Towards Accelerated Unstructured Mesh Particle-in-Cell

1st Gerrett Diamond
SCOREC
Rensselaer Polytechnic Institute
Troy, NY
diamog@rpi.edu

2nd Cameron W. Smith
SCOREC
Rensselaer Polytechnic Institute
Troy, NY
smithc11@rpi.edu

3rd Chonglin Zhang
SCOREC
Rensselaer Polytechnic Institute
Troy, NY
zhangc20@rpi.edu

4th Eisung Yoon
Ulsan National Institute of Science and Technology
Ulsan, South Korea
esyoon@unist.ac.kr

5th Gopan Perumpilly
SCOREC
Rensselaer Polytechnic Institute
Troy, NY
gopan.p@gmail.com

6th Onkar Sahni
SCOREC
Rensselaer Polytechnic Institute
Troy, NY
sahni@rpi.edu

7th Mark S. Shephard
SCOREC
Rensselaer Polytechnic Institute
Troy, NY
shephard@rpi.edu

***Abstract*—The effective execution of unstructured mesh based particle-in-cell, PIC, simulations on GPUs requires careful design and implementation choices to ensure performance while maintaining productive programmability. This paper overviews the developing PUMIPic library that employs a set of mesh centric data structures and algorithms upon which unstructured mesh PIC simulation codes can be developed. Current performance results for component tests on up to 96 GPUs of the Summit system at Oak Ridge National Laboratory are presented.**

***Index Terms*—unstructured mesh, particle-in-cell, plasma physics**

I. Introduction

An important class of multiscale simulation is the Particle-in-Cell (PIC) method in which tracked particle motion is coupled to fields described in terms of partial differential equations that are discretized and solved for on meshes covering the domain of interest. The current and planned exascale computers are now providing the level of computational power required for the PIC method to be effectively applied to a wide range of scientific and engineering problems. For example PIC methods are being used in key simulations central to the design and planned operation of the ITER fusion tokamak. Due to their ability to deal with very general geometries and support general anisotropic mesh gradations, these simulation codes are increasingly employing unstructured mesh discretizations of the simulation domains. The price that has to be paid when using unstructured meshes is the need to employ more complex data structures and core mesh level operations than those that are required when structured meshes are employed. The need to attain performance on GPUs has only increased the complexity of developing performant unstructured mesh methods. The goal of developing the PUMIPic library presented in this paper is to provide the developers of PIC simulation codes that want to employ unstructured meshes with a set of GPU enabled, distributed mesh, data structures and unstructured mesh operations upon which performant PIC simulation codes can be developed.

Before presenting the PUMIPic library, section II briefly overviews the steps and operations common to PIC simulations emphasizing the mesh/particle interactions when unstructured meshes are used. To provide a better appreciation of the complexity of the operations and manipulations carried out, key mesh/particle interactions that are executed by the XGC [1], [2] fusion plasma gyrokinetics code are also outlined.

Section III introduces the key structures, operations and supporting procedures that constitute the PUMIPic library. The core data structure is the one that relates particles to the mesh elements. To meet the need for effective GPU execution the Sell-C-σ structure proposed by Hoefler et.al. [3] for matrix-based graph operations for efficient execution on GPUs is being used. The Omega library [4] from Sandia National Labs provides the unstructured mesh topology and field information on GPUs. A layer is built on top of Omega to better support the distribution of mesh entities for PIC simulations. Key operations using these structures are discussed including: adjacency search for locating the new element of particles after a push and field synchronizations for overlapping regions of mesh entities across processes. The status of the implementation of PUMIPic is presented in section IV while section V indicates the initial results for key operations executed on up to 96 GPUs

This research is supported by the National Science Foundation under Grant No. ACI1533581, and the U.S. Department of Energy, Office of Science, under awards DE-AC52-07-NA27344 (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.

of the Summit system at Oak Ridge National Laboratory.

II. Unstructured Mesh PIC Simulation

PIC codes perform a two scale information passing multiscale simulation in which particles capture the fine scale behavior and PDE defined over the domain of interest capture the coarse scale behavior. When the domains of interest are non trivial the coarse scale PDE fields are solved using a mesh based discretization technique. Given this, a most direct implementation of a PIC simulation consist of four operations that are executed at each time step of the simulation. The four operations are:

- Particle Push where the particle positions are advanced using information assigned to the particles based on the current domain mesh fields.
- Particle-to-Mesh where, based on the new positions of the particles, the mesh based fields driving the time dependent global fields are updated.
- Field Solve where the domain level PDEs are solved using a mesh based discretization to update the full set of global mesh based fields for the current time step.
- Mesh-to-Particle where the particle level information needed to push the particle for the next time step is mapped from the mesh to the particles.

The effective parallel implementation of a PIC calculation needs to define and maintain acceptable load balance while minimizing the impact of data access/motion and communications. This is a substantial challenge in a two scale information passing simulation where calculations are executed on two different scales that must share data between them.

In the common case where the push operation is collisionless (collisions in plasmas are often modeled by methods that avoid the need to intersect particle trajectories), each particle is advanced independently meaning that the push of particles is embarrassingly parallel. However, the particle to mesh operation requires interactions between the particle data and the mesh data which must be determined based on the relationship of the particle to mesh that has, in general, changed due to the particle's motion. The complexity of these interactions may be substantial and are such that the proper parallel implementation requires low level synchronization to ensure correct calculations. To exemplify the types of interactions that can occur, we consider the particle-to-mesh process for electrons and ions in the XGC PIC code.

