PUMIPic: A mesh-based approach to unstructured mesh Particle-In-Cell on GPUs

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

[a]*Rensselaer Polytechnic Institute, 110 8th St, Troy, New York 12180 USA*
[b]*Ulsan National Institute of Science and Technology, 50 UNIST-gil, Eonyang-eup, Ulju-gun, Ulsan, South Korea*

*Corresponding author: E-mail: diamog@rpi.edu

Abstract

Unstructured mesh particle-in-cell, PIC, simulations executing on the current and next generation of massively parallel systems require new methods for both the mesh and particles to achieve performance and scalability on GPUs. The traditional approach to implementing PIC simulations define data structures and algorithms in terms of particles with a full copy of the unstructured mesh on every process. To effectively scale the unstructured mesh and particles, mesh-based PIC uses the unstructured mesh as the predominant data structure with the particles stored in terms of the mesh entities. This paper details the PUMIPic library, a framework for developing efficient and performance-portable mesh-based PIC simulations on GPU systems. A pseudo physics simulation based on a five-dimensional gyro-kinetic code for modeling plasma physics is used to examine the performance of PUMIPic. Scaling studies of the unstructured mesh partition and number of particles are performed up to 4096 nodes of the Summit system at Oak Ridge National Laboratory. The studies show that mesh-based PIC can utilize a partitioned mesh and maintain scaling up to system limitations.

Keywords: Particle-In-Cell, Unstructured Mesh, GPU

Introduction

An important class of multiscale simulation is the Particle-in-Cell (PIC) method in which particle tracking to capture fine-scale behaviors is coupled to fields defined in terms of PDEs represented at the scale of the overall domain. The continued advancement of massively parallel computing technologies along with the ability to effectively scale PIC calculations on those systems is supporting the effective application of PIC codes to the modeling of plasmas in fusion reactors [1, 2, 3, 4, 5], linear accelerators [6] and other systems. An increasing number of high-performance plasma simulation codes are currently, or planning to, take advantage of unstructured mesh methods that can provide the required levels of field accuracy over general domains using the fewest number of elements. The advantages of unstructured meshes come with a cost of larger, more complex, data structures and more complex algorithms to achieve parallel scalability.

PIC methods are implemented as a time advancing procedure in which the position of particles is tracked as they move through a domain, driven by a field that is typically a function of the position of the particles, and thus that field evolves as the particles move. In the coupled case there are four steps carried out in each time advance [2, 3, 4, 7, 8, 9, 10]. Those steps are:

Field to Particle: The values of the current mesh-based fields that drive the particles are associated with each particle through an appropriate interpolation procedure.

Particle Push: The particles are moved, or if you will, pushed, to a new location as a function of the field and time step size.

Charge Deposition: The "charge" information associated with the particles is then related to the domain definition such that the forcing function driving the field evolution is updated.

Field Solve: The partial differential equations (PDEs) governing the field is then solved using this updated forcing function.

Since the PDEs governing the domain fields can rarely be solved in closed form over the domains of interest, the needed fields are solved numerically over a spatial discretization that can range from a uniform grid to a graded unstructured mesh. Figure 1 gives a basic graphical description of the four PIC steps on a mesh. A key operation that needs to be executed after each Particle Push step, is the determination of which grid cell, or mesh element, each particle is within since this information is required by the Field to Particle and Charge Deposition steps. In the case where the PIC method is tracking particles through a uniform grid, this operation is trivially defined by the particle coordinates. In this case, it is natural to drive the PIC code steps using an independent particle data structure. In large scale computations executed on massively parallel computers, this particle data structure must be distributed over the memory spaces of a large number of processes.

When unstructured meshes are used, the determination of the element a particle is located in after a push is not a one-step algebraic evaluation. Instead, it requires a numerical evaluation process that includes explicit consideration of the geometric

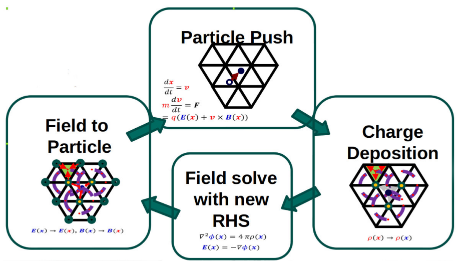
Particle Push
$\frac{d\mathbf{x}}{dt} = \mathbf{v}$
$m\frac{d\mathbf{v}}{dt} = \mathbf{F} = q(\mathbf{E}(\mathbf{x}) + \mathbf{v} \times \mathbf{B}(\mathbf{x}))$
Charge Deposition
$\rho(\mathbf{x}) \rightarrow \rho(\mathbf{x})$
Field solve with new RHS
$\nabla^2\phi(\mathbf{x}) = 4\pi\rho(\mathbf{x})$
$\mathbf{E}(\mathbf{x}) = -\nabla\phi(\mathbf{x})$
Field to Particle
$\mathbf{E}(\mathbf{x}) \rightarrow \mathbf{E}(\mathbf{x}), \mathbf{B}(\mathbf{x}) \rightarrow \mathbf{B}(\mathbf{x})$

Figure 1: Depiction of the 4 main steps involved in an iteration of the PIC loop.

definition of the elements. To avoid the need to evaluate the required geometric calculations involved with determining particle containment for the entire list of elements until the element the particle is within is found, some form of search mechanism that limits the number of elements that must be considered is employed.

A common approach to the implementation of a PIC method on an unstructured grid is to extend the particle data structure to maintain knowledge of the mesh element a particle is in at the current time and to introduce an independent mesh data structure. When using independent particle and mesh data structures it is common to also introduce a spatial data structure such as a uniform grid or spatial tree to support an efficient search process to find the element the particle is contained in after a push. This approach can achieve good scaling and is reasonably performant, particularly for meshes with little or no gradation, when there is a copy of the entire mesh maintained in each process memory space. However, having a copy of the entire mesh on each process has the obvious drawback of not being scalable with growing the mesh size past a point dictated by the available memory on a process for the mesh. Since PIC implementations employ methods in which the number of particles is two or even three orders of magnitude larger than the number of mesh elements, this approach has been satisfactory. However, as more complex, both in terms of physics and geometric complexity, systems are considered, there is a desire to employ distributed meshes with millions of elements that are strongly graded. The use of independent particle and mesh data structures which are both distributed complicates the implementation of efficient memory access patterns and, in the case of graded meshes, complicate the search process of efficiently determining what element a moved particle is within.

The central goal of the approach presented in this paper is to be able to support having both the particles and mesh distributed over the memory spaces of the processes used in the parallel execution of the PIC simulation. Any effective implementation of such an approach must carefully consider appropriate mechanisms to control interprocess communications that may arise through the interactions of the particles and the mesh. A possibility for doing this in the case of independent particle and mesh data structures is to employ mechanisms that distribute closely related mesh elements and particles. For example, the same background grid used to support the element search process could be used. This paper presents an alternative approach in which the mesh data structure is the core data structure and the particles can only be accessed through the mesh. Since this approach maintains an explicit relationship of the particles to the mesh, there is no need for a secondary structure and search process to support maintaining the relationship between them.

With the majority of top supercomputers including some form of accelerators, in most cases being GPUs, it is vital to employ data structures and algorithms that effectively take advantage of the accelerators. For existing simulations running on CPUs in serial and many-core codes parallelized using threads, the common approach to an accelerator-based system is to focus on porting individual sections of code or kernels one at a time to the GPU. In many cases, it is found that to gain performance, it is necessary to modify the underlying data structures and layouts based on the needs of the section of code being worked on. Since the approach presented in this paper is based on a fundamental change in the underlying relationship of the mesh and particles data, the development of an effective implementation on accelerated systems required development of appropriate mesh and particle data structures, as well as the definition of the core kernel operations, that can support the effective execution of PIC operations. This paper describes the Parallel Unstructured Mesh Infrastructure for PIC library (PUMIPic) which is the result of that development.

