# An Introduction to the PUMI libraries

PUMI Developers

January 3, 2019

**Abstract**

The Parallel Unstructured Mesh Infrastructure (PUMI) is a set of C and C++ libraries that provide functions for querying and modifying parallel meshes, and querying the geometric models they discretize. PUMI supports a range of real-world projects which make use of parallel tetrahedral and mixed meshes.

Support for PUMI was provided through the Department of Energy (DOE) office of Science's Scientific Discovery through Advanced Computing (SciDAC) institute as part of the Frameworks, Algorithms, and Scalable Technologies for Mathematics (FASTMath) program, under grant DE-SC0006617.

For all inquiries on PUMI including PUMI-readable model and mesh generation, email to *shephard@rpi.edu*.

## 1 Overview

PUMI is composed of component libraries, each of them serving a particular purpose. The four that all users are likely to deal with are listed below:

1. PCU - Communication and parallel coordination
2. APF - Abstract mesh interface and field implementation
3. MDS - Underlying mesh implementation of choice
4. GMI - Interfaces with geometric models

There are other components in PUMI, which provide state-of-the-art functionality for operations like mesh adaptation and load balance.

All the source code for PUMI is contained in one repository and compiled as one. The repository can be found at:

`https://github.com/SCOREC/core`

The code is written entirely in C and C++, which gives it a decent compromise between high performance and user-friendly abstraction. More information about the C++ language can be found at:

`http://www.cplusplus.com/doc/tutorial/`

Each library provides a header file with structures and functions for users. Documentation for all these low-level structures and functions is provided here:

`http://www.scorec.rpi.edu/~seol/scorec/doxygen`

New user's should access these structures and functions through the `pumi.h` header file. Documentation is provided here:

`http://scorec.rpi.edu/~seol/PUMI.pdf`

# 2 PCU

The Parallel Control Utility (PCU) library is built on top of a Message Passing Interface (MPI) implementation, and is meant to be a higher level interface that people can use instead of MPI itself.

MPI is the standard system for building massively parallel programs to run on supercomputers, which all consist at some level of separate computers connected by a network which transmits messages between them. Although it is not necessary to know MPI in order to use PCU, learning MPI is a recommended starting point. A decent introduction to MPI can be found here:

`https://www.mpi-forum.org/docs/mpi-1.1/mpi1-report.pdf`

PCU was built on the notion that scalable parallel programs often use only the following features of MPI:

1. Basic information like `MPI_Comm_size` and `MPI_Comm_rank`
2. Collectives like `MPI_Reduce` and `MPI_Bcast`
3. More complex communications implemented with variants of `MPI_Isend` and `MPI_Irecv`

In particular, PCU implements an efficient algorithm to solve a complex termination-detection problem and provides an interface that so far has been able to replace all hand-coded algorithms of the kind described in Item 3 above.

Before we describe complex communication, lets begin with an example of simple parallel functionality:

```
#include <PCU.h>
#include <stdio.h>

int main(int argc, char** argv)
{
  int i;
  MPI_Init(&argc, &argv);
  PCU_Comm_Init();
  if (PCU_Comm_Self()==0)
    printf("There are %d total processes:\n", PCU_Comm_Peers());
  printf("Greetings from process %d\n", PCU_Comm_Self());
  i = PCU_Comm_Self() + 1;
  PCU_Add_Ints(&i, 1);
```

```
  if (PCU_Comm_Self()==0)
    printf("The sum of the integers from 1 to %d is %d\n",
        PCU_Comm_Peers(), i);
  PCU_Comm_Free();
  MPI_Finalize();
  return 0;
}
```

This program should be compiled and run as a parallel MPI job. It will print the number of MPI processes in the job, a greeting message from each of them, and the sum of the integers from 1 to $n$, where $n$ is the number of processes in the job. The output will be a bit out of order, which is a fundamental issue with outputting text from a parallel job.

