- In Eq.23 and Eq.24, the background density and magnetic field are constants throughout the simulation, hence the left had side linear matrices do not change during the simulation;
- Eq.23 and Eq.24 are solved independently, using CG method with GAMG preconditioner;
- The solutions of Eq.23 and Eq.24 are the equilibrium and turbulent components of the electrostatic potential, ϕ_0 and $\delta\phi$, respectively;
- On each poloidal plane, N/N_φ MPI ranks are used to solve the gyrokinetic Poisson equation, where N is the total number of MPI ranks in the simulation and N_φ is the number of poloidal planes used in the simulation;
- Values of ϕ_0 and $\delta\phi$ on the boundary of each poloidal plane are set to zero.

The linear equations can be solved either on the CPU or the GPU [37, 40]. In solving the linear equation on the GPU, the left-hand-side linear matrix, the solution vector, and the right-hand-side vector are all stored on the GPU memory. At the beginning of the solution process, the charge density field or the right-hand-side vector is stored on the GPU memory according to Omega mesh data structure. Similarly, at the end of the solution process, the electrostatic potential field or the solution vector is obtained and stored on the GPU memory according to the PETSc library. This is then directly copied to Omega mesh field on the GPU.

4.5. Electric field calculation on unstructured mesh

4.5.1. Numerical algorithm

The electric field vector **E** is needed at the particle position in particle push operation, and is defined on each mesh vertex. To calculate its value on each mesh vertex, the input is the electrostatic potential field defined on all mesh vertices in all poloidal planes. **E** is divided into two components, the poloidal component $\mathbf{E}_\perp$, and the parallel component $\mathbf{E}_\|$ which is parallel to the equilibrium magnetic field **B**,

$$\mathbf{E} = \mathbf{E}_\perp + \mathbf{E}_\|. \tag{25}$$

The parallel component of the electric field, $\mathbf{E}_\|$, is calculated through finite difference of electrostatic potential between neighboring poloidal planes. The calculation is performed along the direction of the equilibrium magnetic field, and the field-following operation discussed in Section 4.3.1 is also performed in the calculation. To ensure the accurate calculation of the poloidal component of the electric field, $\mathbf{E}_\perp$, on an unstructured mesh, the "curvilinear gradient" method [41] is used. The details of the numerical algorithm are discussed in Appendix A.

4.5.2. Implementation on the GPU

The electric field calculations discussed in Section 4.5.1 are performed on the GPU. The numerical implementation is based on the Omega "parallel_for" loop structure. The numerical operations can be performed on different mesh vertices simultaneously and independently. This operation only interacts with the mesh field.

Listing 2 gives a demonstrative code, describing how the electric field calculation is performed on the GPU. Other mesh field operations are performed similarly on the GPU using the Omega "parallel_for" loop structure. In the code segment, "o" stands for the Omega object, "parallel_for" is performed on the GPU and is the equivalent of the "for" loop operation on the CPU, and "mesh" is an Omega variable containing the mesh information and its topology including mesh vertices, triangular elements, and their adjacency relationships. "compute_efield" is the name of a user defined C++ language Lambda expression, and "ivert", "ivert1", and "ivert2" are the indices of three mesh vertices. With "parallel_for", the operations on different mesh vertices are performed simultaneously, where each hardware thread of the GPU corresponds to the operations on a single mesh vertex. "efield" stores a single component of the electric field on all mesh vertices, "pot" stores the electrostatic potential on all mesh vertices, and "nverts" is the total number of mesh vertices in a poloidal plane.

```
o::Write<o::Real> efield(mesh->nverts(), 0.0);
auto compute_efield = OMEGA_H_LAMBDA(const o::LO ivert) {
  o::LO ivert1 = ivert + 1;
  o::LO ivert2 = ivert - 1;
  efield[ivert] = (pot[ivert2] - pot[ivert1]) / 2;
};
o::parallel_for(nverts, compute_efield, "compute_efield");
```

Listing 2: Demonstrative code showing how the electric field calculation is performed on the GPU.

In Listing 2, the electric field on a mesh vertex with index "ivert", "efield[ivert]", depends only on the electrostatic potential values "pot" on two adjacent mesh vertices with index "ivert1" and "ivert2". This is not representative of the actual numerical algorithms discussed in Section 4.5.1 and in Appendix A, and is shown here as a simple demonstration.

4.6. Field gather

In the particle push operation, we need the electric field vector $\mathbf{E}$ at the simulation particle position. Electric field $\mathbf{E}$ is defined on each mesh vertex in each poloidal plane. Specific to ions, the electric field is gyroaveraged to account for its gyrokinetic nature. We denote its (i, q, j) component as $\mathbf{E}_{i,q,j}$. This is defined on a mesh vertex i in a poloidal plane with index j and toroidal angle

$\varphi_j = j\Delta\varphi$, and q is the index of a gyration orbit with radius R_q. R_q follows the same definition as in Section 4.3.1. i is index as $i = 1, 2, 3, ..., N_v$, q is indexed as $q = 0, 1, 2, ..., n_{gr}$, and j is indexed as $j = 0, 1, 2, 3, ..., (N_\varphi - 1)$. The electric field vector $\mathbf{E}$ is stored as a four dimensional array, where the fourth dimension corresponds to its three vector components.

Supposing we have a simulation particle P_k located at position (r_k, φ_k, z_k) with gyroradius ρ_k, it is bounded by two neighboring poloidal planes indexed as j_1 and j_2, with toroidal angle $\varphi_{j_1} = j_1\Delta\varphi$, $\varphi_{j_2} = j_2\Delta\varphi$, and,

$$\varphi_{j_1} \le \varphi_k < \varphi_{j_2}, \tag{26}$$

where $j_2 = j_1 + 1$. The gyroradius of particle i satisfies the following condition,

$$R_{q_1} \le \rho_i < R_{q_2}, \tag{27}$$

where q_1 is indexed as $q_1 = 0, 1, 2, ..., n_{gr}$ and $q_2 = q_1 + 1$. Following Equations 15 and 16, we denote the field-following projection of particle P_k onto poloidal planes j_1 and j_2 as points P_k^1 and P_k^2. The coordinates of points P_k^1 and P_k^2 are $(r_k^1, \varphi_k^1, z_k^1)$ and $(r_k^2, \varphi_k^2, z_k^2)$, respectively. Points P_k^1 and P_k^2 are located in triangular elements t_k^1 and t_k^2, where the indices of the three vertices of each triangular element are $(t_{k,1}^1, t_{k,2}^1, t_{k,3}^1)$ and $(t_{k,1}^2, t_{k,2}^2, t_{k,3}^2)$, and their barycentric coordinates are $(\chi_{k,1}^1, \chi_{k,2}^1, \chi_{k,3}^1)$ and $(\chi_{k,1}^2, \chi_{k,2}^2, \chi_{k,3}^2)$, respectively. The electric field at the location of the simulation particle P_k is denoted as $\mathbf{E}_k$, and is linearly interpolated from the electric field defined on the poloidal plane j_1 and j_2,

$$\begin{aligned}
\mathbf{E}_k = w_{\varphi,1}(&\chi_{k,1}^1 \mathbf{E}_{t_{k,1}^1,q_1,j_1} + \chi_{k,2}^1 \mathbf{E}_{t_{k,2}^1,q_1,j_1} + \chi_{k,3}^1 \mathbf{E}_{t_{k,3}^1,q_1,j_1} + \\
&\chi_{k,1}^1 \mathbf{E}_{t_{k,1}^1,q_2,j_1} + \chi_{k,2}^1 \mathbf{E}_{t_{k,2}^1,q_2,j_1} + \chi_{k,3}^1 \mathbf{E}_{t_{k,3}^1,q_2,j_1}) + \\
w_{\varphi,2}(&\chi_{k,1}^2 \mathbf{E}_{t_{k,1}^2,q_1,j_2} + \chi_{k,2}^2 \mathbf{E}_{t_{k,2}^2,q_1,j_2} + \chi_{k,3}^2 \mathbf{E}_{t_{k,3}^2,q_1,j_2} + \\
&\chi_{k,1}^2 \mathbf{E}_{t_{k,1}^2,q_2,j_2} + \chi_{k,2}^2 \mathbf{E}_{t_{k,2}^2,q_2,j_2} + \chi_{k,3}^2 \mathbf{E}_{t_{k,3}^2,q_2,j_2}).
\end{aligned} \tag{28}$$