The paper is structured as follows. First, data structures for the unstructured mesh and particles for efficient usage in terms of on-process parallelism and multi-process parallelism are detailed. Next, key operations and algorithms included in the PUMIPic library for implementing PIC simulations are discussed. To demonstrate the performance and scalability of PUMIPic, its use within pseudo physics simulation that includes the core PIC operations that interact with PUMIPic are presented. Finally, future directions of research are outlined.

Related work

Designing data structures for simulations running on GPU accelerated machines has new challenges unlike those designed for multi-core machines and, to some extent, many-core machines. Data structures must take into account the massive multithreading capability of GPUs and be conscious of the layout of memory on the device and access patterns associated with the simulation to gain the most performance on these devices. Furthermore, with the upcoming machines featuring different GPU architectures, it is vital to construct data structures that can be used on the different hardware and to be performance portable either automatically or through tuning of parameters.

Many different data structures have been proposed to maximize the locality and alignment of memory accesses on GPUs. For matrix applications, several alternatives to the commonly used Compressed Sparse Row structure have been analyzed [11]. The ELLPACK [12] structure adds zero entries to ensure all rows have an equal length to align memory accesses. The structure also stores memory vertically which allows threads of a warp to work on a row while accessing a contiguous block of memory. The ELLPACK structure is best used on structured matrices due to the extra memory and computation added by the non-zero entries. For matrices without a uniform structure, a variant ELLPACK-R is suggested [13]. ELLPACK-R groups the rows into chunks based on the SIMD width of the target hardware. The zero entries added to ELLPACK-R are such that all rows within a chunk have the same length. This improves from the original ELLPACK by reducing the memory footprint of the zero entries as well as reducing computations on zero entries. Further improvements to reduce the padding are suggested by Kreutzer et al. [14, 15] with the Sell-C-σ structure. The Sell-C-σ sorts the rows by the number of filled entries which when performing full sorting minimizes the number of zero entries added to the structure.

In PIC, different implementations for GPUs have explored structures for storing particles. Burau et al. [16] use a linked particle list to implement a PIC simulation with structured meshes on GPUs. Each mesh element points to the list of particles within the element. This structure allows particles to move quickly between the mesh elements, however as Burau et al. explain when particles move between elements the order of memory accesses becomes fragmented and leads to a reduction in the performance of particle operations. A three-stage memory hierarchy [17] has been used to alleviate the fragmentation problem by tiling particle data within a set of mesh elements.

Cabana [18] is a library for the storage of particle data for the traditional approach to PIC simulations. Cabana uses an array of structures of arrays (AoSoA) to store particle information aligned with and sized by the target hardware. Each struct of arrays (SoA) maintains particle information for the SIMD width number of particles. Then an array of the SoAs is created to store all particles. This results in the memory accesses being aligned with the hardware and execution of the GPU.

Data structures for mesh-based PIC

For designing unstructured mesh-based PIC simulations, two core data structures are needed: a mesh data structure and a particle data structure. An unstructured mesh data structure on GPUs has to account for the complexity of storing and efficient access to the necessary mesh adjacencies and field information. The particle data structure must focus on efficient memory access of particle information and the interactions with the mesh entities and fields.

With the three upcoming DOE leadership-class supercomputers, Aurora, Frontier, and Perlmutter, each using a different vendor GPU with specific language and hardware functionality, it is highly desirable to implement codes for GPUs to be portable across the different hardware. To address the ability to port to different hardware, all data structures in PUMIPIC use the Kokkos library [19] for data management and parallel execution on the GPU. The remainder of this section covers the design choices for storage of the unstructured mesh and particles for on process parallel execution and multi-process parallelism.

Mesh data structure

PUMIPic's mesh data structure is built on top of the Omega_h library [20, 21]. Omega_h is a performance portable unstructured mesh library designed for GPUs supporting simplex elements in two or three dimensions. The structure is device default such that all mesh entities and field data are allocated and remains on the device unless explicitly transferred to the host. Omega_h provides access to all mesh adjacencies through compact arrays ordered to align adjacent entities.

Omega_h is used to store and operate on the mesh elements and fields on a single process and GPU. In this regard, each process has its own instance of an Omega_h mesh that does not know of the mesh on other processes. As discussed in the next section, communications between mesh and fields between processes are handled through PUMIPic to achieve the level of parallel operations desired for a PIC simulation.

Mesh partitioning for PIC

Partitioning an unstructured mesh is both a challenging and important problem. The partitioning must have close balancing of computation across all processes while also minimizing the communication costs that results from partitioning the mesh. Many methods exist in practice for partitioning a mesh to balance computation and communication costs. Most notable are multi-level graph methods [22, 23], geometric methods [24, 25, 26], and diffusive methods [27, 28, 29].

For PIC simulations, the difficulties of partitioning increase due to having to balance mesh computation and communication along with particle computation and migrations across processes. Typically, the mesh is partitioned to optimize either the field solve step or the interactions between the mesh and particles. Partitioning for the mesh field solve considers evenly distributing the number of degrees of freedom across processes and minimizing the number of degrees of freedom on the partition model boundary. Partitioning for particles targets allowing particles to be distributed evenly across processes and minimizing the migration of particles as the simulation evolves. Constructing the partition along the principal direction of motion of particles is ideal to maintain particle balance and minimize communications. Figure 2 shows two partitions of a two-dimensional mesh. The left mesh is partitioned using a multi-level graph method for the field solve. The right mesh is partitioned along flux faces where particles predominantly move within.

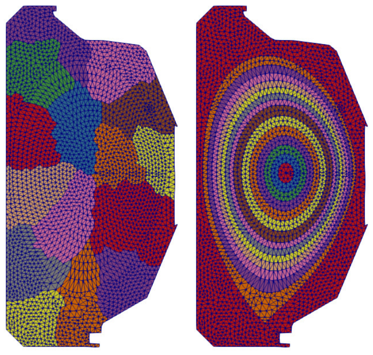

Figure 2: Two-dimensional unstructured mesh partitioned using multi-level graph method for field solve (left) and partitioned along principal direction of particle motion (right).

Regardless of the approach to partitioning the mesh into a set of parts, to be referred to as PICparts, it is important to minimize the communication of particle information by ensuring that no communication is required during the execution of a single push step. By defining each PICpart to have a core and substantial mesh overlap, and requiring that the particles pushed on each PICpart in a single push step start in a location far enough from the boundaries of the PICpart, in what is referred to as the PICpart's safe zone, we ensure that the particles end up in an element on the PICpart after the current push step. Note that particles that at the end of a push have moved into an element outside the current owning PICpart's safe zone will be migrated to a PICpart for which that element is within its safe zone before the next push operation.

The first step in the definition of the PICparts is to define a unique partitioning of the mesh such that each element is in a single part and the elements in a part are a good set of neighbors with respect to where particles will move in a push operation. Each of these unique parts will form the core part of a PICpart. The remainder of the PICpart can be defined as any set of neighboring elements that provide a sufficient buffer such that at least the core part will constitute the PICpart safe zone. In general, it is desirable that the buffer is larger, so the safe zones of each PICpart will overlap to help control the amount of particle migration required between steps and provide more flexibility in the application of dynamic load balancing.

There are multiple options for selecting the set of buffer elements to be used for a PICpart. An obvious choice is to select a number of neighboring elements in the same way remote copies are defined in many PDE discretization methods [30]. For the PIC applications developed to date, it was found that this would require four layers of elements and it was determined that the data storage and update requirements if done as standard remote copies were problematic.