Since PCU uses MPI, MPI must be initialized and finalized around PCU, which has its own initialization and finalization calls. `PCU_Add_Ints` performs an all-reduce operation using an array of integers as the input and output. In this case, the array is just one integer, $i$, which starts out as the process rank plus one and is overwritten with the sum of all such values across the job.

Now lets take a look at the "phased" communication interface of PCU. The general idea is that all processes participate in a "phase", during which each process sends out messages to other processes. PCU takes care of delivering messages such that by the end of the phase all processes have received the messages sent to them. The following program exchanges an integer message between two processes (it should only be run as a two-process job):

```
#include <PCU.h>
#include <stdio.h>

int main(int argc, char** argv)
{
  int message;
  MPI_Init(&argc, &argv);
  PCU_Comm_Init();
  message = PCU_Comm_Self() + 1;
  PCU_Comm_Begin();
  if (PCU_Comm_Self() == 0)
    PCU_COMM_PACK(1, message);
  else
    PCU_COMM_PACK(0, message);
  PCU_Comm_Send();
  while (PCU_Comm_Receive()) {
    PCU_COMM_UNPACK(message);
    printf("%d received \"%d\" from %d\n",
        PCU_Comm_Self(), message, PCU_Comm_Sender());
  }
  PCU_Comm_Free();
  MPI_Finalize();
```

```
  return 0;
}
```

A phase begins when all processes call `PCU_Comm_Begin` . After that, each process may call `PCU_Comm_Pack` or the more convenient `PCU_COMM_PACK` macro to pack data to be sent to some destination. When all packing is over, a process should call `PCU_Comm_Send` and enter a loop based on `PCU_Comm_Receive` . Inside this loop, it should unpack the stream of incoming data. `PCU_Comm_Sender` will tell where the current data is coming from; all data from the same source arrives together in the order it was sent.

The functions `PCU_Comm_Init`, `PCU_Comm_Free`, `PCU_Comm_Peers`, `PCU_Comm_Self`, `PCU_Comm_Barrier` are respectively equivalent to `pumi_start`, `pumi_finalize`, `pumi_size`, `pumi_rank` and `pumi_sync` in `pumi.h`.

# 3 Mesh

MDS is the library which handles the concrete mesh structure. The header file `apfMDS.h` provides the few functions which are not part of the `pumi.h` interface. Probably the first used function will be `pumi_mesh_load`.

```
int num_parts = pumi_size();
pGeom g = pumi_geom_load("model.dmg");
pMesh mesh = pumi_mesh_load(g, "mesh.smb", num_parts);
```

This function will read a geometric model file and mesh files into the data structure referenced by the variable called `g` and `mesh`. See Section 4 for more information about loading geometric model files.

The second argument of *pumi_mesh_load* is the mesh file name. As for the mesh files, if the mesh is partitioned, there is one file per part, where a part is the subset of the mesh stored on one process. So, a two-part mesh would have files named `mesh0.smb` and `mesh1.smb`, and a serial (single-part) mesh just has one file called `mesh0.smb`. The part number at the end of the name is inserted automatically, so calling the function with "`mesh.smb`" in the code results in `mesh0.smb` being read into process 0, and so on for other processes.

The third argument of *pumi_mesh_load* is the number of mesh files. Suppose *pumi_mesh_load* is called on $p$ processes ($p$>1) and the number of mesh file is 1 (in other words, you have only `mesh0.smb`). In this case, the serial mesh is loaded onto the master process (process 0) and partitioned to $p$ processes.

If you load the $p$-part mesh onto $p$ processes, the third argument is $p$, of which value is *pumi_size()*

At the end of your program, you should release the computer memory that was used to store the mesh and geometric model.

```
pumi_mesh_delete(mesh);
```

In some cases, you may want to construct your own small mesh. The easiest way to do this is to make an empty MDS mesh on a null geometry. See Section 4 for info on null geometry.

```
pMesh mesh = pumi_mesh_create(pumi_geom_load(NULL, "null"), 3, false);
```

The second argument to `pumi_mesh_create` specifies the dimensionality of the mesh; either 2D or 3D. The final argument of `pumi_mesh_create` secifies whether periodic associations exist between entities, this is almost always false.