In the above equation, the weights $w_{\varphi,1}$ and $w_{\varphi,2}$ are defined as,

$$w_{\varphi,1} = \frac{\varphi_{j_2} - \varphi_k}{\Delta\varphi}, \tag{29}$$

and,

$$w_{\varphi,2} = 1 - w_{\varphi,1}. \tag{30}$$

In case of electrons, we do not need to perform the gyroaverage operation for the electric field. The second dimension in the electric field vector $\mathbf{E}_{i,q,j}$ is ignored. It is denoted as $\mathbf{E}_{i,j}$, representing its value on a mesh vertex i on a poloidal plane with index j. The electric field at the location of the simulation particle P_k is denoted as $\mathbf{E}_k$, and is linearly interpolated from the electric field defined on the poloidal plane j_1 and j_2,

$$\begin{aligned}\mathbf{E}_k = w_{\varphi,1}(\chi^1_{k,1}\mathbf{E}_{t^1_{k,1},j_1} + \chi^1_{k,2}\mathbf{E}_{t^1_{k,2},j_1} + \chi^1_{k,3}\mathbf{E}_{t^1_{k,3},j_1} + \\ \chi^1_{k,1}\mathbf{E}_{t^1_{k,1},j_1} + \chi^1_{k,2}\mathbf{E}_{t^1_{k,2},j_1} + \chi^1_{k,3}\mathbf{E}_{t^1_{k,3},j_1}) + \\ w_{\varphi,2}(\chi^2_{k,1}\mathbf{E}_{t^2_{k,1},j_2} + \chi^2_{k,2}\mathbf{E}_{t^2_{k,2},j_2} + \chi^2_{k,3}\mathbf{E}_{t^2_{k,3},j_2} + \\ \chi^2_{k,1}\mathbf{E}_{t^2_{k,1},j_2} + \chi^2_{k,2}\mathbf{E}_{t^2_{k,2},j_2} + \chi^2_{k,3}\mathbf{E}_{t^2_{k,3},j_2}).\end{aligned} \tag{31}$$

4.7. Particle push

4.7.1. Numerical algorithm

In XGCm, the ion push operation is performed combining the two-step Runge-Kutta (RK2) and the fourth-order Runge-Kutta (RK4) methods. In each time step, two ion charge scatter operations are performed, corresponding to the two steps of the RK2 method. At the end of each ion charge scatter operation, the gyrokinetic Poisson equation is solved to obtain the updated electrostatic potential. The electric field is then calculated and used to update the particle property corresponding to each step of the RK2 method.

We denote the ion property as $\mathbf{Y}$,

$$\mathbf{Y} = (\mathbf{X}, v_\parallel) = (r, \varphi, z, v_\parallel), \tag{32}$$

and its time derivative as,

$$\frac{d\mathbf{Y}}{dt} = \frac{d\mathbf{Y}}{dt}(t, \mathbf{Y}) = (\frac{d\mathbf{X}}{dt}, \frac{dv_\parallel}{dt}). \tag{33}$$

Its value at time step t_n is denoted as $\mathbf{Y}_n = \mathbf{Y}(t_n)$, where $t_n = n\Delta t$, n is the time step index, and Δt is the simulation time step size. Its value at time step $t_{n+1} = t_n + \Delta t$ is denoted as $\mathbf{Y}_{n+1} = \mathbf{Y}(t_{n+1})$, and is calculated using the following equation,

$$\mathbf{Y}_{n+1} = \mathbf{Y}^{(2)}_{n+1}. \tag{34}$$

$\mathbf{Y}^{(j)}_{n+1}$ corresponds to the updated particle property at step j of the RK2 method, where j is indexed as $j = 1, 2$.

The RK4 method is embedded in each step of the RK2 method. In step j of the RK2 method, the ion property is calculated according to the following RK4 method,

$$\mathbf{Y}^{(j)}_{n+1} = \mathbf{Y}_n + \frac{\Delta t^{(j)}}{6}(k^{(j)}_1 + 2k^{(j)}_2 + 2k^{(j)}_3 + k^{(j)}_4), \tag{35}$$

where,

$$k^{(j)}_1 = \frac{d\mathbf{Y}^{(j-1)}}{dt}(t_n, \mathbf{Y}_n), \tag{36}$$

$$k_2^{(j)} = \frac{d\mathbf{Y}^{(j-1)}}{dt}(t_n + \frac{\Delta t^{(j)}}{2}, \mathbf{Y}_n + \Delta t^{(j)}\frac{k_1^{(j)}}{2}), \tag{37}$$

$$k_3^{(j)} = \frac{d\mathbf{Y}^{(j-1)}}{dt}(t_n + \frac{\Delta t^{(j)}}{2}, \mathbf{Y}_n + \Delta t^{(j)}\frac{k_2^{(j)}}{2}), \tag{38}$$

$$k_4^{(j)} = \frac{d\mathbf{Y}^{(j-1)}}{dt}(t_n + \Delta t^{(j)}, \mathbf{Y}_n + \Delta t^{(j)}k_3), \tag{39}$$

and,

$$\Delta t^{(j)} = \frac{j}{2}\Delta t. \tag{40}$$

In Equations 36, 37, 38, 39, $\frac{d\mathbf{Y}^{(1)}}{dt}(t, \mathbf{Y})$ corresponds to a time derivative calculated using the electric field information after step one of the RK2 method, while $\frac{d\mathbf{Y}^{(0)}}{dt}(t, \mathbf{Y}) = \frac{d\mathbf{Y}}{dt}(t, \mathbf{Y})$ corresponds to a time derivative calculated using the electric field at the beginning of each time step.

Similarly, the electron push operation is performed combining the RK2 and RK4 methods. Due to the smaller mass and faster movement of electrons compared to ions, electron subcycling [17, 18, 19, 42] is used to ensure the accuracy of the electron push calculations. In each step of the RK2 method, n electron subcycling steps are performed. Each electron subcycling step corresponds to a full cycle of the RK4 method. The subcycling time step size, $\Delta t_{sub}^{(j)}$, is determined according to the ratio of the ion and electron mass. We have,

$$\Delta t_{sub}^{(j)} = \frac{j}{2n}\Delta t, \tag{41}$$

where j is indexed as $j = 1, 2$ and corresponds to each step of the RK2 method. n is determined according to the following equation,

$$n = \lfloor\sqrt{\frac{m_i}{m_e}}\rfloor, \tag{42}$$

where m_i, m_e are the ion and electron mass, respectively, $\lfloor\cdot\rfloor$ is the floor function, and n is rounded as an integer.

4.7.2. Implementation on the GPU

Listing 3 provides a simplified code for the implementation of the particle push operation, describing how one step of the RK2 method is performed on the GPU. "ps" stands for the PUMIPic particle structure object [7], while "parallel_for" is performed on the GPU and is the equivalent of the "for" operation on the CPU. "ptcls" is the variable name of the particle structure object, containing all particle information. "push_lambda" is the name of a user defined C++ language Lambda expression and executes one step of the particle push operation. "e" is the triangular element index, "pid" is the particle index, and "mask" is an integer denoting whether the current memory location of "ptcls" object has an actual particle. "part_one" stores the information of a single particle, "magnetic_field" object stores the equilibrium magnetic field, and "electric_field" stores the electric field needed for particle push. "particle_push()" function updates the particle property of a single particle with index "pid". With "parallel_for", the push operations of different particles can be performed on different GPU hardware threads simultaneously.

```
// particle property for all particles
auto x_c = ptcls->get<xgcm::PTCL_COORDS>();
auto ph_c = ptcls->get<xgcm::PTCL_PH>();
auto ct = ptcls->get<xgcm::PTCL_CT>();
auto push_lambda = PS_LAMBDA(const int& e, const int& pid,
                             const int& mask) {
  if (mask) {

    // particle property for a particle with index "pid";
    // particle with "pid" is located in triangular element "e".
    Vector9d part_one = {x_c(pid,0), x_c(pid,1), x_c(pid,2),
                         ph_c(pid,0), ph_c(pid,1), ph_c(pid,2),
                         ct(pid,0), ct(pid,1), ct(pid,2)};
    // particle push operation for a single particle with index "pid".
    particle_push(magnetic_field, ..., electric_field, ..., part_one);
  }
};
ps::parallel_for(ptcls, push_lambda, "ion_push_op");
```

Listing 3: Simplified code showing how the particle push operation is performed on the GPU.

A key operation performed as part of the particle push step is to determine the element in which each particle is after the push, or if the particle has hit the boundary of the simulation domain. Since in a push step, particles only move a small distance with respect to the mesh size, the fact that Omega maintains a complete mesh topology supports the effective implementation of adjacency-based search [43] to determine the element in which the particles are contained after a push operation. In addition, Omega maintains classification of the mesh entities with respect to geometric model [44]. This classification information includes knowledge of which element bounding entities represent portions of plasma facing components and domain boundary, and thus supports effective determination of when and where particles hit a plasma facing component or domain boundary. As the simulation proceeds, particles will need to be migrated between PICparts due either to the fact that they are too close to the PICpart boundary, or to regain load balance.

5. XGCm Code Validation and Test

To validate and test the XGCm code, a series of unit tests were created during the code development process. These unit tests serve as the building blocks to ensure the correctness and accuracy of different numerical algorithms and their implementations in XGCm.