The alternative approach used herein defines each PICpart as a core part and a sufficient set of the neighboring parts to ensure that there is sufficient buffer with respect to the core part. Although this approach yields PICparts with more elements than a layerwise buffering would produce, it requires substantially less total data since it does not require maintaining remote copy data associated with each element. This approach also yields large safe zones that reduce particle migration and provides greater load balancing options.

In cases where the memory requirement to store the entire mesh and required fields on each PICpart is low, it is convenient for each PICpart to store the entire mesh. In this case, a specific PICpart is a core and its buffer is the remaining PICparts that define the mesh.

When the amount of memory required to store the entire mesh and required fields is such that it is desirable to distribute the mesh, the buffer for each PICpart is only a subset of the the initial set of parts defined by executing a breadth-first traversal (BFT) out some number of layers from the boundary of the PICpart's core part. Each part reached during the BFT is buffered in the PICpart. An example PICpart constructed using 4 layers of BFT buffering is shown in Figure 3. The PICpart is for the core part labeled A which then fully buffers each core region within 4 iterations of BFT including the non-adjacent part labeled B since elements in part B are included in the third layer determined by the BFT. The safe zone is also constructed using BFT with fewer layers than the buffering where the core region and each element reached by the BFT is in the safe zone. However, this safe zone construction can be overridden by the simulation if a different definition is required.

The BFT algorithm is implemented using a bottom-up approach [31]. In the bottom-up approach, each iteration of the BFT processes has every mesh element check its neighbors to see if it is visited this iteration. This is performed through a selected lower dimension of entities called the bridge dimension. Each entity of the bridge di-

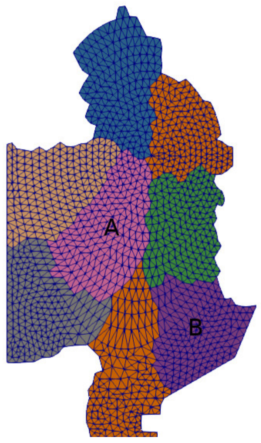

Figure 3: PICpart generated for the core part A using 4 layers of breadth-first traversal (right) of the left partition from Figure 2. Note that the part labeled B is also fully buffered.

mension first checks its upward adjacencies to elements if any have been visited in a previous iteration. If at least one element has been visited then all of the upward adjacent elements are visited in the current iteration of the BFT.

The bottom-up approach is better than the more traditional top-down approach for many-core and GPU architectures when the frontier of the BFT is sufficiently large. Since the starting point for the BFT is the entire core region, the iterations start with a sufficiently large frontier such that using the bottom-up method is more performant than the top-down.

Algorithm 1 provides pseudocode for constructing PICparts using bottom-up BFT iterations. Lines 2-6 setup the initial state for the BFT. The visited array is filled with true values for each element in the core part. The BFT algorithm is iteratively run on lines 7-25. The bottom-up step on lines 8-20 runs each vertex checking if any of its upward adjacent elements were visited in a previous iteration then setting all upward adjacent elements to visited. At the end of each iteration, the *visited_next* array is copied to the *visited* array to set up the next iteration on line 21. Lines 22-24 constructs the safe zone from the visited elements after the specified number of iterations for the safe zone is performed. Then the mesh parts that include any visited elements are designated to make up the buffer are collected at the end on lines 26-31.

Particle data structure

Particle information such as position and velocity must be stored on the GPU in a structure that allows efficient access. For mesh-based PIC simulations, the main requirement for the particle data structure is to group particles by the parent mesh element. For performance on accelerators, it is important that the structure can be evenly distributed to threads and mapped to the hardware memory layout and the access pattern. As a library, it is also important that the particle data structure is tunable to support different application characteristics. The characteristics present in the target

Algorithm 1 pseudocode for pull-based BFT to construct buffer region and safe zone.

```
function BFT($mesh$, $safe\_layers$, $buffer\_layers$, $bridge\_dim$)
    $visited$ = DeviceArray($mesh$.num_elems())
    $visited\_next$ = DeviceArray($mesh$.num_elems())
    parallel_for Element $e$ in mesh do
        $visited[e]$ = $mesh$.isCoreRegion($e$)
    end parallel_for
    for $iter$ = 0 to $buffer\_layers$ do
        parallel_for Entity $e$ of dimension $bridge\_dim$ in $mesh$ do
            $visit = false$
            for all Element $n$ adjacent to $e$ do
                if $visited[n]$ then
                    $visit = true$
                end if
            end for
            if $visit$ then
                for all Element $n$ adjacent to $e$ do
                    $visited\_next[n] = true$
                end for
            end if
        end parallel_for
        $visited \leftarrow visited\_next$
        if $iter = safe\_layers$ then
            $safe\_zone \leftarrow visited\_next$
        end if
    end for
    $buffer\_cores$ = DeviceArray(number of processes)
    parallel_for Element $e$ in $mesh$ do
        if $visited[e]$ then
            $buffer\_cores[mesh$.owner($e$)$] = true$
        end if
    end parallel_for
    return $buffer\_cores$, $safe\_zone$
end function
```

applications of PUMIPic include: up to 10,000 particles per element, uniform and non-uniform particle distributions, particles will only traverse through a small number of neighboring elements during a single iteration, and simulation defined representations of particle data including multiple definitions of particles for a single simulation.

Sell-C-σ

The choice of data structure in PUMIPic to satisfy the requirements is a structure from Kreutzer et al. called the Sell-C-σ (SCS) [15]. The structure was designed for efficient use of sparse matrices for tunable performance on different architectures including many-core and GPUs. The SCS stems from the commonly used Compressed-

Sparse-Row (CSR) structure. The problem with the CSR on accelerators is that the storage does not align well with GPU memory and access patterns. The SCS groups rows into chunks of size C and orders the memory vertically through chunks and horizontally across rows. The parameter C is set based on the SIMD width of the hardware being run on allowing the structure to map to the memory layout and align memory accesses with the hardware. Additional padding of empty cells is used to fill in the rows such that each row in a chunk has the same length and perfectly aligns with the hardware memory layout. A second parameter σ controls sorting the rows where sigma ranges from 1, no sorting, to the number of rows, full sorting. For non-uniform distributions, a higher σ will group longer rows which reduces the amount of padding and excess computations performed at the cost of running a sorting routine.

Figure 4 shows the storage of particles on a set of mesh elements in a CSR and two different SCS structures with $C = 4$ where the first uses no sorting ($\sigma = 1$) and the second has full sorting ($\sigma = 12$). Each row represents one mesh element with an entry in the row per particle within the mesh element. Arrows on the CSR and SCS structures show the continuous layout in memory. Empty cells in either SCS structures are padded cells with no particle data stored.

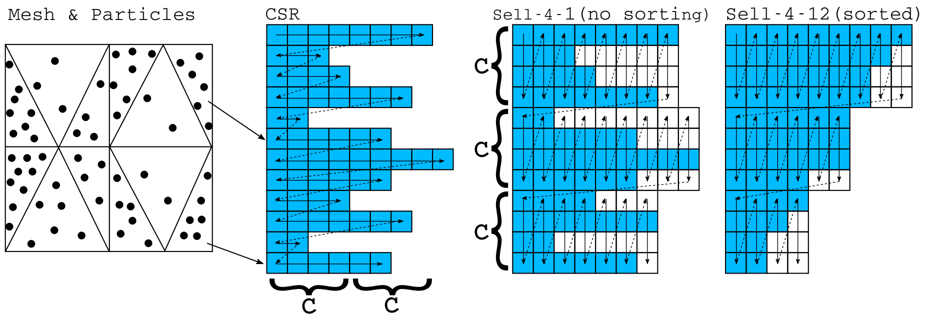

Figure 4: The storage of particles in a set of mesh elements (left) in a CSR (middle) and two SCS (right) with no sorting and with sorting. Arrows on each structure show the continous layout of memory.