In XGC particles exist inside a 3D tokamak, while the unstructured mesh is stored as a 2D cross section representing poloidal planes around the tokamak. The position of a particle is represented with ρ, z, ϕ where ρ and z define the in-plane coordinate on the mesh and ϕ is the angle around the tokamak. A particle's parent element is defined as the element which spatially contains the particle's planar position. The parent element is determined by projecting the particles position to a 'virtual' plane, V. The term 'virtual' is used as the particles position along ϕ is fixed, to support a static mapping to 'real' planes where the particles contribute to mesh field values. The hashed ellipse in Figure 1 depicts a virtual plane with

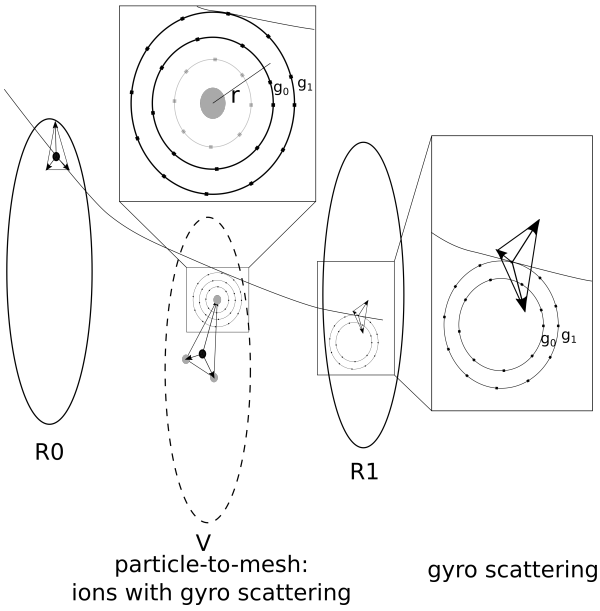

Fig. 1. Scatter (particle-to-mesh) transformation for ions. The black disc within the triangles is the particle being evaluated. The gray discs in the ion sub-figures are mesh vertices. Scatter points along the gyro rings (e.g., the rings labeled g_0 and g_1) are marked with small black dots.

a black disc marking the particle within the triangular parent element. To perform the subsequent scatter (particle-to-mesh) operation XGC must use field following projections to the polodial planes ahead, $R1$, and behind, $R0$, it. The projection is computed using the particle's ϕ coordinate and the static field-following background field. Note, the ϕ of the virtual plane bisects the bounding real planes $R0$ and $R1$. The curved line segment in Figure 1 represents the projection of a point in the virtual plane to a point in the $R0$ and $R1$ planes. The parent elements of the projected points are the 'real' elements $M_i^2 \sqsubset R0$ and $M_i^2 \sqsubset R1$.

The projection process for ions uses gyro scattering. Given the state of an ion, gyro scattering distributes ion information to multiple mesh vertices on $R0$ and $R1$ following a predefined pattern. For each mesh vertex the pattern is defined by a series of concentric rings centered on the vertex. Along each ring a number of points are defined; the black dots in the top-middle and right gyro ring and scattering sub-figures of Figure 1. Each of these point is projected to the $R0$ and $R1$ planes using the field following projection operation. Weighted field contributions from the ion are added to the bounding vertices of the 'real' parent elements containing these points. The vertices bounding the parent elements on $R0$ and $R1$ are stored in the gyro scattering pattern associated with the source vertex and its gyro ring number. For electrons, the projection does not use gyro scattering. Instead the vertices of the parent element of the electron are projected to $R0$ and $R1$ where weighted contributions are applied similarly as the ion gyro scatter.

Given the use of the 'virtual' plane fixed at $\phi = 0$, this vertex-to-vertex mapping is precomputed for both electron and

ion scatter. At runtime the ion state defines the gyro radius r at which information will be distributed. In Figure 1 the rings bounding r, depicted as g_0 and g_1, are selected to receive contributions from the particle marked with a black disc on the virtual plane.

Storage of the gyro ring pattern for each mesh vertex requires at most $2*3*G*M$ integers where G is the number of gyro rings, M is the number of points along each ring. The factor of three accounts for the three vertices bounding each element containing a point along each gyro ring while the factor of two accounts for the projection onto the $R0$ and $R1$ planes. If G and M are five and eight, respectively, then each vertex will need to store at most $2 * 3 * 5 * 8 = 240$ integers; 960B for four byte integers.

After all the particles are pushed and the mesh level fields have been updated based on the new particle positions, the updated mesh based fields must be evaluated. At this point this is a purely mesh-based operation that must also be executed in parallel to ensure scalability and effective parallel execution. Typically this operation is not the dominate step in a PIC simulation. For example, in the current XGC code this step consist of a set of poloidal plane implicit finite element solves that account for approximately 10 percent of the total time. In implementations where the mesh is distributed, an issue that does arise is that distribution of the mesh that is most effective to support the mesh-based PDE analysis is likely different from that which is optimal for the other three steps. It is clear the cost of alternating between two different distributions of the mesh for each time step would not be satisfactory and that one wants to employ a distribution that is most effective for the other three steps and make the mesh based solve as effective as it can be based on that mesh distribution.

The mesh to particle step for ions is the inverse of the gyro scatter step. For each mesh vertex bounding the parent element of an ion the gyro scatter map is used to lookup the mapped vertices and read their field values. The values from the parent vertices of the ions are then interpolated to the ions position within the element. Likewise, the electron mesh to particle step is the inverse of the particle to mesh step.