5.1. Validation of the XGCm code

To further validate the code, the circular geometry cyclone base case is performed to study the ion temperature gradient (ITG) turbulence [13], which corresponds to case 5 of Burckel et al. [12]. In this case, the cross-section of the Tokamak device is a circle, representing the closed-flux surface region of a realistic device. With a circular cross-section, the complexity associated with realistic geometrical boundary can be ignored.

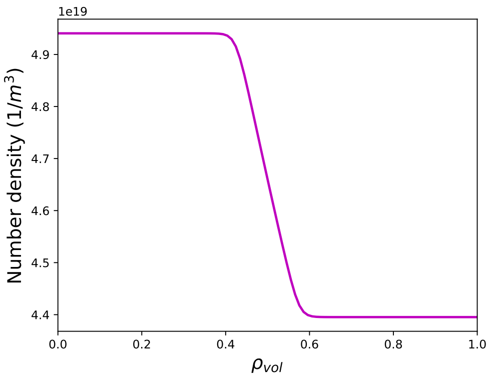
5.1.1. Initial and boundary conditions

In this case, a deuterium plasma is considered with an adiabatic electron assumption. Same temperature profile is used for both ion and electron. The initial ion number density is perturbed, corresponding to a single toroidal mode number of 24 with Guassian shape in both the radial and polodial directions within each poloidal plane [13].

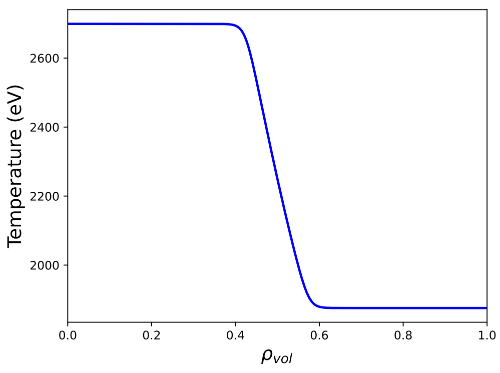
The equilibrium ion number density profile is given in Figure 7a and the equilibrium ion temperature profile is given in Figure 7b. They are defined according to the following function [13],

$$f(\rho_{vol}(\psi)) = f_0 \left[\frac{\cosh(\frac{\rho_{vol}-\rho_0+\delta f}{\Delta f})}{\cosh(\frac{\rho_{vol}-\rho_0-\delta f}{\Delta f})} \right]^{-\kappa_f \epsilon \Delta f/2}, \tag{43}$$

where function f denotes either number density n or temperature T. $\cosh(\cdot)$ is the hyperbolic cosine function. ρ_{vol} is the radial coordinate with $\rho_{vol}(\psi) = \sqrt{V(\psi)/V_{\text{LCFS}}}$, where $V(\psi)$ is the volume enclosed by a given flux surface ψ and V_{LCFS} is the volume of the last closed flux surface. At a given flux surface ψ or radial coordinate $\rho_{vol}(\psi)$, both number density and temperature are constant. f_0 is the reference value of either temperature or number density at radial coordinate $\rho_{vol}(\psi) = \rho_0 = 0.5$. The minor and major radius of the system are $a = 0.59$ meter and $R_0 = 1.68$ meter, respectively. The system aspect ratio is $\epsilon = a/R_0 = 0.35$. Magnetic field on the magnetic axis is $B_0 = 2.09$ T. κ_n and κ_T are the maximum value of logarithmic gradient of number density and temperature, respectively, with $\kappa_n = 2.22$ and $\kappa_T = 6.91$. The characteristic profile widths, δ and Δ, are chosen as 0.075 and 0.02, respectively. Further details of initial conditions can be found in the work of Merlo et al. [13].



(a) Number density



(b) Temperature

Figure 7: Background number density and temperature profiles. Here, ρ_{vol} is the radial coordinate and $\rho_{vol} = 1$ corresponds to the last closed flux surface.

5.1.2. Simulation setup

The simulation mesh is the same at different poloidal planes. Eight poloidal planes are used along the toroidal direction. A coarse poloidal plane mesh is given in Figure 8 for visualization purpose. A mesh convergence study is performed to ensure that the solution does not depend on the mesh size. For results reported here, a final fine mesh with 590,143 triangular elements and 296,046 mesh vertices in each poloidal plane is used. A total of 160 million ion particles are used in the simulation. This ensures that a sufficient number of particles are used and the simulation results are converged with respect to the number of simulation particles. A simulation time step size of 3.91×10^{-7} second is used and the simulation is run for a total of 200 time steps.

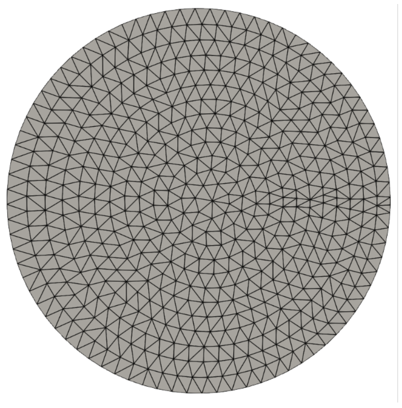


Figure 8: Simulation mesh in each poloidal plane. Coarse mesh is shown here for visualization purpose.

5.1.3. Simulation results and validation

The contour plot of the turbulent electrostatic potential on a poloidal plane cross-section is shown in Figure 9 for the final time step of 200. The mean-squared average of the turbulent electrostatic potential, $\bar{\phi}(t)$, is obtained as,

$$\bar{\phi}(t) = \frac{1}{N_v N_\phi} \sum_{j=0}^{N_\varphi - 1} \sqrt{\sum_{i=1}^{N_v} \phi^2(\mathbf{x}_i, \phi_j, t)}, \tag{44}$$

where $\phi(\mathbf{x}_i, \phi_j, t)$ is the electrostatic potential on vertex $\mathbf{x}_i$ at time t for a specific poloidal plane indexed as j with toroidal angle ϕ_j, N_v is the number of mesh vertices in each poloidal plane, and $N_\phi = 8$ is the number of poloidal planes used in the simulation.

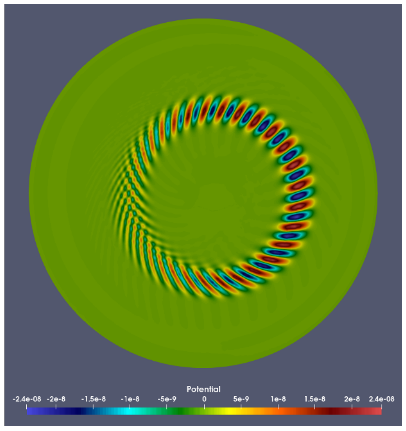


Figure 9: The contour plot of the turbulent electrostatic potential on one poloidal plane cross-section at time step 200.

The turbulence growth rate, $\gamma(t)$, is calculated as

$$\gamma(t) = \frac{d}{dt} \ln(\bar{\phi}(t)), \tag{45}$$

where $\ln(\cdot)$ is the natural logarithm function. The turbulence growth rate from XGCm simulation is compared with that of XGC in Figure 10. Overall, excellent agreement is observed and the average relative difference of turbulence growth rate is less than 0.7% in the linear growth stage.

5.2. ITG Simulation with DIII-D Geometry

As a further test of the XGCm code, realistic DIII-D geometry is used to perform the ITG turbulence simulation. A deuterium plasma is considered with an adiabatic electron assumption. Same temperature profile is used for both ion and electron. The equilibrium ion number density profile is given in Figure 11a and the equilibrium ion temperature profile is given in Figure

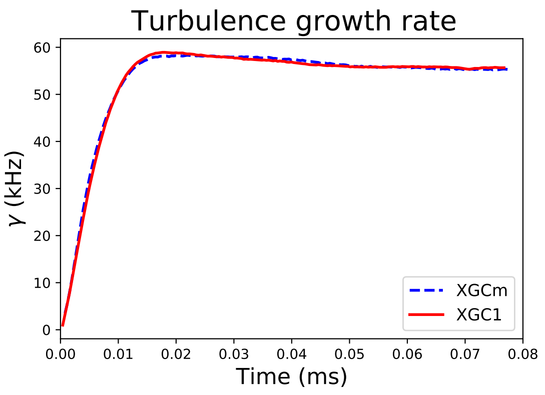
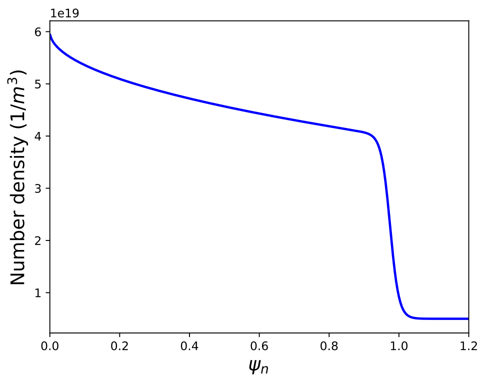
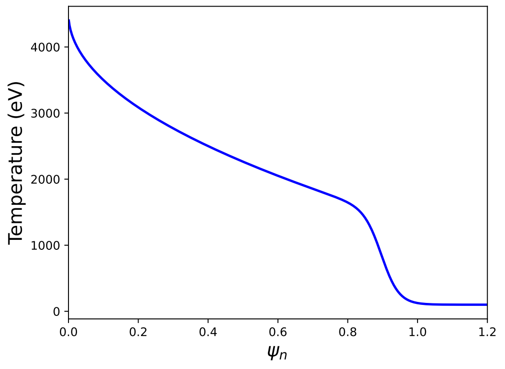


Figure 10: Comparison of the growth rate of the turbulent electrostatic potential between XGCm and XGC over time.