Further improvements to the SCS structure are suggested by Besta et al. in their structure SlimSell [32]. For non-uniform distributions, there can be workload imbalances between large chunks and small chunks. Besta et al. add vertical slicing to the chunks so that approximately equal slices are distributed to blocks of threads as opposed to the uneven chunks. Figure 5 shows the SCS with vertical slices boxed as done in PUMIPic.

To store all data associated with particles, a separate SCS structure is constructed per requested type for the simulation. For example, take a particle being defined as a three-dimensional position, three-dimensional velocity, and an integer. The particle data structure would consist of three SCS structures. The first two would store 3 doubles per entry while the third would store a single integer per entry. This approach allows full customization of the data layout to control the data locality of particle information within the particle data structure. Each particle datatype can be separated into different SCS or combined into a single SCS by bundling the particle data into a single structure.

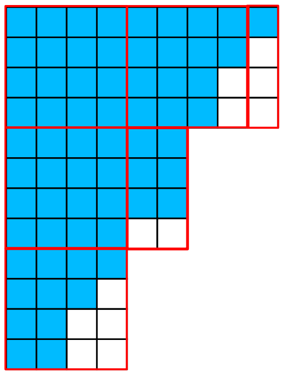

Figure 5: Sell-C-Sigma with full sorting and vertical slicing. Each box of entries represents a block of work given to a block of threads.

The SCS satisfies the requirement for a mesh-based particle data structure as the particles in an element are accessible from the equivalent row in the SCS. The tuning of σ and vertical slicing allow near-even distribution of workload to threads while the parameter C and padding allow tunable performance on different accelerator hardware in terms of memory layout and access pattern. These parameters also allow the structure to be adapted to the different simulation characteristics including particle distribution and density.

Algorithms to support PIC operations

Mesh Field Synchronization

To maintain dynamic mesh field information across processes, it is necessary to perform field synchronizations each iteration. This synchronization is performed after the charge deposition operation. Each process has contributions associated with the mesh entities that must be summed before the solve operation. Due to the additional buffering included in the PICparts, there is a significant amount of data that must be communicated across processes. When the PICparts are constructed with full buffering of the mesh, field synchronization can be performed using a single reduction across all ranks. The general approach for distributed mesh buffering is to use a fan-in fan-out communication protocol leveraging the full part buffer regions.

The fan-in fan-out algorithm is a three-stage approach shown in algorithm 2. The three steps on lines 1-5 are the fan-in, reduction, and fan-out. The fan-in phase sends contributions of each core region that makes up the PICpart to the owning process of the core region shown on lines 6-11. Since the mesh field is ordered by the core region and each core is fully buffered this is done by a bulk communication of a contiguous block of memory for each core region in the same order on the sending and receiving process. After the fan-in phase, the contributions of each copy of the field for the locally owned core region are reduced using an operation such as sum, max, or min on lines 12-17. As a result of using nonblocking communications, the reduction is done as each copy of the field is received. This allows overlapping the communication and computation as the field contributions can be transferred to the GPU and then reduced as each one is received. Then, the fan-out phase is performed on lines 18-23 by sending the reduced field of the core region to each of the PICparts that buffer it. Again taking advantage of bulk communication of contiguous memory.

Algorithm 2 pseudocode for fan-in fan-out algorithm for mesh field synchronization.

```
 1: function FieldSync($picpart, field, reduction\_op$)
 2:     FanIn($picpart, field$)
 3:     Reduction($picpart, field, reduction\_op$)
 4:     FanOut($picpart, field$)
 5: end function
 6: function FanIn($picpart, field$)
 7:     for all Core regions $p$ in $picpart$ do
 8:         Nonblocking Send $field[p]$ to $p$
 9:         Nonblocking Recv $field[self]$ from $p$
10:     end for
11: end function
12: function Reduction($picpart, field, reduction\_op$)
13:     for $i$ in 1 to number of core regions in $picpart$ do
14:         Wait for field $f$ from any picpart $p$
15:         Reduce $f$ into $field[self]$ using $reduction\_op$
16:     end for
17: end function
18: function FanOut($picpart, field$)
19:     for all Core regions $p$ in $picpart$ do
20:         Nonblocking Send $field[self]$ to $p$
21:         Nonblocking Recv $field[p]$ from $p$
22:     end for
23: end function
```

Particle Search

The push operation moves the particles to new positions. In addition to recording the new position, the parent element the particle is within after moving must be recorded before the next PIC operation. There are two approaches to locating the new parent elements of particles. The first is to construct a spatial data structure, such as a uniform grid or spatially-based tree decomposition, over the domain to quickly find the potential elements a particle may be within [33]. An alternative approach which we call adjacency search is to directly use mesh topology to iteratively step toward the new parent element starting at the old element. This is done by determining an exit face from the current element towards the particle's position using some set of testing criteria [34, 35]. Adjacency search is more performant than using a spatial data structure when particles move a short distance each iteration, where short distance means that the number of elements a particle traverses through a single push operation is at most a few. This aligns with the plasma physics simulations that PUMIPic targets wherein a given iteration most particles either do not leave the element they start in or move no more than two or three elements in any iteration. Additionally, no additional data structure needs to be allocated for adjacency search so there is both a memory and runtime improvement.

In PUMIPic, adjacency search is implemented using barycentric coordinates and ray tracing to determine the exit face. Since particles often remain within the element after a single push, the first step in determining the new parent element is to calculate its barycentric coordinates for the element it started within. If all the barycentric coordinates are valid the particle has remained within the same element. If any of the barycentric coordinates are invalid, the exit face is determined by the invalid barycentric coordinate plus ray tracing when needed to resolve edge cases. This method is iteratively applied until all particles have valid barycentric coordinates for the new element. Figure 6 shows an example particle and its path through elements using adjacency search. An additional result from using the adjacency search algorithm is that if a particle's path passes through a domain model boundary, the collision is recorded and passed back to the simulation to perform the specific wall interactions appropriate for the simulation.

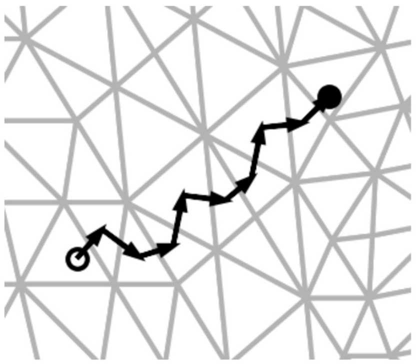

Figure 6: Path of a particle through a 2D triangular mesh using edge adjacencies.

In addition to tracking particles as they move through the mesh, most PIC applications will want to determine where particles hit the outer boundaries, and, when present, interior boundaries that demark interior components. Since the Omega_h [21] data structure maintains a complete mesh topological hierarchy of mesh regions, faces, edges, and vertices, as well as maintaining knowledge of when mesh faces, edges, and vertices are part of an exterior or interior boundary [36], it is easy to determine when particles hit these boundaries, thus allowing the ability to include physics associated with such interactions. In fact, in one PIC application, we are currently developing using PUMIPic, there are specific physics terms that must be accounted for even when particles are very close to boundaries. Since accounting for these terms is expensive and their contribution outside the region close to the boundary is negligible, procedures are being developed that employ the mesh topology information to activate those terms only when particles are approaching either an interior or exterior boundary.