III. PUMIPIC

The goal of the PUMIPic library is to provide a set of data structures and services to support the development of unstructured mesh based PIC simulation codes. PUMIPic takes a mesh centric perspective of the data structures which is not what is commonly used in the development of PIC codes. More specifically, in PUMIPic access to the particles must be done through the mesh. The common approach taken in PIC codes is to employ independent particle and mesh data structures and to append information to the particle data structure indicating the mesh element each particle is currently associated with. Since the mesh data is substantially smaller than the particle data, most implementations assume that the core mesh data needed is small enough that a complete copy can be stored on each process. In addition, these implementations employ a spatial background grid, typically uniform, to support searches to determine the association of particles with elements once they have been pushed. At the basic level there are two concerns with the more standard approach. The first is that scalability with respect to the mesh is not possible since it is replicated and not distributed. The second is that if a PIC code wants to take advantage of graded anisotropic unstructured meshes, more complex, and more expensive, search structures than a uniform grid must be used. A less obvious concern for the standard approach is that as the simulation progresses the memory access patterns can degrade.

If one wants to have distributed data for both the mesh and particles it is clear that having independent structures for both will not be tractable since there will be no clear means to effectively control the expensive interprocess communications that are required. The approach taken in the development of PUMIPic is to have the mesh as the core distributed data structure and to relate the particles directly to the mesh elements. This provides a consistent means to deal with fully distributed PIC calculations and has specific data access pattern advantages. Of course since the particles move from element to element, there are complexities that must be addressed for an effective execution.

PUMIPic's core data structures for the unstructured mesh and particles are natively designed for performant execution on GPUs. An additional layer is built on top of the mesh data structure for the specialized multi-process parallel design for distributed mesh PIC in PUMIPic. PUMIPic also provides operations that are required for unstructured mesh PIC simulations. These operations include determining the new parent elements of particles after a push, restructuring the particle data structure with new parent elements, and synchronizing field information associated with mesh entities across the distributed mesh.

A. Mesh Data Structure

PUMIPic uses Omega [4] for maintaining the unstructured mesh and field information on GPUs. Omega defaults all memory and execution to the GPU ensuring minimum need to transfer between the host and device. Omega provides abstractions to common routines to perform mesh operations in a data parallel approach.

B. Particle Data Structure

Since particle operations tend to dominate the computation in PIC simulations, it is vital to have a particle data structure that is optimized to support efficient operations on GPUs. PUMIPic uses a structure called the Sell-C-σ (SCS) [3]. The SCS can be thought of as a rotated Compressed Sparse Row (CSR) data structure. Figure 2 shows an example of converting a matrix to CSR to SCS. The SCS groups rows into chunks of size C which is equal to the SIMD width of the hardware being run on. Each chunk is first ordered vertically then horizontally through the rows of the chunk. Padding is added to the ends of each row such that all rows in a chunk have the same length. Additionally a parameter σ controls sorting of the rows before chunks are setup in order to decrease padding and evenly

distribute the workload within a chunk. Figure 2 includes two SCS with no sorting and full sorting. The SCS with full sorting has significantly less padding than no sorting: 12 to 32.

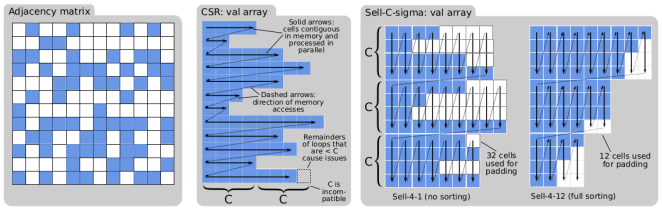

Fig. 2. Conversion of a matrix (left) to Compressed Sparse Row (middle) to Sell-C-σ (right) with no sorting and full sorting [5].

The ordering of chunks and sizing to the SIMD width allows the structure to map directly to the access patterns of the hardware. Each chunk can be distributed to a team of threads for efficient operation and memory access. However, highly irregular data will result in an imbalance of work being sent to each team. Besta et al. suggest using vertical slicing of chunks to create approximately equal slices for operations [5]. Vertical slicing leads to more slices to operate on allowing more concurrency and a better balance of work per team.

For usage as a particle data structure, the SCS has a row for each element in the mesh. Cells are then added for each particle in the element. To store all of the data associated with a particle, an SCS is constructed for each piece of information for a particle that are maintained throughout the simulation. In order to support PIC simulations, the SCS must support movement of particles between elements, addition and removal of particles, and migration of particles between processes.

C. Mesh Partitioning for Particle Push

For distributed-mesh PIC, the unstructured mesh is partitioned across processes where each process is assigned ownership of a collection of elements. The set of mesh elements owned by the process and their closure make up a core 'part'. Additionally, each process must maintain sufficient neighboring mesh information to perform particle calculations and particle-mesh interactions without communications. The core part and neighboring information constitute a PICpart. Figure 3 shows a PICpart constructed with a core region plus several layers of buffered elements.