11b. In Figure 11a and 11b, $\psi_n = \frac{\psi}{\psi_{\text{LCFS}}}$ is the normalized flux surface, where ψ corresponds to a flux surface and ψ_{LCFS} corresponds to the last closed flux surface. Both number density and temperature are constant at a given flux surface ψ or ψ_n.



(a) Number density

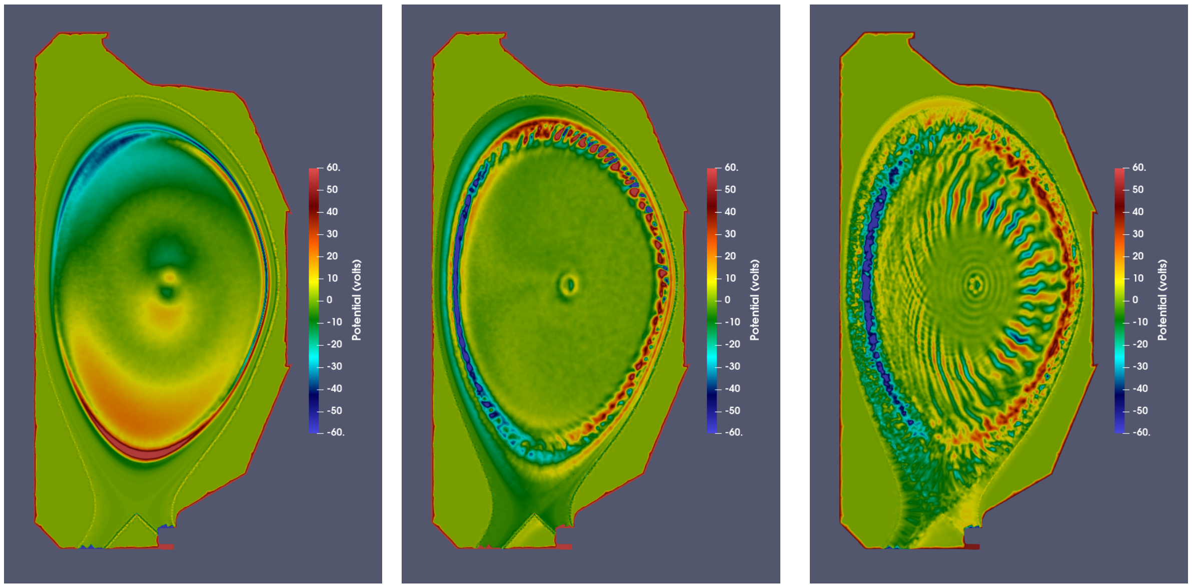


(b) Temperature

Figure 11: Background number density and temperature profile. Here, ψ_n is the normalized poloidal magnetic flux. $\psi_n = 1$ corresponds to the last closed flux curve.

In this case, 32 poloidal planes are used along the toroidal direction. The cross-section is the same as the DIII-D Tokamak as shown in Figure 2. In the simulation, a poloidal plane mesh with 102,359 triangular elements and 51,373 mesh vertices is used. A total of 1.6 billion ion particles are used in the simulation. This ensures that a sufficient number of particles are used and the simulation results are converged with respect to the number of simulation particles. A simulation time step size of 1.5×10^{-7} second is used, and the simulation is run for a total of 4400 time steps.

Three contour plots of the turbulent electrostatic potential on one poloidal plane cross-section are shown in Figure 12 for time step 100, 1000, and 4400, respectively. From the contour plots, it can be ssen that turbulence driven by the temperature gradient grows as time proceeds. The turbulent electrostatic potential first develops close to region with $\psi_n = 0.9$, where the background temperature has the largest gradient. The turbulence is then transported towards the core region with smaller ψ_n. This agrees with the physical picture of the ITG turbulence transport in the Tokamak [45].



(a) t=100 (b) t=1000 (c) t=4400

Figure 12: The contour plots of the turbulent electrostatic potential on one poloidal plane cross-section at different time steps.

6. Performance and Scaling

Using the Kokkos programming model, XGCm works fully on current GPU architectures. This includes the Summit supercomputer at the Oak Ridge National Laboratory [46], the Artificial Intelligence Multiprocessing Optimized System (AiMOS) supercomputer at Rensselaer Polytechnic Institute, and the Perlmutter supercomputer at the National Energy Research Scientific Computing Center (NERSC).

XGCm weak scaling study is performed on Summit supercomputer using the same cyclone ITG case as discussed in Section 5.1. In the weak scaling study, a poloidal plane mesh with 590,143 triangular elements and 296,046 mesh vertices is used. 25 million ion particles per GPU are used with an adiabatic electron assumption. The Sell-C-Sigma particle structure is used in PUMIPic to store the simulation particles. 64 poloidal planes are used along the toroidal direction. In each poloidal plane, 24 mesh partitions are used. The number of MPI ranks in each group partition is increased from 1 to 18 in different simulation cases. This results in larger amount of ion particles being simulated when more MPI ranks are used in a group partition. Each simulation is run for 5 time steps.

A group partition with 1 MPI rank is used in the simulation case with smallest number of computing nodes, while a group partition with 18 MPI ranks is used in the simulation case with largest number of computing nodes. Overall, 256 to 4,608 Summit computing nodes are used in the weak scaling study, equivalent to 1,536 to 27,648 GPUs, where 4,608 computing nodes correspond to full Summit system.

The weak scaling result is shown in Figure 13 where the time cost of major components of the gyrokinetic PIC method are shown. The result presented here is the averaged value, where the time cost of each component is first summed over all the MPI ranks and then averaged by the number of MPI ranks used in a simulation. The time value is then normalized by that of the simulation case with smallest number of computing nodes. Overall, each component in XGCm scales well with the increase in the number of computing nodes or number of MPI ranks. This shows that XGCm demonstrates good weak scaling.

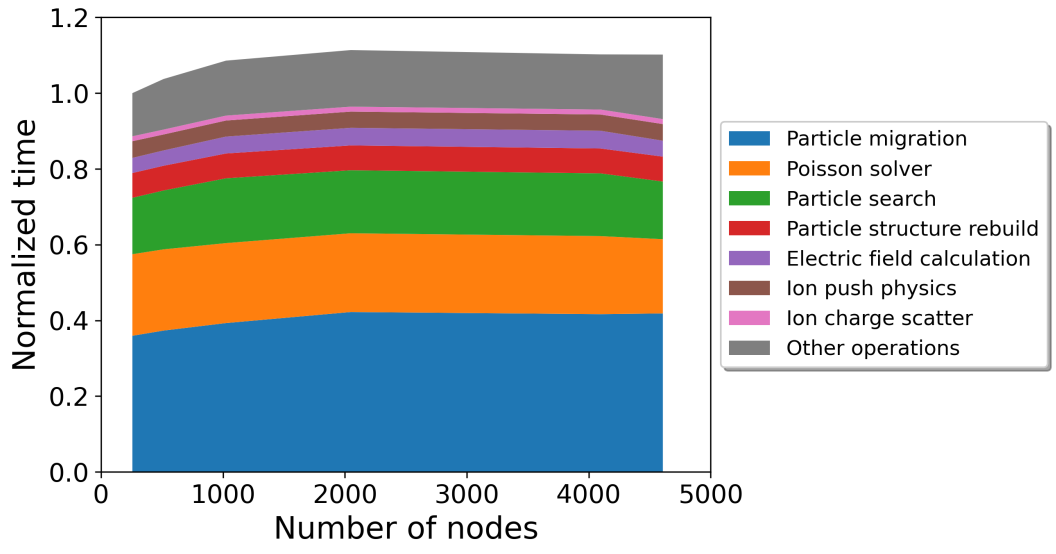


Figure 13: XGCm weak scaling result on Summit supercomputer using the cyclone ITG case.

We performed another study varying the number of particles used in each GPU without reaching the memory limit. In this study, the same cyclone ITG case is used, with 590,143 triangular elements per poloidal plane. Eight poloidal planes and a total of eight GPU devices are used in different simulation cases, corresponding to one GPU per poloidal plane. The motivation behind this study is to find the optimal workload for each GPU or the optimal number of particles per GPU. The result is shown in Figure 14, where the time cost per particle is plotted for different particle operations in XGCm. To ensure optimal GPU usage, the time cost per particle needs to be small. According to Figure 14, it is clear that a large number of particles are needed for each GPU in the current simulation case. This number also needs to be balanced by the GPU memory limit. On Summit, the optimal number of particles per GPU is in the range of 20 to 30 million for the current simulation case. Similar values are obtained for other simulation cases not discussed here.