Particle Structure Rebuild

After the particles are pushed and the new parent elements are determined using adjacency search, the particle structure must regroup the particles by the new parent elements. This regrouping, referred to from here on as rebuild, in general, is performed by fully reconstructing the particle data structure. The choice to fully reconstruct the SCS as opposed to supporting a growth mechanism is largely due to the parallelism supported on a GPU. Supporting a per element growing structure would result in a large amount of contention between the concurrent threads of a GPU. Reconstructing the data structure is done through data-parallel operations per particle along with efficient reductions that are better designed for running on GPUs.

The rebuild algorithm constructs a new particle data structure with the new distribution of particles. Then, the particle information is copied from the old structure to the new structure and finishes by destroying the old structure. Figure 7 shows an example of rebuild being performed. On the left is the SCS with filled cells and padded cells with element rows labeled on the side. Numbers inside the cells represent the element the particle is moving into, while no label means the particle did not leave its previous parent element. The right side of the figure shows the new SCS after rebuild. Note the element labels are in a different order now to maintain the sorting

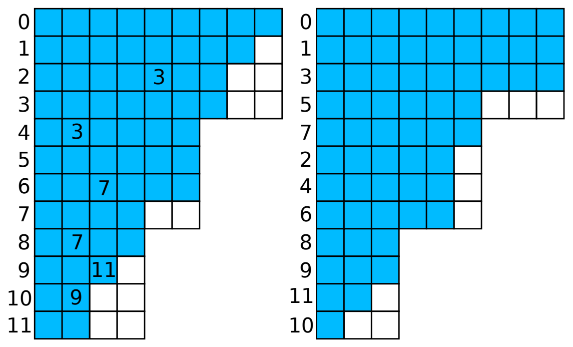

Figure 7: Example SCS before rebuild (left) with element numbering to the left. New parent elements of particles moving are listed within cells. The SCS after rebuild (right) with a new memory allocation along with resorting of the rows.

An alternative approach to rebuild is to utilize the padding of the SCS. Instead of copying the data from an old structure to a new structure, rebuild can be performed in-place by copying the particles moving to a new element into the padded cells of the element. The in-place rebuild can only be performed when each element receiving particles has at least as many padded cells as incoming particles. If this is not the case, then the full reconstruction rebuild is used. The occurrence of in-place rebuild can be increased by adding additional padding to each element when the SCS is constructed at the cost of more memory usage.

Figure 8 performs in-place rebuild on the same example from Figure 7. In this case, the right SCS uses the same memory as the original SCS after performing rebuild. Darker cells depict the new location of particles that have moved to a new element. Note that in-place rebuild does not perform sorting of the rows based on length. Along with the empty cells left throughout the structure, it is still ideal to perform a full rebuild after several iterations to improve the data layout.

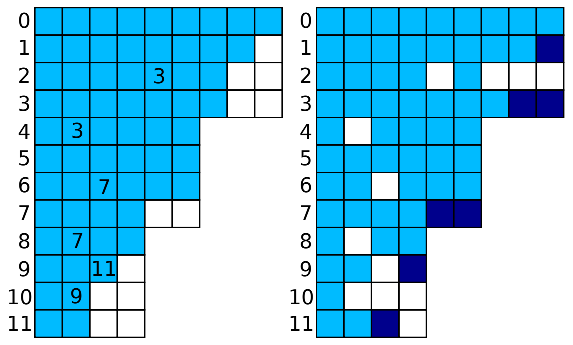

Figure 8: Example SCS before rebuild (left) with element numbering to the left. New parent elements of particles moving are listed within cells. The SCS after in-place rebuild (right) using the same memory as the initial SCS. Darker cells show the particles that moved to a new cell.

An additional capability of the rebuild step is to introduce new particles or remove particles from the system. For the full rebuild algorithm, new particles are added when creating the new particle structure and removed particles are ignored as if they were padded cells resulting in their omission from the new data structure. The in-place rebuild has to account for the new particles when checking if the in-place operation is possible. Particles leaving the system can act as padded cells allowing particles to replace those entries in the current rebuild.

Particle Migration

For particles that were pushed to an element outside the safe zone of the process, the particles must be migrated to a new process such that they are well within the safe zone of the new process. Due to the overlapping of PICparts, there are many processes that a particle can be migrated to. A straightforward strategy is to send migrating particles to the process which has the new parent element within its core. This guarantees the particles are always sent to a process that the element is within the safe zone. However, continuous execution of this strategy over several iterations will likely result in an imbalance of the particles across the processes. In this case, load balancing will be required to reassign particles across overlapping regions of safe zones across processes. An approach to this load balancing has been explored previously [37]. Implementing this routine efficiently for GPUs is a work in progress.

The particle migration routine is performed prior to the rebuild of the particle data structure such that the migrated particles can be added to the new process during the same rebuild. The migration routine communicates all particle data from the old process to the new process and is collected as new particles on the receiving process. Then the migrated particles are marked as left the simulation on the old process. The migrated particle information is passed onto the rebuild routine which will add the migrated particles as new particles to the particle data structure. On the old process, since the particles are marked as leaving the simulation, these particles are ignored when performing rebuild and will be deleted.

By default, the particle migration routine does not know which processes it will be communicating particles between. As such, the routine must perform an all-to-all communication to receive the counts of particles being sent and received on all processes. Alternatively, this all-to-all communication can be vastly reduced by using the partition of the mesh and PICparts. Particles can only be migrated between two processes that overlap PICparts. Since PICparts are a collection of cores, the process owners of each core that makes up the local PICpart is all that is needed for determining the process being sent to and received from.

Pseudo Plasma Simulation

We are currently working on the development of PUMIPic based implementations of two plasma physics codes, GITR and XGC. GITR [38, 39] is a plasma impurity transport code in which the PUMIPic implementation is using a fully 3D mesh to track the impurities as they move through the plasma and interact with the vessel wall. XGC [40, 41] is a 5D gyro-kinetic code specifically designed for the modeling of tokamak edge plasma physics. As outlined below XGC employs a specialized discretization of the tokamak geometry. The performance test performed here carry out core XGC PIC operations using the overall methods for data transfer and mesh-based operations used in XGC, but with simplified pseudo physics calculations. Using simplified pseudo physics allows us to focus on evaluating the performance of PUMIPic.

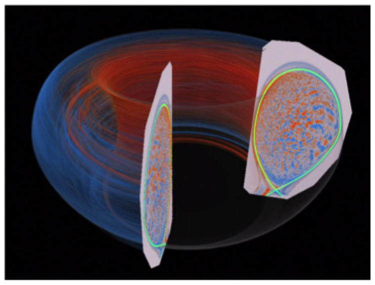

Figure 9: Toroidal representation of a tokamak with two poloidal planes.

In an XGC simulation, the tokamak domain is discretized into a set of cross sections perpendicular to the toroidal direction referred to as poloidal planes shown in Figure 9. A poloidal plane is represented by a two-dimensional unstructured mesh of triangles. The same unstructured mesh is used for each poloidal plane. The interior domain between adjacent poloidal planes can be considered as an extrusion of the triangles around the tokamak through which the particles traverse. Figure 10 shows an example model for a poloidal plane. Although the particles representing the ions and electrons are moving at a very high velocity, they are generally field following meaning that they will mostly stay near the same flux surface. XGC takes advantage of this by defining a nearly field-following mesh using a set of curves of constant magnetic flux in the definition of the domain (see Figure 10) and its mesh.