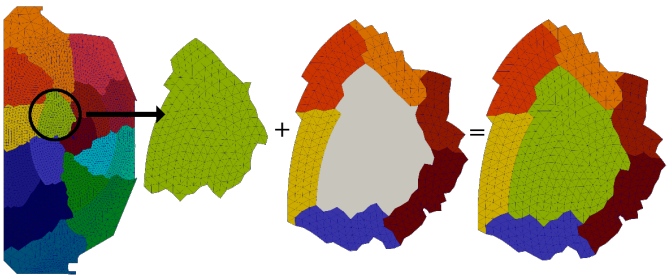

Fig. 3. From left to right: Partitioned mesh, core part, buffer around core part, PICpart

In a past version of PUMIPic, buffering was done using layers of elements off the part boundary as shown in Figure 3. This approach had memory problems when the overlap of buffers was large due to the mesh data structure maintaining remote copy information for each copy of a given mesh entity. In the new version of PUMIPic, full parts are buffered around the boundary instead of layers of elements as seen in Figure 5.

Constructing PICparts with additional elements beyond those that satisfy the field dependency of particles comes with an increased memory cost. Offsetting this cost is an increase in mesh field synchronization performance via bulk communications. On each part a global entity numbering enables single-instruction multi-thread (SIMT) parallel array operations for sending, receiving, and updating the large arrays of part field data.

As particles move around a PICpart, some will move too close to the edge of the buffer and will not be able to perform all mesh-particle interactions or be in danger of being pushed outside the domain of the PICpart. Any element that cannot guarantee a particle will remain on process after a push or cannot perform an interaction is referred to as unsafe. If a particle moves to an unsafe element then the particle must be moved to a PICpart where its element is in the safe zone. Determining unsafe elements prior to running the simulation is not straight forward and is different per simulation. Instead, PUMIPic labels the elements of the owned part and a small portion of the buffer elements as safe and migrates the particles when they leave the safe elements. Assuming a conservative strategy is taken to assign safe elements then particles will be migrated before entering the unsafe region of the PICpart.

Continuous migration of particles due to leaving the safe zone of a PICpart can lead to an imbalance of particles. Since the PICparts have overlapping regions of buffered elements, there is the capability to employ dynamic load balancing to maintain a better balance of particles by exchanging particles within the buffered regions. Diamond et.al. [6] discuss an approach to perform dynamic load balancing on distributed mesh PIC with consideration of the safe zone.

D. Adjacency Search

After each particle push operation, particles are displaced to a new position. As such, some particles will be pushed to a new element. In order to find the new element the particles belongs to, an adjacency walk procedure is used called adjacency search. This operation starts at the initial element for each particle and iterates to adjacent elements until the new element is found. Figure 4 depicts the path of a particle through a 2D mesh using edge adjacencies.

E. SCS Rebuild/Migration

Once the new parent element of each particle is computed, the SCS structure must be reconstructed to properly account for the changes. First particles that have moved outside the safe elements of a PICpart must be migrated to a PICpart where they will be in a safe element. Once all particles on a process are within safe elements of the PICparts the SCS is

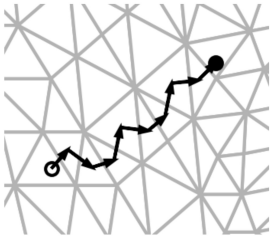

Fig. 4. Path of a particle through a 2D triangular mesh using edge adjacencies.

rebuilt by moving particles to their new parent element and reapplying sorting and padding to fill the structure's format.

F. Field Synchronization

A common mesh operation on a distributed mesh is field synchronization. This operation takes values of a field across multiple processes and combines the values to get the full value of the field across all mesh entities. Since PUMIPic buffers entire parts to make up PICparts, field synchronizations can be done by exchanging the large field data arrays associated with each part. Additional care must be taken at the boundary of PICparts, as lower dimension entities exist on PICparts that do not have a full copy of the parts.

IV. Implementation

PUMIPic [7], Parallel Unstructured Mesh Infrastructure for Particle-in-cell, is an open source library being designed for performance running on supercomputers with GPUs. For performance portability, Kokkos [8] is used to abstract memory access and parallel execution on GPUs. Implementation details for the particle data structure, PICparts, and the operations in PUMIPic are discussed below.

A. Particle Data Structure

The implementation of the SCS for PUMIPic's particle structure is designed to support different applications' definition of a particle. As such, the storage for each particle consists of an application defined set of statically sized datatypes. The particle structure maintains an SCS for each data type that uses the same indexing scheme.

In order to hide the complexity of the SCS and its indexing scheme, an abstraction of the parallel loops is provided for users. Algorithm 1 shows the necessary code to perform a loop over elements/particles in the SCS. Line 1 declares a lambda to perform some operation. Line 2 uses a mask to ignore the padded entries of the SCS. Line 6 runs the lambda over the SCS on the GPU.

Algorithm 1 Example pseudocode to loop over SCS particles

```
1: lambda = LAMBDA(element_id, particle_id, mask) {
2: if mask is true then
3:     Perform operation on particle
4: end if
5: }
6: scs.parallel_for(lambda);
```

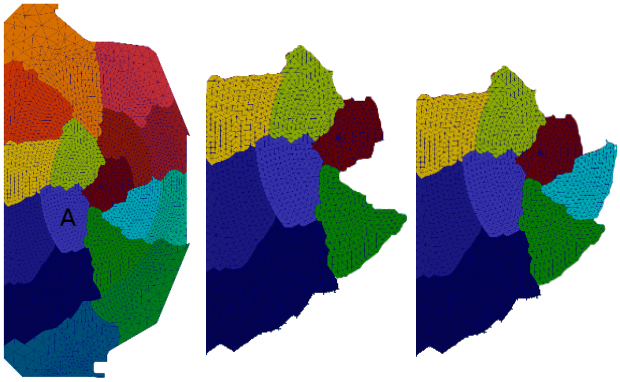

Fig. 5. A partitioned mesh (left), the PICpart for part A with minimum buffering (middle), and the PICpart for part A with 6 layers of BFS buffering (right)