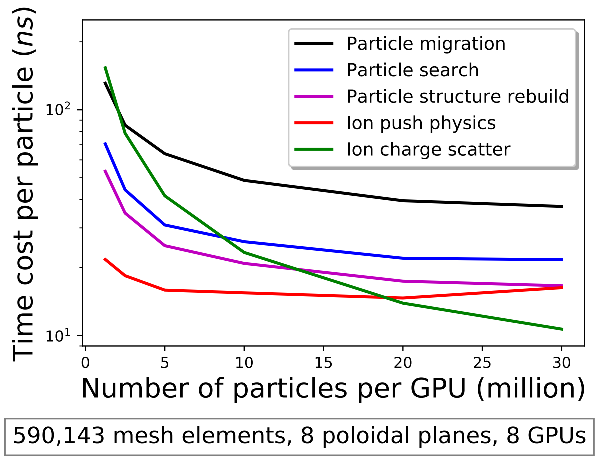


Figure 14: XGCm time cost results on Summit supercomputer using the Cyclong ITG test case. In this study, the number of particles used in each GPU is changed, while the total number of GPUs used is kept constant in different simulation cases.

At this point, the XGCm code development has focused on implementation of the capabilities needed for plasma simulations using delta-f gyrokinetic PIC method. Code profiling and performance improvements are not yet completed. Given that, an initial comparison indicates that the XGCm code exhibits good performance comparable with that of the XGC code. Using the same DIII-D simulation case as discussed in Section 5.2, we compared the simulation time cost of the two codes. We performed two comparisons on NERSC's Perlmutter supercomputer. In one case, we only used ions with an adiabatic electron assumption, while in another case, we used both ion and electron species. In both cases, 16 poloidal planes are used. In each poloidal plane, there are 102,359 triangular elements and 51,373 mesh vertices. 16 MPI ranks are used, corresponding to one MPI rank per poloidal plane or one GPU per poloidal plane. 64 million ions per GPU are used in the first case, while 40 million ions and 40 million electrons per GPU are used in the second case. In both cases, the Cabana particle structure [34] is used in PUMIPic [7]. The simulations are run for 20 time steps and the time cost is calculated by averaging the result over the number of time steps. The results are shown in Table 1. In both cases, the time cost of the XGCm code is smaller compared to that of the XGC code.

Table 1: Time cost comparison between the XGC and XGCm code (unit in seconds).

	XGC code	XGCm code
Adiabatic electron case	9.75	3.08
Kinetic electron case	30.35	24.01

7. Conclusions and Future Work

In this paper, we present XGCm, a validated gyrokinetic Particle-in-Cell (PIC) code that works fully on GPU accelerators. XGCm uses the open-source Omega mesh library and PUMIPic particle library for mesh and particle management, respectively. It achieves good scalability on Nvidia GPU devices and has good code portability using the Kokkos programming model. The gyrokinetic Poisson equation was discretized using the finite element method, and the solution of the resulting linear equation was obtained through the open source PETSc library. We validated the code using the circular geometry cyclone base case, comparing the turbulence growth rate of the electrostatic potential with results from the XGC code, which showed excellent agreement. We also demonstrated good weak scaling of the XGCm code using up to full system (27,648 GPUs) on Oak Ridge National Laboratory's Summit supercomputer.

Future work will focus on detailed performance analysis of the XGCm code and potential improvement under the current mesh and particle management framework. This paper is only the first step in a mesh-centric approach to the PIC method. As another important aspect, the data access to and the numerical computations on the mesh field should be consistent with those of the particle operations. Mesh-based operations should allow for better memory coalescence, suitable for GPU accelerators. In addition to having particle-to-mesh operation centered on each mesh element, mesh-to-particle and mesh-to-mesh operations should only rely on the local mesh field information. Future work will explore methods to achieve this objective, further improving the performance of current mesh-centric PIC method.

Acknowledgment

This research was supported by the U.S. Department of Energy, Office of Science, under awards DE-SC0021285 (FASTMath SciDAC Institute) and DE-SC0018275 (Unstructured Mesh Technologies for Fusion Simulation Codes). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Department of Energy. The authors would like to thank Dr. Eisung Yoon from Ulsan National Institute of Science and Technology for his work developing an earlier version of XGCm code working on the CPU. The authors would also like to thank researchers from the XGC team, including Drs. Choong-Seok Chang, Seung-Hoe Ku, Robert Hager, and Julien Dominski at the Princeton Plasma Physics Laboratory, and Dr. Aaron Scheinberg at Jubilee Development. Additionally, the authors would like to thank Usman Riaz at Rensselaer Polytechnic Institute for help with simulation mesh generations. The authors would like to acknowledge the work of Undergraduate Research Program students at Rensselaer Polytechnic Institute, including Matthew Christoff, Lucas Davis, Steven Spreizer, and Qing Zhu, in the development of PUMIPic library and the integration of Omega library with PETSc library,

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory and National Energy Research Scientific Computing Center, both of which are supported by the Office of Science of the U.S. Department of Energy. This research also used resources of the Center for Computational Innovations at Rensselaer Polytechnic Institute, particularly the Artificial Intelligence Multiprocessing Optimized System (AiMOS).

Appendix A. Gradient Calculation on Unstructured Mesh

Here, we describe the curvilinear gradient method [41] used in XGCm to calculate the gradient of the electrostatic potential on an unstructured mesh. The electrostatic potential at a position (r, z) on a poloidal mesh is $\phi(r, z)$ in the $(\hat{r}, \hat{z})$ coordinate system, where $\hat{r}$ and $\hat{z}$ are the two basis vectors as in the cylindrical coordinate system. Poloidal components of the electric field vector $\mathbf{E}_\perp$ are calculated as $\mathbf{E}_\perp = -\nabla_\perp \phi(r, z)$. $\mathbf{E}_\perp = (E_r, E_z)$, where E_r and E_z are the $\hat{r}$ and $\hat{z}$ component of the electric field, respectively.

Denote ϕ_i as the electrostatic potential on mesh vertex i with coordinate (r_i, z_i), and denote $E_{r,i}$ and $E_{z,i}$ as the $\hat{r}$ and $\hat{z}$ components of the electric field on mesh vertex i. i is indexed as $i = 1, 2, 3, ..., N_v$, where N_v is the number of mesh vertices in the poloidal mesh.

Appendix A.1. Gradient calculation on a triangular element

We first calculate the gradient of the electrostatic potential on any triangular element in the poloidal mesh. For a triangular element t in the poloidal mesh, assuming its three vertices are t_1, t_2, t_3 in counter clockwise direction. t is indexed as $t = 1, 2, 3, ..., M$, where M is the number of triangular elements in the poloidal plane mesh. In the $(\hat{r}, \hat{z})$ coordinate system, the coordinates of the three vertices are $(r_1, z_1), (r_2, z_2), (r_3, z_3)$. From the three vertices, we can form 2 vectors $\overrightarrow{l_1}$, $\overrightarrow{l_2}$ with,

$$\overrightarrow{l_1} = (r_2 - r_1, z_2 - z_1), \tag{A.1}$$

$$\overrightarrow{l_2} = (r_3 - r_1, z_3 - z_1). \tag{A.2}$$

We can form a curvilinear coordinate system using two basis vectors $\hat{\xi}$, $\hat{\eta}$ with origin located on vertex t_1. We have the freedom to choose the two basis vectors. Here we choose the two vectors $\overrightarrow{l_1}$, $\overrightarrow{l_2}$ as the basis vectors. In the curvilinear coordinate system ($\hat{\xi}$, $\hat{\eta}$), the coordinates of vertices t_1, t_2, and t_3 are $(0,0)$, (ξ_a, η_a), and (ξ_b, η_b), respectively. We have,

$$\xi_a = 1, \tag{A.3}$$

$$\eta_a = 0, \tag{A.4}$$

$$\xi_b = 0, \tag{A.5}$$

$$\eta_b = 1. \tag{A.6}$$

In the $(\hat{r}, \hat{z})$ coordinate system, the gradient of the electrostatic potential, $\nabla_\perp \phi$, can be written as,

$$\nabla_\perp \phi = \hat{r}\frac{\partial \phi}{\partial r} + \hat{z}\frac{\partial \phi}{\partial z}. \tag{A.7}$$

Similarly, in the curvilinear coordinate system ($\hat{\xi}$, $\hat{\eta}$), it can be written as

$$\nabla_\perp \phi = \hat{\xi}\frac{\partial \phi}{\partial \xi} + \hat{\eta}\frac{\partial \phi}{\partial \eta}. \tag{A.8}$$