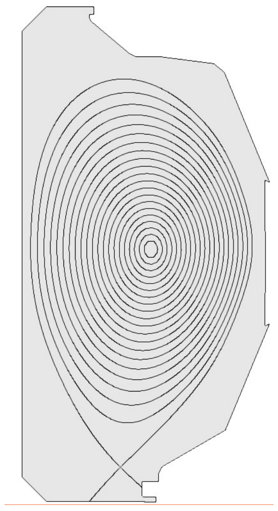

Figure 10: Model of a poloidal planes. Curves on the model represent flux curves of equal field values.

As noted in the Particle Migration section, in order to minimize the number of particles migrated, it is ideal to partition the mesh along with the motion of particles. Therefore, the core of each PICpart is defined by a number of adjacent flux faces. The buffer of each PICpart will be some number of flux faces around the core. The partition on the right in Figure 2 shows such a partition with one flux face per core.

The notion of poloidal planes adds additional partitioning of work necessary beyond the partition of the unstructured mesh. With simulations using 96 poloidal planes [42] and upwards of 128 or more poloidal planes [43], storing all necessary mesh fields for a given PICpart for all poloidal planes would be impossible in the GPU memory. Thus, partitioning the planes is required. In this study, the planes are partitioned such that each process stores the fields associated with two adjacent planes.

Particle Push

The goal of defining a push routine for the pseudo simulation is to approximate the general behavior of particles, without needing to define all of the physical routines required to accurately push particles. The desired behavior is that particles move few elements per iteration and generally move within flux faces. To achieve this, particles are pushed in an elliptical pattern around the center of the poloidal plane. The ellipse that each particle is pushed along is designed such that particles are roughly moving within a flux face. The distance each particle moves per iteration is controlled based on the distance from the center to ensure every particle moves only a short distance. In the toroidal direction, particles are pushed at a constant rate each iteration. This push routine accurately triggers the particle migration and rebuild of the particle data structure, but requires little computation per iteration and does not represent the computational cost of a real simulation's push.

Particle to Mesh

After the particle push, the charge deposition phase occurs where particles add contributions of charge onto the mesh fields. In a poloidal plane based simulation, this involves every particle depositing onto the mesh vertices of the two poloidal planes surrounding the particle. As a 5D gyro-kinetic code, XGC is only following the mean path of the particles. Since the fast gyro motion that is not tracked is not small with respect to the mesh size, using simple mean path location charge deposition on the mesh via simple interpolation is not sufficient since this would only deposit charge to three mesh vertices. Two methods can be used to approximate the gyro-motion, using gyroaveraging matrices [44] or defining a set of gyro rings for each vertex [45, 46]. In this study, we use the gyro ring method. The general idea is that each mesh vertex is surrounded by several gyro rings. Points along the rings are projected to the poloidal planes along field lines to distribute contributions on to the mesh vertices. Figure 11 depicts the steps of the approximation. First, a particle's position is projected to a virtual plane, V, that resides halfway between the two surrounding planes, P0 and P1. The mesh element on V containing the projected particle is found using adjacency search. Each vertex of the element distributes contributions on two gyro rings, g_0 and g_1, based on particle properties. Each point along the gyro rings are projected to planes P0 and P1. The mesh elements of these projected points are again found and contributions are divided between the vertices of these elements.

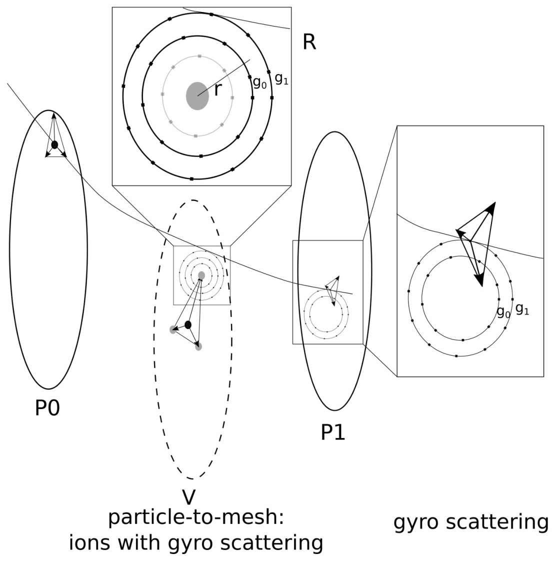

Figure 11: Depiction of charge deposition approximation for one particle. Particle on virtual plane V distributes contributions to gyro rings, g_0 and g_1, surrounding mesh vertices of element particle is within. Points along the gyro rings project contributions to poloidal planes P0 and P1 and deposit on mesh vertices.

Due to the number of projections and indirection of this approximation method, a mapping is constructed once at the beginning of the simulation from each mesh vertex to the projected mesh vertices on both surrounding planes. This reduces the operation to a projection to V followed by a lookup on the mapping based on particle properties.

Solve and Field to Particle

The pseudo simulation forgoes both the mesh field solve and field to particle phases. The field to particle phase follows similar calculations and data access patterns as the particle to mesh phase and as such were not implemented for the pseudo simulation. The field solve phase involves the solution of a Poisson equation [47] with extensive research in performing the field solve efficiently [33]. Implementing these operations is left for future work as part of the physically accurate simulation in development.

In place of these operations, the resulting field from charge deposition is accumulated across all processes such that each process has the same values for equivalent planes across all PICparts. This requires synchronization in the toroidal direction as well as utilizing PUMIPic's field synchronization across PICparts.

Performance Study

Scaling experiments are performed on the Summit supercomputer using a two-million triangle mesh executing 100 iterations of the particle loop. Scaling studies are performed using a constant number of particles per GPU as the number of nodes increases. Each node is configured to use one core per GPU for a total of six MPI ranks and six GPUs per node. Two sets of scaling studies are performed. First, the number of PICparts that the mesh is partitioned into is increased. This scaling has both weak scaling in terms of particles and strong scaling in terms of mesh entities per process. Then for a constant mesh partition, the number of poloidal planes is increased representing pure weak scaling in terms of particles.

Mesh Partition Scaling

The mesh partition scaling study is performed using one poloidal plane with six to 192 PICparts using BFS buffering. This study requires one Summit node for the lower case ranging up to 32 nodes for the 192 PICpart case. The pseudo simulation is executed with up to 48 million particles per GPU (mppg). Figure 12 presents the results from the simulation. The left plot shows the total time which is normalized by the single node experiment. Some increase in performance is gained by the partitioning of the mesh which is greater seen for lower particle counts. For four mppg there is up to a 55% decrease in time at 192 PICparts and a 7% decrease at 48 mppg. For larger particle counts, the particle operations dominate the computation and as a result, the scaling of the mesh has less of an effect on the total runtime. The scaling of the mesh also exhibits the expected diminishing performance gains. Around 90% of the reduction in runtime is achieved by 48 PICparts across the particle counts. For the given mesh entity count, partitioning the mesh further does not significantly decrease the PICpart size resulting in a majority of the part to be comprised of the buffer. This results in no additional performance gains to the mesh-based operations.

The right plot of Figure 12 shows the timing of the operations involved in each time step for the 48-mppg case. The six operations included in the figure are the particle push represented by the push line, the reconstruction of the particle structure labeled rebuild, the adjacency search labeled search, the particle to mesh or charge deposition phase labeled deposition, particle migration labeled migration, and the field synchronization step labeled sync. The predominately mesh-based operations are search and sync. As the number of PICparts increases, the cost for these operations decreases until flattening out. The rebuild operation is the dominant operation which is expected as it is the most data movement intensive operation. The remaining operations see no significant changes from scaling the mesh and the number of nodes used including particle migration.