Once you have constructed your mesh as described in Section 5.6, you may want to create a simple geometry around it using this special function:

```
pumi_mesh_freeze(mesh);
```

# 4 Geometric Model

GMI is the library which bridges geometric model structures to a mesh that users interact with. Interactions are mainly through `pumi.h`, but for advanced modeling operations direct calls to GMI may be necessary. Usually, though, users are concerned only with loading geometric model files. As there are many kinds of geometric models, the user has to specify the "model type" along with geometric model files. In case of PUMI API, the function *pumi_geom_load* performs both model type set-up and model loading as the second argument is a character string specifying the model type ("null" for no model, "mesh" for mesh-driven model, and "analytic" for analytic model). The simple format is the "mesh model" format whose files end in `.dmg` and contain just enough information to maintain the relationship between the mesh and geometric model. To load a mesh model,

```
pumi_geom_load(/path/to/your/mesh/model.dmg, "mesh");
```

In some cases, users do not really care about the geometric model at all. For such cases, there is a "null" geometric model that can be used, which just mimicks the minimal necessary behavior to support the rest of the code. You can get a null model as follows:

```
pumi_geom_load(NULL, "null");
```

# 5 APF

APF is a library to provide an abstract interface to various mesh implementations and fields on meshes. Part of its structures and functionalities are accessible through `pumi.h`. However, since it is abstract, users may need to interact a little with the concrete implementations behind it. When the underlying objects need to be accessed it is important to understand that a `apf::Mesh` object can be queried but not modified, while a `apf::Mesh2` object may be queried and modified. In `pumi.h`, the data type to a modifiable mesh instance (pointer to a mesh data structure) is `pMesh` .

## 5.1 Entities

These functions will refer to mesh entities in the form of `pMeshEnt` variable. One of the most fundamental pieces of information you can get out of an `pMeshEnt` variable is what kind of entity it is. The kind is encoded in a C++ enumeration value,

```
enum PUMI_EntTopology {
  PUMI_VERTEX,    // 0
  PUMI_EDGE,      // 1
  PUMI_TRIANGLE,  // 2
  PUMI_QUAD,      // 3
  PUMI_TET,       // 4
  PUMI_HEX,       // 5
  PUMI_PRISM,     // 6
  PUMI_PYRAMID    // 7
);
```

and obtained using *pumi_ment_getTopo*.

```
pMeshEnt e = ...;
int topology = pumi_ment_getTopo(e);
if (topology == PUMI_QUAD)
   std::cout << "this is a quad\n";
```

## 5.2 Numbering

First-time users may want a way to identify mesh entities for printing to application output (i.e. via `printf` and friends) and/or for viewing in Paraview (see Appendix A). A general way to do this is using a `Numbering` object. To number the regions, one can do the following:

```
pNumbering numbers = pumi_numbering_createOwned(
    mesh, "my_numbers", 3);
```

The second argument is a name of your choice, unique to this numbering object. The last argument says to number three-dimensional entities, i.e. regions.

Due to details of how we communicate with Paraview, there is a function which is better for numbering vertices than `pumi_numbering_createOwned(..., 0)`, which is `pumi_numbering_createOwnedNode`.

After numbering entities, you can call the following to obtain the number of an entity:

```
pMeshEnt vertex = ...;
int vertex_id = pumi_ment_getNumber(numbers, vertex, 0, 0);
```

The last two arguments are used to number high-order fields and multiple degrees of freedom per node, if you don't use those advanced features just set them to zero.

In other cases, you may want to assign numbers yourself. To do that, start by making an empty numbering:

```
pNumbering numbers = pumi_numbering_create(
       mesh, "my_numbers",  pumi_mesh_getShape(mesh), 1);
```

The third argument specifies that we are numbering the mesh nodes and the last argument specifies that there is just one number per node. Then you can set the numbers yourself using `pumi_ment_getNumber`.