In calculating the gradient of the electrostatic potential numerically, we assume that it is piecewise constant on each triangular element. We denote the electrostatic potential values on three vertices of the triangular elment t as ϕ_1, ϕ_2, and ϕ_3, respectively. When the curvilinear coordinate system ($\hat{\xi}$, $\hat{\eta}$) is transferred to a Cartesian coordinate system with orthogonal bases, we can easily calculate the gradient components on the triangular element t according to Equation A.8,

$$\frac{\partial \phi}{\partial \xi} = \frac{\phi_2 - \phi_1}{\xi_a} = \phi_2 - \phi_1, \tag{A.9}$$

$$\frac{\partial \phi}{\partial \eta} = \frac{\phi_3 - \phi_1}{\eta_b} = \phi_3 - \phi_1. \tag{A.10}$$

Using the chain rule, we can also obtain the $\hat{r}$ and $\hat{z}$ components of $\nabla_\perp \phi$ in the $(\hat{r}, \hat{z})$ coordinate system,

$$\frac{\partial \phi}{\partial r} = \frac{\partial \phi}{\partial \xi}\frac{\partial \xi}{\partial r} + \frac{\partial \phi}{\partial \eta}\frac{\partial \eta}{\partial r}, \tag{A.11}$$

$$\frac{\partial \phi}{\partial z} = \frac{\partial \phi}{\partial \xi}\frac{\partial \xi}{\partial z} + \frac{\partial \phi}{\partial \eta}\frac{\partial \eta}{\partial z}. \tag{A.12}$$

In matrix format, Equations A.11 and A.12 can be written as:

$$\begin{bmatrix} \frac{\partial \phi}{\partial r} \\ \\ \frac{\partial \phi}{\partial z} \end{bmatrix} = \begin{bmatrix} \frac{\partial \xi}{\partial r} & \frac{\partial \eta}{\partial r} \\ \\ \frac{\partial \xi}{\partial z} & \frac{\partial \eta}{\partial z} \end{bmatrix} \begin{bmatrix} \frac{\partial \phi}{\partial \xi} \\ \\ \frac{\partial \phi}{\partial \eta} \end{bmatrix} = (J^{-1})^T \begin{bmatrix} \frac{\partial \phi}{\partial \xi} \\ \\ \frac{\partial \phi}{\partial \eta} \end{bmatrix}. \tag{A.13}$$

In Equation A.13, J is the Jacobian transformation matrix from $(\hat{r}, \hat{z})$ coordinate system to $(\hat{\xi}, \hat{\eta})$ coordinate system,

$$J = \begin{bmatrix} \frac{\partial r}{\partial \xi} & \frac{\partial r}{\partial \eta} \\ \\ \frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} \end{bmatrix}. \tag{A.14}$$

Using geometric relations between the two coordinate systems on triangular element t, we can determine all elements of the Jacobian transformation matrix,

$$\frac{\partial r}{\partial \xi} = \frac{r_2 - r_1}{\xi_a} = r_2 - r_1, \tag{A.15}$$

$$\frac{\partial r}{\partial \eta} = \frac{r_3 - r_1}{\eta_b} = r_3 - r_1, \tag{A.16}$$

$$\frac{\partial z}{\partial \xi} = \frac{z_2 - z_1}{\xi_a} = z_2 - z_1, \tag{A.17}$$

$$\frac{\partial z}{\partial \eta} = \frac{z_3 - z_1}{\eta_b} = z_3 - z_1. \tag{A.18}$$

Substituting Equations A.14, A.15, A.16, A.17, and A.18 into Equation A.13, the electrostatic potential gradient $\nabla_\perp \phi$ on triangular element t is calculated in the $(\hat{r}, \hat{z})$ coordinate system,

$$\begin{bmatrix} \frac{\partial \phi}{\partial r} \\ \\ \frac{\partial \phi}{\partial z} \end{bmatrix} = \frac{1}{det(J)} \begin{bmatrix} (z_2 - z_3) & (z_3 - z_1) & (z_1 - z_2) \\ \\ (r_3 - r_2) & (r_1 - r_3) & (r_2 - r_1) \end{bmatrix} \begin{bmatrix} \phi_1 \\ \\ \phi_2 \\ \\ \phi_3 \end{bmatrix}. \tag{A.19}$$

Here $det(J)$ is the determinant of the Jacobian transformation matrix J.

Appendix A.2. Area-weighted gradient calculation on a mesh vertex

To obtain the gradient of electrostatic potential on each mesh vertex in the poloidal mesh, we use area-weighted average of the triangular element gradient. All triangular elements surrounding a mesh vertex are included in the average process. We denote the gradient of the electrostatic potential on a mesh vertex i as $\nabla_\perp \phi|_i = (\frac{\partial \phi}{\partial r}|_i, \frac{\partial \phi}{\partial z}|_i)$, and similarly denote the gradient of the electrostatic potential on a triangular element t as $\nabla_\perp \phi|^t = (\frac{\partial \phi}{\partial r}|^t, \frac{\partial \phi}{\partial z}|^t)$. $\nabla_\perp \phi|^t$ is calculated in Appendix A.1.

We denote the number of triangular elements surrounding vertex i as n_i. For a triangular element t, its area is a_t, where t is indexed as $k = 1, 2, 3, ..., n_i$. The $\hat{r}$ and $\hat{z}$ components of $\nabla_\perp \phi|_i$ are calculated using the area weighted average,

$$\frac{\partial \phi}{\partial r}|_i = \sum_{t=1}^{n_i} w_{i,t} \frac{\partial \phi}{\partial r}|^t, \tag{A.20}$$

$$\frac{\partial \phi}{\partial z}|_i = \sum_{t=1}^{n_i} w_{i,t} \frac{\partial \phi}{\partial z}|^t. \tag{A.21}$$

In the above equations, $w_{i,t}$ is the area weight and $w_{i,t} = a_t / \sum_{t=1}^{n_i} a_t$.

Appendix A.3. Gradient calculation suitable for numerical simulation

In this section, we rewrite the gradient calculation discussed in Appendix A.1 and Appendix A.2 using matrix-vector operations, so that it is suitable for numerical simulation. Written as matrix-vector products, the gradient calculation can be performed for each mesh vertex by looping through all mesh vertices on the CPU. Alternatively, it can be performed on multiple hardware threads of the GPU simultaneously. Here, we discuss the numerical algorithms suitable for the GPU.

We introduce the electrostatic potential vector Φ, which stores the values of electrostatic potential on all mesh vertices,

$$\Phi = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \cdots \\ \phi_j \\ \cdots \\ \phi_{N_v} \end{bmatrix}. \tag{A.22}$$

The length of vector Φ is N_v, which is the number of mesh vertices in each poloidal plane mesh. We also define a row vector R_i^t corresponding to a pair of vertex and triangular element, with index i and t, respectively,

$$R_i^t = \begin{bmatrix} g_{i,1} & g_{i,2} & g_{i,3} & \cdots & g_{i,N_v} \end{bmatrix}. \tag{A.23}$$

In vector R_i^t, i is vertex index and is indexed as $i = 1, 2, 3, ..., N_v$, while t is triangular element index and is indexed as $t = 1, 2, 3, ...M$.

The $\hat{r}$ component gradient term defined on a triangular element t is $\frac{\partial \phi}{\partial r}|^t$. It can be written as a product of R_i^t and Φ according to Equation A.19,

$$\frac{\partial \phi}{\partial r}|^t = R_i^t \Phi. \tag{A.24}$$

In reference to Equation A.19, only 3 elements of vector R_i^t are non-zero. The 3 non-zero elements of vector R_i^t correspond to the three vertices t_1, t_2, and t_3 of the triangular element t. They are g_{i,t_1}, g_{i,t_2}, and g_{i,t_3}, with,

$$g_{i,t_1} = \frac{1}{det(J_t)}(z_2 - z_3), \tag{A.25}$$

$$g_{i,t_2} = \frac{1}{det(J_t)}(z_3 - z_1), \tag{A.26}$$

$$g_{i,t_3} = \frac{1}{det(J_t)}(z_1 - z_2). \tag{A.27}$$

All other elements of R_i^t are 0. Here $det(J_t)$ is the determinant of the Jacobian transformation matrix J_t corresponding to triangular element t.

The $\hat{r}$ component gradient term defined on a mesh vertex i is $\frac{\partial \phi}{\partial r}|_i$. It can be similarly written as product of R_i^t and Φ, by substituting Equation A.24 into Equation A.20,

$$\frac{\partial \phi}{\partial r}|_i = \sum_{t=1}^{n_i} w_{i,t} R_i^t \Phi = R_i \Phi. \tag{A.28}$$

Here, the row vector R_i is defined as,

$$R_i = \sum_{t=1}^{n_i} w_{i,t} R_i^t. \tag{A.29}$$

As defined above, R_i is calculated as the area weighted average of row vectors R_i^t. The non-zero elements of row vector R_i correspond to all the mesh vertices of triangular elements surrounding mesh vertex i. We denote the maximum number of non-zero elements as q_i. q_i is the number of unique mesh vertices sharing a triangular element with vertex i. In general, q_i is less than $3n_i$, and is much smaller than the total number of mesh vertices N_v. Each non-zero element of R_i is determined through a combination of Equations A.25, A.26, A.27, and A.29. R_i is determined by the geometric relations between mesh vertices and triangular elements in the poloidal mesh.