B. PICparts

PUMIPic supports different methods to construct the buffering for PICparts and designating the safe elements. Given the initial partition of the mesh, the buffering is constructed from one of three methods: full, minimum, or breadth-first search (BFS). The full case buffers the entire mesh for each PICpart. This is equivalent to traditional PIC where each process maintains a copy of the entire mesh. Full buffering is used when the memory cost of the mesh is not an issue. Minimum buffers only the parts adjacent to the boundary of the owned part. This method can be too conservative depending on the mesh partition. The BFS method is a safer method that buffers parts within a number of layers of a topological breadth-first search. This approach captures parts that are close to the boundary of the owned part but not directly adjacent. Figure 5 shows a partition of a mesh and PICparts for one part using minimum and BFS buffering. With minimum buffering the PICpart lacks mesh entities near the part boundary that are required in some PIC simulations.

The safe zone can be constructed with the same three methods as the buffering. Full safe zone is only possible to use when the buffer is also full. Minimum designates only the owned part as safe which is too conservative and will lead to more particle migrations. The BFS sets all elements within a breadth-first traversal from the owned part to be safe and is the best choice when the mesh is distributed.

C. Adjacency Search

Given a particle within an element, its current position, and a destination position computed by the push operation, a walk to adjacent elements via M^{d-1} mesh entities (edges bounding faces in 2D or faces bounding elements in 3D) is performed. The walk is executed on the GPU by running a sequence of parallel SCS lambdas over the particles within a loop that terminates once all particles have reached the element containing their destination position. The first parallel lambda computes the barycentric coordinates (u, v, w for triangles) for the destination position and the current element. If the position is contained within the element, $[u, v, w] >= 0$,

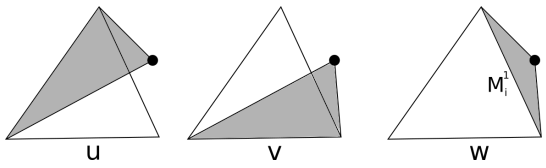

Fig. 6. A triangle and a point outside it marked with a black dot. The three barycentric coordinates u, v, w are depicted as the area of the sub-triangles formed by an edge and the point. Coordinate w has the minimum value and is associated with mesh edge M_i^d.

then the particle is set as 'done'. Otherwise, the mesh entity M^{d-1} associated with the minimum coordinate is marked as the next entity to traverse. For example, Figure 6 depicts the minimum coordinate w associated with mesh edge M_i^d for the point (black dot) outside the triangle. Next, a lambda checks if the marked entities are classified on the geometric model boundary. If so, application specific particle-wall interactions are computed. Next, remaining particles set their next element to be the upward adjacent element of the marked entity that is not the current element. An iteration of the while loop is completed by running a parallel reduction over the particles to see if any particles are not 'done'. If particles remain, loop execution continues.

D. SCS Rebuild and Migration

The rebuild routine combines two updates to the particle structure on process. Primarily, particles are regrouped by their new parent elements. Additionally new particles can be added to the system. Rebuild involves constructing a new SCS with different indexing and padding. To avoid allocating the data every rebuild, two copies of the SCS are maintained and each rebuild fills the inactive SCS and swaps the two structures. This results in roughly two times the memory being used throughout the simulation. Reallocation of the arrays only happens if the rebuild requires a larger structure either because new particles have been added or more padding was required.

Particle migration is built on top of the rebuild routine. First particle data is communicated to the new process it belongs to. The particles migrated away are treated as leaving the system and particles that are received are supplied to the rebuild routine as new particles being added to the SCS.

E. Field Synchronization

Field Synchronization in PUMIPic is performed through bulk communications using a fan-in, fan-out approach. In the first step, field information on buffered copies of a part are sent to the owner of each part. Then a reduction is performed on each part. Followed by the fan-out phase sending the result from the owned part to the non-owned copies. Since the mesh is consistently ordered across all PICparts, the fields can be communicated and operated on as a contiguous block with SIMT parallel array operations. Note, when every PICpart contains the entire domain this operation is optimized by performing an MPI_Allreduce on the entire field across all processes.

For the lower dimension mesh entities (mesh vertices, edges and faces in 3D) on PICpart boundaries, a halo-exchange is performed to collect contributions from boundaries to the owning part. The reduction is applied to these entities along with the field information gathered in the bulk communications. Then updated field values are returned to the buffered entities. The halo exchange utilizes a static mapping to account for the different ordering of the boundary entities on the PICparts in which the entities exist.

MPI Communications for particle migration and field synchronization are performed by copying GPU memory to host memory and then communicating between CPUs. Usage of CUDA-aware MPI where no explicit host-to-device copy is required is currently being tested for particle migration, but is not currently in use.