Also note that `Numbering` is a good way to store any integer value on mesh entities, not necessarily an ordered numbering.

## 5.3 Iteration

Most operations will require iterating over mesh entities, which can be done one dimension at a time. Iteration uses an iterator pointer as follows:

```
pMeshIter it = mesh->begin(1);
pMeshEnt e;
while ((e = mesh->iterate(it))) {
  /* use "e" somehow */
}
mesh->end(it);
```

## 5.4 Coordinates and Vector3

You can query and modify the coordinates of mesh nodes. Using `pumi.h` the coordinate information is handled in the form of an array of three double-precision floating point values. For example, the following code changes the y-coordinate of vertex `v` and print:

```
#include <iostream>
#include <pumi.h>
double v[3];
pumi_node_getCoord(vertex, 0, v);
v[1] = 0.8;
pumi_node_setCoord(vertex, 0, v);
std::cout << "vertex coord (" << v[0]<<", "<<v[1]<<", "<<v[2]<< ")\n";
```

In the low-level implementation, the coordinate information is handled in the form of `Vector3` objects. You can treat `Vector3` just like an array, but it is possible to apply mathematical operators like addition, subtraction, multiplication by a scalar, etc. to these objects. There is also a C++ stream output operator for these objects, which means they can be given to C++'s `std::cout`.

```
#include <iostream>
#include <pumi.h>
Vector3 v1;
pumi_node_getCoordVector(vertex1, 0, v1);
```

```
std::cout << "vertex coord "<<v1<<"\n";
Vector3 v2(8.0 / 4 * i, 8.0 / 6 * j, 0);
pumi_node_setCoordVector(vertex2, 0, v2);
// cross product of v1 and v2
double V = pumi_vector3_cross(v1, v2);
```

The second argument to set/get coordination information is intended to handle high order fields where there is more than one node per mesh entity. For first-order mesh entities, set this to zero. For quadratic meshes, these functions will operate on edges as well as vertices.

Speaking of quadratic meshes, you can convert a linear mesh to a quadratic mesh using the following call:

```
pumi_mesh_setShape(mesh, pumi_shape_getLagrange(2));
```

## 5.5 Adjacency

Most finite element algorithms require adjacency information. The most important information is downward adjacency, for example, getting the vertices of an element.

```
pMeshEnt element = ...;
std::vector<pMeshEnt> v;
pumi_ment_getAdj(element, 0, v);
size_t numVertices = v.size();
pMeshEnt lastVertex = v[numVertices - 1];
```

The second argument to `pumi_ment_getAdj` is the target dimension, in this case 0 meaning vertices.

The inverse of downward adjacency is upward adjacency, for example, what elements are around a vertex.

```
pMeshEnt vertex = ...;
std::vector<pMeshEnt> e;
pumi_ment_getAdj(vertex, pumi_mesh_getDim(m), e);
pMeshEnt element = e[e.size() - 1 ];
```

Below is a full example program demonstrating file reading, adjacency queries, and coordinate output using PUMI API. It will print one line for each element in the mesh, containing its vertex coordinates.

```
#include <mpi.h>
#include <vector>
#include <iostream>
#include "pumi.h"

int main(int argc, char** argv)
{
  MPI_Init(&argc, &argv);
```

```
  pumi_start(); // equivalent to PCU_Comm_Init()
  pGeom g = pumi_geom_load("cube.dmg", "mesh");
  pMesh mesh = pumi_mesh_load(g, "tet-mesh-1.smb", pumi_size());
  pMeshIter it = mesh->begin(pumi_mesh_getDim(mesh));
  pMeshEnt e;
  while ((e = mesh->iterate(it))) {
    std::vector<pMeshEnt> vertices;
    pumi_ment_getAdj(e, 0, vertices);
    size_t numVertices = vertices.size();
    for (size_t i = 0; i < numVertices; ++i) {
      double point[3];
      pumi_node_getCoord(vertices[i], 0, point);
      std::cout <<"point "<<i<<": "<<point[0]<<' '<<point[1]<<' '<<point[2]<<' ';
    }
    std::cout<<"\n";
  }
  mesh->end(it);
  pumi_mesh_delete(mesh);
  pumi_finalize(); // equivalent to PCU_Comm_Free()
  MPI_Finalize();
}
```