Similar to Φ, we introduce the $\hat{r}$ component electric field vector $\mathbf{E}_r$, which stores the values of electric field on all discrete mesh vertices,

$$\mathbf{E}_r = \begin{bmatrix} E_{r,1} \\ E_{r,2} \\ \cdots \\ E_{r,j} \\ \cdots \\ E_{r,N_v} \end{bmatrix}, \tag{A.30}$$

We further define the $\hat{r}$ component gradient operator matrix R,

$$R = \begin{bmatrix} R_1 \\ R_2 \\ \dots \\ R_i \\ \dots \\ R_{N_v} \end{bmatrix}. \tag{A.31}$$

In the above equation, each row of matrix R is the row vector R_i. The size of matrix R is $N_v \times N_v$. Matrix R is a sparse matrix with a maximum of q_i non-zero elements in row i. Similar to row vectors R_i, matrix R is determined by the geometric relations between mesh vertices and triangular elements in the poloidal plane mesh. It can be precomputed once at the beginning of the simulation, stored, and used later when needed. 750

In reference to Equations A.28, A.30, and A.31, we can compute the $\hat{r}$ component electric field on all discrete mesh vertices, by applying the gradient operator matrix R to electrostatic potential vector Φ,

$$\mathbf{E}_r = -R\Phi. \tag{A.32}$$

We can similarly define the $\hat{z}$ component gradient operator matrix Z,

$$Z = \begin{bmatrix} Z_1 \\ Z_2 \\ \dots \\ Z_i \\ \dots \\ Z_{N_v} \end{bmatrix}. \tag{A.33}$$

Here row vector Z_i is defined as,

$$Z_i = \sum_{t=1}^{n_i} w_{i,t} Z_i^t, \tag{A.34}$$

and row vector Z_i^t is defined as,

$$Z_i^t = \begin{bmatrix} h_{i,1} & h_{i,2} & h_{i,3} & \dots & h_{i,N_v} \end{bmatrix}. \tag{A.35}$$

The three non-zero elements in row vector Z_i^t are defined as,

$$h_{i,t_1} = \frac{1}{det(J_t)}(r_3 - r_2), \tag{A.36}$$

$$h_{i,t_2} = \frac{1}{\det(J_t)}(r_1 - r_3), \tag{A.37}$$

$$h_{i,t_3} = \frac{1}{\det(J_t)}(r_2 - r_1). \tag{A.38}$$

Similarly introducing the $\hat{z}$ component electric field vector $\mathbf{E}_z$,

$$\mathbf{E}_z = \begin{bmatrix} E_{z,1} \\ E_{z,2} \\ \cdots \\ E_{z,j} \\ \cdots \\ E_{z,N_v} \end{bmatrix}. \tag{A.39}$$

we can compute $\mathbf{E}_z$ by applying the gradient operator matrix Z to electrostatic potential vector Φ,

$$\mathbf{E}_z = -Z\Phi. \tag{A.40}$$

In calculating the $\hat{r}$ and $\hat{z}$ components of the electric field $\mathbf{E}_r$ and $\mathbf{E}_z$ numerically, we can take advantage of the factor that gradient operator matrices R and Z are sparse matrices. By doing so, we can significantly reduce the numerical operations needed from $O(N_v \times N_v)$ to $O(\sum_{i=1}^{N_v} q_i)$.

References

[1] Committee on the Key Goals and Innovation Needed for a U.S. Fusion Pilot Plant, Bringing fusion to the U.S. grid (2021), Tech. rep., The National Academies of Sciences, Engineering and Medicine (2021). doi:https://doi.org/10.17226/25991.

[2] E. DÁzevedo, S. Abbott, T. Koskela, P. Worley, S. Ku, S. Ethier, E. Yoon, M. Shephard, R. Hager, J. Lang, J. Choi, N. Podhorszki, S. Klasky, M. Parashar, C. S. Chang, The fusion code XGC: Enabling kinetic study of multi-scale edge turbulent transport in ITER, in: T. Straatsma, K. Antypas, T. Williams (Eds.), Exascale Scientific Applications: Scalability and Performance Portability, CRC Press, Taylor & Francis Group, 2017, pp. 529–551.

[3] R. Khaziev, D. Curreli, hPIC: A scalable electrostatic Particle-in-Cell for Plasma-Material Interactions, Computer Physics Communications (2018).

[4] K. Madduri, K. Z. Ibrahim, S. Williams, E. Im, S. Ethier, J. Shalf, L. Oliker, Gyrokinetic toroidal simulations on leading multi- and manycore HPC systems, in: SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–12. doi:10.1145/2063384.2063415.

[5] B. Wang, S. Ethier, W. Tang, K. Z. Ibrahim, K. Madduri, S. Williams, L. Oliker, Modern gyrokinetic particle-in-cell simulation of fusion plasmas on top supercomputers, The International Journal of High Performance Computing Applications 33 (1) (2019) 169–188. doi:10.1177/1094342017712059.
URL https://doi.org/10.1177/1094342017712059

[6] W. Tang, Z. Lin, Global Gyrokinetic Particle-in-Cell Simulation, in: T. Straatsma, K. Antypas, T. Williams (Eds.), Exascale Scientific Applications: Scalability and Performance Portability, CRC Press, Taylor & Francis Group, 2017, pp. 507–528.

[7] G. Diamond, C. W. Smith, C. Zhang, E. Yoon, M. S. Shephard, PUMIPic: A mesh-based approach to unstructured mesh Particle-In-Cell on GPUs, Journal of Parallel and Distributed Computing 157 (2021) 1–12. doi:https://doi.org/10.1016/j.jpdc.2021.06.004.
URL https://www.sciencedirect.com/science/article/pii/S0743731521001337

[8] S. Ku, C. S. Chang, M. Adams, J. Cummings, F. Hinton, D. Keyes, S. Klasky, W. Lee, Z. Lin, S. Parker, et al., Gyrokinetic particle simulation of neoclassical transport in the pedestal/scrape-off region of a tokamak plasma, in: Journal of Physics: Conference Series, Vol. 46, IOP Publishing, 2006, p. 87.

[9] D. A. Ibanez, Conformal mesh adaptation on heterogeneous supercomputers, Ph.D. thesis, Rensselaer Polytechnic Inst., Troy, NY (2016).

[10] D. Ibanez, Omega_h GitHub repository (2016).
URL https://github.com/SNLComputation/omega_h

[11] H. Carter Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, Journal of Parallel and Distributed Computing 74 (12) (2014) 3202–3216. doi:https://doi.org/10.1016/j.jpdc.2014.07.003.
URL https://www.sciencedirect.com/science/article/pii/S0743731514001257

[12] A. Burckel, O. Sauter, C. Angioni, J. Candy, E. Fable, X. Lapillonne, On the effects of the equilibrium model in gyrokinetic simulations: from s-α to diverted MHD equilibrium, Journal of Physics: Conference Series 260 (1) (2010) 012006. doi:10.1088/1742-6596/260/1/012006.
URL https://dx.doi.org/10.1088/1742-6596/260/1/012006

[13] G. Merlo, J. Dominski, A. Bhattacharjee, C. S. Chang, F. Jenko, S. Ku, E. Lanti, S. Parker, Cross-verification of the global gyrokinetic codes GENE and XGC, Physics of Plasmas 25 (6) (2018) 062308. doi:10.1063/1.5036563.
URL https://doi.org/10.1063/1.5036563

[14] W. W. Lee, Gyrokinetic approach in particle simulation, Physics of Fluids 26 (1983) 556–562. doi:10.1063/1.864140.

[15] W. W. Lee, Gyrokinetic particle simulation model, Journal of Computational Physics 72 (1987) 243–269. doi:10.1016/0021-9991(87)90080-5.

[16] I. Manuilskiy, W. W. Lee, The split-weight particle simulation scheme for plasmas, Physics of Plasmas 7 (2000). doi:doi:10.1063/1.873955.