V. Initial Tests

Initial testing of the key operations described in Section IV was performed on the IBM AC922 Summit system at Oak Ridge National Laboratory [9]. In all tests one MPI process is used per GPU and all six GPUs on each node are used. Each process has a full copy of the mesh, as done in XGC, with a safe zone defined based on the BFS method described in **??**. The non-full safe zone enables testing of the particle migration and synchronization operations. Particles are uniformly distributed within the mesh elements classified on geometric model faces defined by closed magnetic flux curves shown in Figure 7. No particles are created outside the last closed curve. Tests execute 100 iterations of particle push, adjacency search, particle structure rebuild, particle migration, gyro scattering particle to mesh, and mesh field synchronization on the 24 thousand triangle mesh [10] depicted in Figure 7. The particle push operation used in these initial tests is a non-physical proxy for the XGC push that moves each particle along an elliptical path centered near the central mesh vertex. Likewise, the gyro scattering procedure does not project the gyro ring to the forward and back planes following the background mesh field as done in XGC; the gyro ring is centered on the mid-plane mesh vertex on the forward and back plane. This simplification captures the highly irregular data access pattern that is the key performance limiter for the particle to mesh operation.

Figure 8 plots the rate at which iterations are executed in terms of millions of particles per second. Figure 9 plots the weak scaling efficiency. For both plots higher values indicate better performance. As the number of particles per GPU (PPG) increases the rate at which push iterations are completed increases. Relative to six GPUs, the 36 GPU push iteration rate is 5.2 times higher for two million PPG, and 4.2 times higher for 48 million PPG. This trend is reflected in the weak scaling plot where two million PPG has an efficiency of 87% on 36 GPUs while 48 million PPG is only 70%.

Figure 10 depicts the timing of each operation on one GPU to 36 GPUs with 48 million PPG. Note, each depicted operation is preceded with an MPI_Barrier to isolate it from imbalances in the prior operations. Figures 8 and 9 were

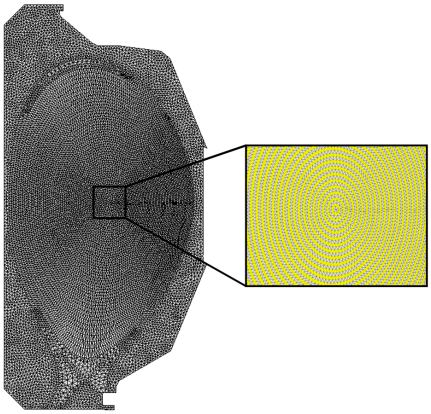

Fig. 7. Simmetrix [11] generated XGC mesh with 24 thousand triangles and 58 geometric model faces defined by magnetic field flux curves.

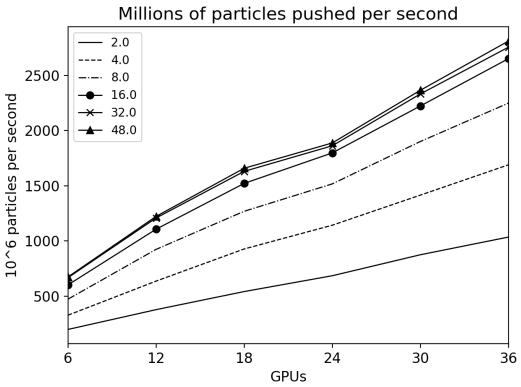

Fig. 8. Millions of particles pushed per second with 2 million to 48 million particles per GPU on up to 36 GPUs.

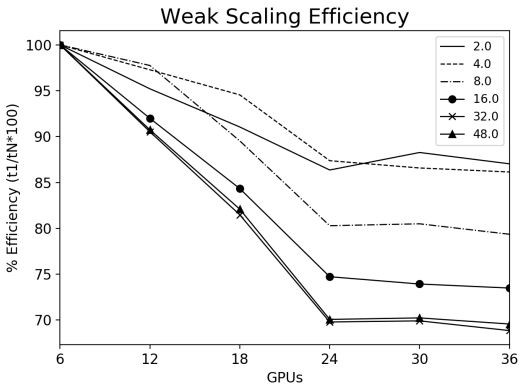

Fig. 9. Weak scaling efficiency of 100 push iterations with 2 million to 48 million particles per GPU on up to 36 GPUs.

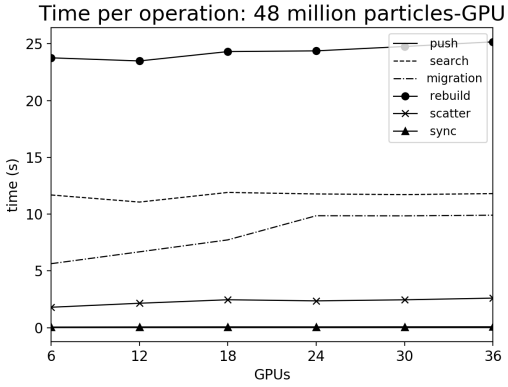

Fig. 10. Time spent in each push operation with 48 million particles per GPU. Each operation is preceded with an MPI_Barrier to isolate the operation from imbalance.

created from runs without barriers. A partial cause of weak scaling loss is the increased time spent in the particle migration operation (labelled 'migration'). From 6 GPUs to 24 GPUs the migration cost increases from 5.6 seconds to 9.9 seconds and remains nearly constant from 24 to 36 GPUs. More studies are needed to determine the source of this behavior.