### 5.6 Construction

Finally, you may want to construct your own mesh at some point. This is usually best separated into two steps:

1. create the vertices at the right points.
2. create the elements based on their vertices

Prior to these steps, refer to Section 3 for how to create an empty mesh. Then use `pumi_mesh_createVtx` to create a vertex. Its arguments are mesh instance, the geometric classification, and *xyz* coordinates. The geometric classification can be left as *NULL* if the model derivation described in 3 is used after construction.

It is advisable to store the vertex pointers in some sort of structure to prepare them for the second step, which is to create elements using `pumi_mesh_createElem`. Note that `pumi_mesh_createElem` is given an array of vertices, and the order of those vertices is important. The documentation for `apf::Mesh::getDownward` shows pictures of the expected ordering for vertices in this array, for each element type.

Finally, when mesh construction is over, the function `pumi_mesh_freeze` has to be called to finalize the mesh data structure. After `pumi_mesh_freeze` is called, it is possible to run a stringent verification on the mesh structure itself, to make sure no mistakes were made during construction. This is done using `pumi_mesh_verify`.

Here is a full example code which constructs a one-tet mesh and writes Paraview files.

```
#include <mpi.h>
#include "pumi.h"

int main(int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  pumi_start();

  pGeom g = pumi_geom_load(NULL, "null");
  pMesh mesh = pumi_mesh_create(g, 3, false);
  double points[4][3] = {{0,0,0},{1,0,0},{0,1,0},{0,0,1}};
  pMeshEnt vertices[4];

  for (int i = 0; i < 4; ++i)
    vertices[i] = pumi_mesh_createVtx(mesh, NULL, points[i]);
  pumi_mesh_createElem(mesh, NULL, PUMI_TET, vertices);

  pumi_mesh_freeze(mesh);
  pumi_mesh_verify(mesh);
  pumi_mesh_write(mesh, "onetet", "vtk");
  pumi_mesh_delete(mesh);

  pumi_finalize();
  MPI_Finalize();
}
```

## 5.7 Classification

When the mesh is classified onto a geometric model of some sort, we can query that relationship. In `pumi.h`, the data type to a geometric model instance (a pointer to geometric model data structure) is `pGeom`. Given a mesh instance, the function `pumi_mesh_getGeom` returns its associated geometric model instance. The data type to a geometric model entity is `pGeomEnt` . For many meshing operations, two integers are all the information we need about a geometric model entity. First, its dimension (like mesh entities, there are vertices, faces, etc.). Second, a unique integer ID which sets the entity apart from all other entities of the same dimension (these ID's are not necessarily unique across *all* model entities). Given a geometric model entity, the function `pumi_ment_getGeomClas` returns the geometric model entity instance where it's generated from.

```
pGeom g = pumi_mesh_getGeom(mesh);
pMeshEnt e = ...;
pGeomEnt ge = pumi_ment_getGeomClas(e);
```

```
int dimension = pumi_gent_getDim(ge);
int id = pumi_gent_getID(ge);
assert(pumi_geom_getNumEnt(g, dimension));
```

## 5.8 Parallel Meshes

Once you are dealing with parallel, or partitioned, meshes, it will be important to get information about how the mesh is partitioned. In PUMI, we partition meshes by elements. That is, each element will exist in exactly one part. All mesh entities which bound that element will also exist on the same part. Because mesh entities may bound multiple elements, they may exist on multiple parts. This is where remote copies come in. Each mesh entity stores information about the other parts on which it exists, including pointers to its other copies.

```
pMeshEnt vertex = ...;
Copies remotes;
pumi_ment_getAllRmt(vertex, remotes);
for (pCopyIter it = remotes.begin();
     it != remotes.end(); ++it)
  std::cout << "shared with part " << it->first << '\n;
```

`Copies` is a `std::map` whose keys are part numbers and whose values are pointers to copies on those parts. We will return later to the issue of these pointers and why they are there.