[17] S. Ku, C. S. Chang, P. H. Diamond, Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry, Nuclear Fusion 49 (11) (2009) 115021. doi:10.1088/0029-5515/49/11/115021.
URL https://doi.org/10.1088/0029-5515/49/11/115021

[18] C. S. Chang, S. Ku, P. H. Diamond, Z. Lin, S. Parker, T. S. Hahm, N. Samatova, Compressed ion temperature gradient turbulence in diverted tokamak edge, Physics of Plasmas 16 (5) (2009) 056108. doi:10.1063/1.3099329.
URL https://doi.org/10.1063/1.3099329

[19] S. Ku, C. S. Chang, R. Hager, R. M. Churchill, G. R. Tynan, I. Cziegler, M. Greenwald, J. Hughes, S. E. Parker, M. F. Adams, E. D'Azevedo, P. Worley, A fast low-to-high confinement mode bifurcation dynamics in the boundary-plasma gyrokinetic code XGC1, Physics of Plasmas 25 (5) (2018) 056107. doi:10.1063/1.5020792.
URL https://doi.org/10.1063/1.5020792

[20] Y. Idomura, S. Tokuda, Y. Kishimoto, Gyrokinetic Simulations of Tokamak Micro-Turbulence Including Kinetic Electron Effects, Journal of Plasma Fusion Research SERIES 6 (2004) 17–22.

[21] Y. Nishimura, Z. Lin, J. Lewandowski, S. Ethier, A finite element Poisson solver for gyrokinetic particle simulations in a global field aligned mesh, Journal of Computational Physics 214 (2) (2006) 657–671. doi:https://doi.org/10.1016/j.jcp.2005.10.011.
URL https://www.sciencedirect.com/science/article/pii/S0021999105004675

[22] S. E. Parker, W. W. Lee, A fully nonlinear characteristic method for gyrokinetic simulation, Physics of Fluids B: Plasma Physics 5 (1) (1993) 77–86. doi:10.1063/1.860870.
URL https://doi.org/10.1063/1.860870

[23] G. Colonna, Boltzmann and Vlasov equations in plasma physics, in: Plasma Modeling: Methods and Applications, IOP Publishing, 2016, pp. 1–23. doi:10.1088/978-0-7503-1200-4ch1.
URL https://dx.doi.org/10.1088/978-0-7503-1200-4ch1

[24] A. J. Brizard, T. S. Hahm, Foundations of nonlinear gyrokinetic theory, Rev. Mod. Phys. 79 (2007) 421–468. doi:10.1103/RevModPhys.79.421.
URL https://link.aps.org/doi/10.1103/RevModPhys.79.421

[25] R. Kleiber, R. Hatzky, A. Könies, A. Mishchenko, E. Sonnendrücker, An explicit large time step particle-in-cell scheme for nonlinear gyrokinetic simulations in the electromagnetic regime, Physics of Plasmas 23 (3) (2016) 032501. doi:10.1063/1.4942788.
URL https://doi.org/10.1063/1.4942788

[26] R. Hager, S. Ku, A. Y. Sharma, C. S. Chang, R. M. Churchill, A. Scheinberg, Electromagnetic total-f algorithm for gyrokinetic particle-in-cell simulations of boundary plasma in XGC, Physics of Plasmas 29 (11) (2022) 112308. doi:10.1063/5.0097855.
URL https://doi.org/10.1063/5.0097855

[27] T. S. Hahm, Nonlinear gyrokinetic equations for tokamak microturbulence, The Physics of Fluids 31 (9) (1988) 2670–2673. doi:10.1063/1.866544.
URL https://aip.scitation.org/doi/abs/10.1063/1.866544

[28] M. F. Adams, S. Ku, P. Worley, E. D'Azevedo, J. C. Cummings, C. S. Chang, Scaling to 150K cores: Recent algorithm and performance engineering developments enabling XGC1 to run at scale, Journal of Physics: Conference Series 180 (2009) 012036. doi:10.1088/1742-6596/180/1/012036.

[29] J. Dominski, S. Ku, C. S. Chang, Gyroaveraging operations using adaptive matrix operators, Physics of Plasmas 25 (5) (2018) 052304. doi:10.1063/1.5026767.
URL https://doi.org/10.1063/1.5026767

[30] Z. X. Lu, P. Lauber, T. Hayward-Schneider, A. Bottino, M. Hoelzl, Development and testing of an unstructured mesh method for whole plasma gyrokinetic simulations in realistic tokamak

geometry, Physics of Plasmas 26 (12) (2019) 122503. `doi:10.1063/1.5124376`.
URL `https://doi.org/10.1063/1.5124376`

[31] F. Zhang, R. Hager, S. Ku, C. S. Chang, S. C. Jardin, N. M. Ferraro, E. S. Seol, E. Yoon, M. S. Shephard, Mesh generation for confined fusion plasma simulation, Engineering with Computers 32 (2) (2016) 285–293.

[32] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Bishop, A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units, SIAM Journal on Scientific Computing 36 (5) (2014) C401–C423. `doi:10.1137/130930352`.

[33] M. Besta, F. Marending, E. Solomonik, T. Hoefler, SlimSell: A Vectorized Graph Representation for Breadth-First Search, in: Proceedings of the 31st IEEE International Parallel & Distributed Processing Symposium (IPDPS'17), IEEE, 2017, pp. 32–41.

[34] S. Slattery, S. T. Reeve, C. Junghans, D. Lebrun-Grandié, R. Bird, G. Chen, S. Fogerty, Y. Qiu, S. Schulz, A. Scheinberg, A. Isner, K. Chong, S. Moore, T. Germann, J. Belak, S. Mniszewski, Cabana: A Performance Portable Library for Particle-Based Simulations, Journal of Open Source Software 7 (72) (2022) 4115. `doi:10.21105/joss.04115`.
URL `https://doi.org/10.21105/joss.04115`

[35] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, J. Zhang, PETSc/TAO Users Manual, Tech. Rep. ANL-21/39 - Revision 3.17, Argonne National Laboratory (2022).

[36] S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith, Efficient Management of Parallelism in Object Oriented Numerical Software Libraries, in: E. Arge, A. M. Bruaset, H. P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhäuser Press, 1997, pp. 163–202.

[37] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. M. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith,

S. Zampini, H. Zhang, H. Zhang, J. Zhang, PETSc Web page, https://petsc.org/ (2022). URL https://petsc.org/

[38] J. Wesson, Tokamaks, 4th Edition, Oxford University Press, Oxford, UK, 2011.

[39] T. Moritaka, R. Hager, M. Cole, S. Lazerson, C. S. Chang, S. Ku, S. Matsuoka, S. Satake, S. Ishiguro, Development of a Gyrokinetic Particle-in-Cell Code for Whole-Volume Modeling of Stellarators, Plasma 2 (2) (2019) 179–200. doi:10.3390/plasma2020014.
URL https://www.mdpi.com/2571-6182/2/2/14

[40] R. T. Mills, M. F. Adams, S. Balay, J. Brown, A. Dener, M. Knepley, S. E. Kruger, H. Morgan, T. Munson, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, J. Zhang, Toward performance-portable PETSc for GPU-based exascale systems, Parallel Computing 108 (2021) 102831. doi:https://doi.org/10.1016/j.parco.2021.102831.
URL https://www.sciencedirect.com/science/article/pii/S016781912100079X

[41] E. Sozer, C. Brehm, C. C. Kiris, Gradient Calculation Methods on Arbitrary Polyhedral Unstructured Meshes for Cell-Centered CFD Solvers, 52nd Aerospace Sciences Meeting, AIAA 2014-1440. doi:10.2514/6.2014-1440.
URL https://arc.aiaa.org/doi/abs/10.2514/6.2014-1440

[42] J. Adam, A. Gourdin Serveniere, A. Langdon, Electron sub-cycling in particle simulation of plasma, Journal of Computational Physics 47 (2) (1982) 229–244. doi:https://doi.org/10.1016/0021-9991(82)90076-6.
URL https://www.sciencedirect.com/science/article/pii/0021999182900766

[43] W. Celes, G. H. Paulino, R. Espinha, A compact adjacency-based topological data structure for finite element mesh representation, International journal for numerical methods in engineering 64 (11) (2005) 1529–1556.

[44] M. W. Beall, M. S. Shephard, A general topology-based mesh data structure, International Journal for Numerical Methods in Engineering 40 (9) (1997) 1573–1596.

[45] J. Candy, R. E. Waltz, Anomalous Transport Scaling in the DIII-D Tokamak Matched by Supercomputer Simulation, Phys. Rev. Lett. 91 (2003) 045001. doi:10.1103/PhysRevLett.91.045001.
URL https://link.aps.org/doi/10.1103/PhysRevLett.91.045001

[46] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. Oral, D. E. Maxwell, V. G. V. Larrea, A. Bertsch, R. Goldstone, W. Joubert, C. Chambreau, D. Appelhans, R. Blackmore, B. Casses, G. Chochia, G. Davison, M. A. Ezell, T. Gooding, E. Gonsiorowski, L. Grinberg, B. Hanson, B. Hartner, I. Karlin, M. L. Leininger, D. Leverman, C. Marroquin, A. Moody, M. Ohmacht, R. Pankajakshan, F. Pizzano, J. H. Rogers, B. Rosenburg, D. Schmidt, M. Shankar, F. Wang, P. Watson, B. Walkup, L. D. Weems, J. Yin, The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems, in: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, 2018, pp. 661–672. `doi:10.1109/SC.2018.00055`.