An additional test with 1.4 billion particles was performed on 96 GPUs; approximately 15 million PPG. The test was on a mesh with 126k triangles and 99 geometric model faces formed by closed flux curves. Initial particle distribution and PICPart creation follow the approach of the previously described test; a full mesh copy on each process, a BFS defined safe zone, and uniform particle distribution within the last closed flux curve. Weak scaling on 96 GPUs, from 36 GPUs, is 86% and has a push rate of 4,372 million particles pushed per second, versus 1,867 on 36 GPUs. The times of each operation are listed in Table I. Performance of push, search, rebuild are at most 5% slower in the 96 GPU run. A minimal change in the run time of these operations is expected as they require no inter-process (MPI) communications. Scaling is mostly lost in migration where an increase of 18% is observed. Field synchronization cost increases by 35% in the 96 GPU run, but relative to other operations, it takes an insignificant amount of time. Understanding the source of the migration cost increase (system network noise, mesh distribution and the induced communication graph, particle load imbalance, etc.) is the focus of ongoing testing efforts.

	GPUs	
time (sec.)		
operation	36	96
push	0.0063	0.0054
search	5.4	5.7
rebuild	7.1	7.3
migration	2.7	3.2
scatter	0.95	0.98
sync	0.17	0.23

TABLE I

TIME SPENT IN EACH OPERATION FOR A 36 AND 96 GPU RUN WITH 15 AND 16 MILLION PPG, RESPECTIVELY. EACH OPERATION IS PRECEDED WITH AN MPI_BARRIER TO ISOLATE THE OPERATION FROM IMBALANCE.

VI. Closing Remarks and Future Work

Current progress for the PUMIPic library was presented for distributed-mesh PIC simulations. The library provides structures and operations for performance and usability on GPUs. Weak scaling studies showed 70%-86% efficiency for the experiments up to 48 million particles per GPU on up to 36 GPUs. Weak scaling degradation is a result of particle migration which includes the majority of communications in the simulation. Performance is also being restrained by the rebuild operation on the particle structure due to large amounts of data movement required. An additional experiment with a larger mesh and 1.4 billion particles was ran on 96 and 36 GPUs. Initial results again indicate that a significant source of scaling loss is in particle migration. Ongoing testing efforts are underway to quantify these losses and sources of load imbalance.

Moving forward additional attention will be focused on improving the performance of the key operations slowing down simulations and reducing weak scaling efficiency. Primarily this is the SCS rebuild operation. Rebuild requires optimizations to avoid memory allocation and minimize data movement. Foreseeable improvements to rebuild will result from taking advantage of the fact that for a given push most particles do not leave the starting element. With this in mind, a more efficient shuffling procedure can be employed to minimize the movement of data. Further improvements are expected from finer optimization of operations and the implementation of load balancing for the new PUMIPic library.

While some operations do not achieve the expected performance for the proxy simulations, the PUMIPic library includes the necessary steps to begin building real physical PIC simulations. When a significant portion of the physics for these simulations have been implemented using PUMIPic, additional comparisons will be analyzed against the existing CPU and GPU implementations to get a more detailed result for PUMIPic.

VII. Acknowledgments

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] S. Ku, R. Hager, C. Chang, J. Kwon, and S. Parker, "A new hybrid-lagrangian numerical scheme for gyrokinetic simulation of tokamak edge plasma," *J. Computational Physics*, vol. 315, pp. 467–475, 2016.

[2] M. F. Adams, S. H. Ku, P. Worley, E. D'Azevedo, J. C. Cummings, and C. Chang, "Scaling to 150k cores: Recent algorithm and performance engineering developments enabling xgc1 to run at scale," *J. Physics: Conf. Series*, vol. 180, no. 1, 2009.

[3] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and 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*, vol. 36, no. 5, pp. C401–C423, 2014.

[4] D. Ibanez. (2016) Omega_h GitHub repository. [Online]. Available: https://github.com/ibaned/omega_h

[5] M. Besta, F. Marending, E. Solomonik, and 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, May 2017.

[6] G. Diamond, C. W. Smith, E. Yoon, and 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*, ser. ScalA '18. New York, NY, USA: ACM, Nov. 2018, pp. 73–80.

[7] C. Smith and G. Diamond. (2019) PUMIPic github repo. [Online]. Available: http://github.com/SCOREC/pumi-pic

[8] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *2013 Extreme Scaling Workshop*, Aug. 2013, pp. 18–24.

[9] "IBM Power System AC922: Technical Overview and Introduction," IBM, USA, Tech. Rep. REDP-5494-00, 2018.

[10] F. Zhang, R. Hager, S.-H. Ku, C.-S. Chang, S. C. Jardin, N. M. Ferraro, E. S. Seol, E. Yoon, and M. S. Shephard, "Mesh generation for confined fusion plasma simulation," *Engineering with Computers*, vol. 32, no. 2, pp. 285–293, apr 2016.

[11] Simmetrix. (1997) Simmetrix: Enabling simulation-based design. [Online]. Available: http://www.simmetrix.com/

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: TOWARD ACCELERATED UNSTRUCTURED MESH PARTICLE-IN-CELL

A. Abstract

Information is provided to execute the test cases described in Section V of the ScalA19 workshop paper titled "Toward Accelerated Unstructured Mesh Particle-in-Cell".

B. Description

1) Check-list:

- **Algorithm:** Unstructured mesh particle-in-cell
- **Program:** psedoPXGCm
- **Compilation:** GNU GCC
- **Transformations:**
- **Data set:** github.com/SCOREC/pumipic-data
- **Hardware:** OLCF Summit: IBM AC922
- **Output:** github.com/SCOREC/pumipic-docs/tree/scala19
- **Publicly available?:** Yes

2) How software can be obtained: The specific versions (via Git SHA1 hash) of the libraries, along with the name of the GitHub repo (prefix with https://github.com to open in a web browser), used to execute the tests are listed below.