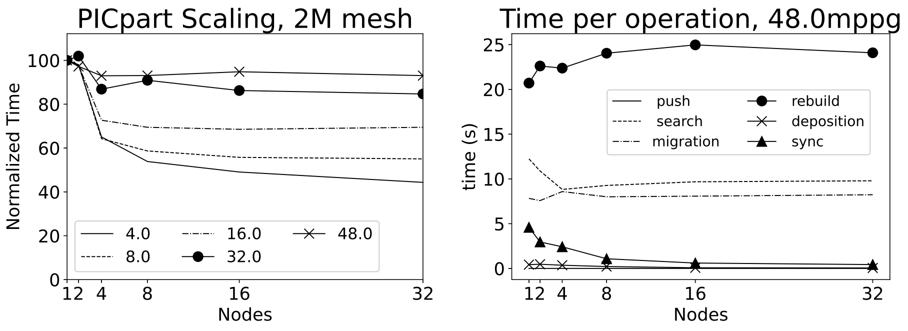

Figure 12: Simulation plots scaling PICparts from 6 to 192 with 2 to 48 mppg for 100 iterations of the particle loop. Normalized time (left) and breakdown on the time of the major operations (right)

Plane Scaling

The pure weak scaling study is performed by using the 192-PICpart partition with BFS buffering for the two-million element mesh and scaling the number of poloidal planes from one to 128. At 128 poloidal planes, the simulation uses 4096 of the 4608 nodes available on Summit. Results are presented with up to 48 mppg. Results are shown in Figure 13. Weak scaling efficiency is presented in the left plot. Scaling up to 256 nodes shows increased time by up to 30% at 4 mppg and 5% at 48 mppg. Scaling from 256 nodes up to 4096 nodes shows only fluctuations in total time with no major increase. The timing of the same major operations shown in the mesh scaling figures for 48 mppg is shown in the right plot. The increase in time scaling up to 256 nodes is caused by the migration and rebuild operations. All other operations show no major increase in time cost.

Closing Remarks and Future Work

The PUMIPic library was presented for developing distributed mesh-based particle in cell simulations. The library includes data structures for efficient storage of the mesh and particles and implementations of common operations required for PIC on GPUs. Scaling studies are performed up to 4096 nodes of the Summit supercomputer using a two million element mesh with up to 1.1 trillion particles. Scaling the mesh partition and weak scaling of the particles achieved between 7-55% decrease in time

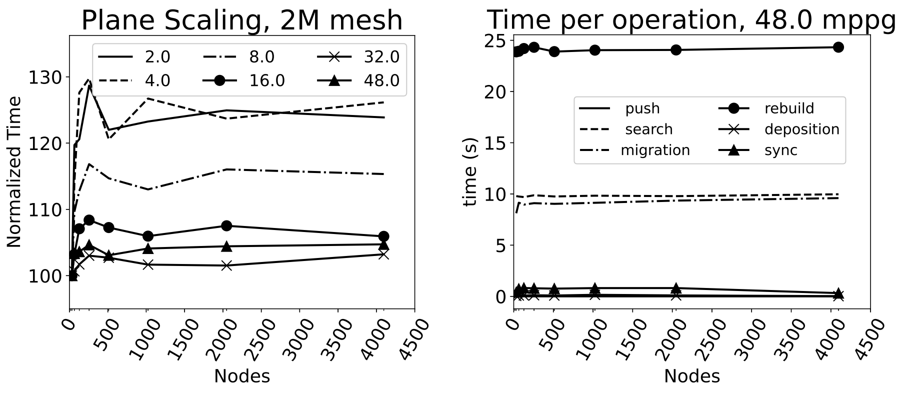

Figure 13: Simulation plots scaling planes from 1 to 128 with 2 to 48 mppg for 100 iterations of the particle loop. Normalized time (left) and breakdown on the time of the major operations (right)

with larger decreases in time seen at lower particles per GPU. Purely weak-scaling with a fixed mesh partition resulted in increases of time between 5-30% with less increase for larger counts of particles per GPU.

The performance studies show good weak scaling with respect to the number of particles when scaling both the number of PICparts and the number of planes that make up the computational problem. The next major step for the PUMIPic library is to implement a physically realistic simulation using the structures described here. Equivalent simulations for GITR [38, 39] and XGC [40, 41] using PUMIPic are ongoing work. Once simulations are implemented using PUMIPic, the performance can be compared between the traditional methods and PUMIPic to determine the effectiveness of our mesh-based approach to PIC simulations.

To further analyze the performance of the Sell-C-Sigma structure as a particle data structure, additional underlying data structures will be explored on GPUs. Current development is being done to implement the particle data structure in PUMIPic using the Cabana AoSoA [18]. Similarly, structures such as the CSR and SIMD width based structures will be implemented to compare with the Sell-C-Sigma. Given that the rebuild and migration operations are two of the biggest time consuming operations in the scaling studies, further structures with more efficient reconstruction of the data will be explored.

Another major time component from the studies is the adjacency search method. While our barycentric coordinate plus ray tracing is faster than using a spatial data structure and correctly manages edge cases, alternative approaches to adjacency search will be explored such as those presented by Chordá et al. [35] and Macpherson et al. [34].

In the pseudo physics simulation used in our scaling study particles remained well balanced due to the controlled motion of the particles. In a real physical simulation, it is expected that there will be an issue with particle load balance as the simulation evolves. As such, a dynamic particle load balancing strategy must be implemented to maintain even computation cost across the GPUs. Previously, a method for load balancing was explored on CPUs exploiting the overlapping regions of mesh to perform efficient diffusive balancing of particles. Implementing this method on GPUs is a work in progress.

Acknowledgements

This research was supported by the National Science Foundation under Grant No. ACI 1533581, (SI2-SSE: Fast Dynamic Load Balancing Tools for Extreme Scale Systems) and the U.S. Department of Energy, Office of Science, under awards DE-AC52-07NA27344 (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 National Science Foundation.

An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] E. DÁzevedo, S. Abbott, T. Koskela, P. Worley, S.-H. 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.

[2] R. Khaziev, D. Curreli, hPIC: A scalable electrostatic particle-in-cell for plasma–material interactions, Computer Physics Communications 229 (2018) 87–98. doi:https://doi.org/10.1016/j.cpc.2018.03.028.

[3] K. Madduri, K. Ibrahim, S. Williams, E.-J. Im, S. Ethier, J. Shalf, L. Oliker, Gyrokinetic toroidal simulations on leading multi-and manycore HPC systems, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2011, p. 23.

[4] B. Wang, S. Ethier, W. Tang, K. 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.

[5] W. Tang, Z. Lin, Global gyrokinetic particle-in-cel 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.

[6] K. Ko, A. Candel, L. Ge, A. Kabel, R. Lee, Z. Li, C. Ng, V. Rawat, G. Schussman, L. Xiao, et al., Advances in parallel electromagnetic codes for accelerator science and development, Tech. rep., SLAC National Accelerator Laboratory (SLAC) (2011).

[7] C. Birdsall, A. Langdon, Plasma physics via computer simulation, CRC press, 2004.

[8] J. Neudorfer, A. Stock, R. Schneider, S. Roller, C. Munz, Efficient parallelization of a three-dimensional high-order particle-in-cell method for the simulation of a 170 GHz gyrotron resonator, IEEE Transactions on Plasma Science 41 (1) (2013) 87–98.

[9] J. Qiang, X. Li, Particle-field decomposition and domain decomposition in parallel particle-in-cell beam dynamics simulation, Computer Physics Communications 181 (12) (2010) 2024–2034.

[10] H. Vincenti, M. Lobet, R. Lehe, L.-L. Vay, J. Deslippe, PIC codes on the road to exascale architectures, in: T. Straatsma, K. Antypas, T. Williams (Eds.), Exascale Scientific Applications: Scalability and Performance Portability, CRC Press, Taylor & Francis Group, 2017, pp. 375–407.