- PUMIPic:
  SCOREC/pumi-pic 6ea3f9c
- particle_structures:
  SCOREC/particle_structures db90ede
- Omega:
  SNLComputation/omega_h 66209e0
- Kokkos:
  kokkos/kokkos 2983b80

3) Hardware dependencies: The experiments were performed on the Summit IBM AC922 system at Oak Ridge National Laboratory.

4) Software dependencies: PUMIPic depends on the following libraries listed above in addition to CUDA 10 and an MPI-3 implementation. The Summit runs used CUDA 10.1.168 and Spectrum MPI 10.3.0.1-20190611.

5) Datasets: The input meshes and partition files are located in the GitHub repo: github.com/SCOREC/pumipic-docs/tree/scala19

C. Installation

For Summit we load the following modules:

```
module swap xl gcc/7.4.0
module load cuda/10.1.168 cmake
```

and the following modules were provided by default:

```
hsi/5.0.2.p5
xalt/1.1.3
lsf-tools/2.0
darshan-runtime/3.1.7
DefApps
spectrum-mpi/10.3.0.1-20190611
```

To build Kokkos:

```
mkdir build
cd build

cmake .. \
  -DCMAKE_CXX_COMPILER=/path/to/kokkos/bin/nvcc_wrapper \
  -DKOKKOS_ARCH=Volta70 \
  -DKOKKOS_ENABLE_SERIAL=ON \
  -DKOKKOS_ENABLE_CUDA=ON \
  -DKOKKOS_ENABLE_CUDA_LAMBDA=ON \
  -DKOKKOS_ENABLE_DEBUG=OFF \
  -DKOKKOS_ENABLE_PROFILING=ON \
  -DCMAKE_INSTALL_PREFIX=$PWD/install

make install
```

To build Omega_h:

```
kk=/path/to/kokkos/install
kksrc=/path/to/kokkos/
export CMAKE_PREFIX_PATH=$kk:$CMAKE_PREFIX_PATH
export OMPI_CXX=$kksrc/bin/nvcc_wrapper #spectrum is based on openmpi
export PATH=$CUDA_DIR/bin:$PATH
export LD_LIBRARY_PATH=$CUDA_DIR/lib64:$LD_LIBRARY_PATH

mkdir build
cd build

cmake .. \
  -DCMAKE_INSTALL_PREFIX=$PWD/install \
  -DBUILD_SHARED_LIBS=OFF \
  -DOmega_h_USE_CUDA=ON \
  -DOmega_h_USE_MPI=ON \
  -DCMAKE_CXX_COMPILER=mpiCC \
  -DOmega_h_CXX_WARNINGS="OFF" \
  -DOmega_h_USE_Kokkos=ON \
  -DKokkos_PREFIX=$kk/lib/CMake

make install
```

To build particle_structures:

```
omega_h=/path/to/omega_h/install/
export CMAKE_PREFIX_PATH=$omega_h:$CMAKE_PREFIX_PATH

mkdir build
cd build

cmake .. \
  -DCMAKE_CXX_COMPILER=mpiCC \
  -DENABLE_KOKKOS=ON \
  -DPS_ENABLE_DEBUG_SYMBOLS=ON \
  -DPS_ENABLE_OPT=ON \
  -DCMAKE_INSTALL_PREFIX=$PWD/install

make install
```

To build PUMIPic:

```
ps=/path/to/particle_structures/install/
export CMAKE_PREFIX_PATH=$ps:$CMAKE_PREFIX_PATH

mkdir build
cd build

cmake .. \
  -DCMAKE_CXX_COMPILER=mpiCC \
  -DPP_ENABLE_DEBUG_SYMBOLS=ON \
  -DPP_ENABLE_OPT=ON \
  -DIS_TESTING=ON \
  -DTEST_DATA_DIR=$PWD/../pumipic-data

make
```

D. Experiment workflow

The job submission scripts used to run the test cases are located in the github.com/SCOREC/pumipic-docs/tree/scala19 repo on GitHub in the scala19/results sub-directory.

To submit the sweep from 6 to 36 GPUs for 2 to 48 million particles per GPU run the following command:

```
#edit the project id in submit.sh
cp /path/to/scala19/results/submitSweep.sh .
cp
/path/to/scala19/results/401311_itg24k_mfull_sbfs_i100_n1-6/*_n6t36/subm
cp /path/to/scala19/results/401311_itg24k_mfull_sbfs_i100_n1-6/*_n6t36/r
#edit the paths in runPXGCmSweep.sh for the input mesh and partition fil
./submitSweep.sh
```

Likewise, to run the 96 GPU case with 16 million particles per GPU run the following commands:

```
cp /path/to/scala19/results/501211_itg126k_mfull_sbfs_i100_n16/*_n16t96/runPXGCmSweep.sh .
#edit the paths in runPXGCmSweep.sh for the input mesh and partition files
#uncomment the indicated lines in submitSweep.sh for running the 96 GPU case
./submitSweep.sh
```

E. Evaluation and expected result

Expected results are located in the github.com/SCOREC/pumipic-docs/tree/scala19 repo.

Timing plots of are produced by running the Python pandas/numpy/matplotlib script `plot.py` in the `results/*itg*` directories.

Note, there is a unresolved race condition in the code that prevents some executions of the 96 GPU case from completing successfully.