[11] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: Proc. Conf. High Performance Comput. Networking, Storage and Anal., 2009, pp. 1–11.

[12] R. Grimes, D. Kincaid, D. Young, ITPACK 2.0 User's Guide, CNA-150, Center for Numerical Anal., Univ. of Texas, Autin, Texas 78712 (1979).

[13] F. Vázquez, J. J. Fernández, E. M. Garzón, A new approach for sparse matrix vector product on NVIDIA GPUs, Concurrency and Computation: Practice and Experience 23 (8) (2011) 815–826. doi:10.1002/cpe.1658.

[14] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, A. R. Bishop, Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation, in: 2012 IEEE 26th Int. Parallel and Distributed Process. Symp. Workshops PhD Forum, 2012, pp. 1696–1702.

[15] 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.

[16] H. Burau, R. Widera, W. Hönig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T. E. Cowan, R. Sauerbrey, M. Bussmann, PIConGPU: A fully relativistic particle-in-cell code for a GPU cluster, IEEE Trans. Plasma Sci. 38 (10) (2010) 2831–2839.

[17] W. Hoenig, F. Schmitt, R. Widera, H. Burau, G. Juckeland, M. Müller, M. Bussmann, M. Muller, A generic approach for developing highly scalable particle-mesh codes for GPUs (01 2010).

[18] S. Slattery, C. Junghans, D. L-G, G. Chen, ascheinb, R. Bird, C. Smith, ECP-copa/cabana 0.1.0 (Feb. 2019). doi:10.5281/zenodo.2558369.
URL `https://doi.org/10.5281/zenodo.2558369`

[19] H. C. 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, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. doi:https://doi.org/10.1016/j.jpdc.2014.07.003.
URL `http://www.sciencedirect.com/science/article/pii/S0743731514001257`

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

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

[22] U. Çatalyürek, M. Deveci, K. Kaya, B. Uçar, UMPa: A multiobjective, multi-level partitioner for communication minimization, Contemporary Math. 588 (1) (2013) 53–64.

[23] G. Karypis, V. Kumar, Parallel multilevel series k-way partitioning scheme for irregular graphs, Siam Rev. 41 (2) (1999) 278–300.

[24] M. J. Berger, S. H. Bokhari, A partitioning strategy for nonuniform problems on multiprocessors, Comput., IEEE Trans. 100 (5) (1987) 570–580.

[25] J. Skilling, Programming the hilbert curve, in: 23rd Int. Workshop on Bayesian Inference and Maximum Entropy Methods in Sci. and Eng., Vol. 707, 2004, pp. 381–387. doi:http://dx.doi.org/10.1063/1.1751381.

[26] R. D. Williams, Performance of dynamic load balancing algorithms for unstructured mesh calculations, Concurrency: Pract. Exper. 3 (5) (1991) 457–481. doi:10.1002/cpe.4330030502.

[27] G. Cybenko, Dynamic load balancing for distributed memory multiprocessors, J. Parallel and Distributed Comput. 7 (2) (1989) 279–301.

[28] C. W. Smith, M. Rasquin, D. Ibanez, K. E. Jansen, M. S. Shephard, Improving unstructured mesh partitions for multiple criteria using mesh adjacencies, SIAM J. Scientific Comput. (2018) C47–C75doi:10.1137/15M1027814.

[29] M. Zhou, O. Sahni, K. Devine, M. Shephard, K. Jansen, Controlling unstructured mesh partitions for massively parallel simulations, SIAM J. Scientific Comput. 32 (6) (2010) 3201–3227.

[30] M. Mubarak, S. Seol, Q. Lu, M. S. Shephard, A parallel ghosting algorithm for the Flexible Distributed Mesh Database., Scientific Programming 21 (1) (2013) 17–42.

[31] S. Beamer, K. Asanovic, D. Patterson, Direction-optimizing breadth-first search, in: SC '12: Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal., 2012, pp. 1–10.

[32] 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.

[33] M. F. Adams, S.-H. 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.

[34] Macpherson, Graham B. and Nordin, Niklas and Weller, Henry G., Particle tracking in unstructured, arbitrary polyhedral meshes for use in CFD and molecular dynamics, Communications in Numerical Methods in Engineering 25 (3) (2009) 263–273. doi:https://doi.org/10.1002/cnm.1128.

[35] R. Chordá and J.A. Blasco and N. Fueyo, An efficient particle-locating algorithm for application in arbitrary 2D and 3D grids, International Journal of Multiphase Flow 28 (9) (2002) 1565–1580. doi:https://doi.org/10.1016/S0301-9322(02)00045-9.

[36] Beall, Mark W. and Shephard, Mark S., A General Topology-Based Mesh Data Structure, Int. J. Numerical Methods in Eng. 40 (9) (1997) 1573–1596. doi:10.1002/(SICI)1097-0207(19970515)40:9¡1573::AID-NME128¿3.0.CO;2-9.

[37] G. Diamond, C. W. Smith, E. Yoon, M. S. Shephard, Dynamic load balancing of plasma and flow simulations., in: Proceedings of the 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA '18, ACM, New York, NY, USA, 2018, pp. 73–80. doi:10.1109/ScalA.2018.00013.

[38] T. Younkin, D. Green, R. Doerner, D. Nishijima, J. Drobny, J. Canik, , B. Wirth, GITR simulation of helium exposed tungsten erosion and redistribution in PISCES-A, in: 59th Annual Meeting of the APS Division of Plasma Physics, 2017, p. UO4.002. doi:http://meetings.aps.org/link/BAPS.2017.DPP.UO4.2.

[39] T. Younkin, D. Green, A. B. Simpson, B. Wirth, GITR: A Global Scale Accelerated Particle Tracking Code For Wall Material Erosion and Redistribution In Fusion Relevant Plasma-Material InteractionsIn preparation.

[40] S. Ku, R. Hager, C. Chang, J. Kwon, S. Parker, A new hybrid-lagrangian numerical scheme for gyrokinetic simulation of tokamak edge plasma, J. Computational Physics 315 (2016) 467–475. doi:10.1016/j.jcp.2016.03.062.

[41] S. Ku, C. Chang, R. Hager, R. Churchill, G. Tynan, I. Cziegler, M. Greenwald, J. Hughes, S. E. Parker, M. Adams, et al., A fast low-to-high confinement mode bifurcation dynamics in the boundary-plasma gyrokinetic code XGC1, Physics of Plasmas 25 (5) (2018) 056107.

[42] K. Kim, J.-M. Kwon, C. S. Chang, J. Seo, S. Ku, W. Choe, Full-f XGC1 gyrokinetic study of improved ion energy confinement from impurity stabilization of ITG turbulence, Physics of Plasmas 24 (6) (2017) 062302. doi:10.1063/1.4984991.

[43] 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.

[44] J. Dominski, S.-H. Ku, C. Chang, Gyroaveraging operations using adaptive matrix operators, Physics of Plasmas 25 (02 2018). doi:10.1063/1.5026767.

[45] W. Lee, Gyrokinetic particle simulation model, Journal of Computational Physics 72 (1) (1987) 243 – 269. doi:https://doi.org/10.1016/0021-9991(87)90080-5. URL `http://www.sciencedirect.com/science/article/pii/0021999187900805`

[46] Z. Lin, W. W. Lee, Method for solving the gyrokinetic poisson equation in general geometry, Phys. Rev. E 52 (1995) 5646–5652. doi:10.1103/PhysRevE.52.5646. URL `https://link.aps.org/doi/10.1103/PhysRevE.52.5646`

[47] 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.