Before that, a few more basics. First, recall from Section 2 that `pumi_rank` gives you the process rank. This is identical to the part number, so remember that the two can be used interchangeably. Second, given a mesh entity with copies, the PUMI system decides on one of the copies to designate as the owner. You can check whether the local copy is the owner:

```
if (pumi_ment_isOwned(vertex))
  std::cout << "yes\n";
```

Now we can put those concepts all together and write a program which sets up a numbering by first assigning numbers to the vertices which it owns, and then communicating those numbers to the non-owned copies so that all copies of the same vertex have the same number. To do this, we need to use the pointer values in `Copies`. This gives us `pMeshEnt` values which are usable on the destination part to refer to the copy.

```
#include <mpi.h>
#include "PCU.h"
#include "pumi.h"

int main(int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  pumi_start();
```

```
pGeom g = pumi_geom_load("cube.dmg", "mesh");
pMesh mesh = pumi_mesh_load(g, "parallelMesh.smb", pumi_size());

pMeshEnt vertex;
pMeshIter it;
int numOwned = 0;
it = mesh->begin(0);
while ((vertex = mesh->iterate(it)))
  if (pumi_ment_isOwned(vertex))
    ++numOwned;
mesh->end(it);
int offset = numOwned;
PCU_Exscan_Ints(&offset, 1);
pNumbering numbers = pumi_numbering_create(
    mesh, "my_numbers", pumi_mesh_getShape(mesh), 1);
int i = offset;
it = mesh->begin(0);
while ((vertex = mesh->iterate(it)))
  if (pumi_ment_isOwned(vertex))
    pumi_ment_setNumber(vertex, numbers, 0, 0, i++);
mesh->end(it);
PCU_Comm_Begin();
it = mesh->begin(0);
while ((vertex = mesh->iterate(it)))
  if (pumi_ment_isOwned(vertex)) {
    Copies remotes;
    pumi_ment_getAllRmt(vertex, remotes);
    int number = pumi_ment_getNumber(vertex, numbers, 0, 0);
    for (pCopyIter it = remotes.begin();
         it != remotes.end(); ++it) {
      PCU_COMM_PACK(it->first, it->second);
      PCU_COMM_PACK(it->first, number);
    }
  }
mesh->end(it);
PCU_Comm_Send();
while (PCU_Comm_Receive()) {
  int number;
  PCU_COMM_UNPACK(vertex);
  PCU_COMM_UNPACK(number);
  pumi_ment_setNumber(vertex, numbers, 0, 0, number);
}
pumi_mesh_write(mesh, "numbered", "vtk");
pumi_mesh_delete(mesh);
pumi_finalize();
```

```
  MPI_Finalize();
}
```

### 5.9 Migration

An additional topic of interest when dealing with parallel meshes is migration. In this case, that means the ability to send elements from one part to another. Since each part is controlled by a separate process, the process collects a list of elements which it wants to send and the destination parts to which it wants to send them. This "plan" is stored in an `Migration` object. This object should be created like this:

```
Migration* plan = new Migration(mesh);
```

Once the migration plan of elements to send from the local process is complete, it is given to the `pumi_mesh_migrate` function. All processes should call this function at the same time, even if they have nothing to send. This function will delete the plan, so users do not have to delete it themselves.

## A Paraview

Paraview is program created by Kitware, Inc. which can visualize meshes and fields on meshes. It is the program of choice for viewing meshes created by the PUMI libraries. PUMI provides the function `pumi_mesh_write` in the `pumi.h` header file. If this code is executed for a mesh distributed over two processes:

```
pumi_mesh_write(mesh, "output", "vtk");
```

It would create the folder `output` and files `0.vtu`, `1.vtu`, and `output.pvtu`. Opening the `output.pvtu` file in Paraview will show users the mesh.

By default, Paraview will just render the mesh in "Surface" mode. Changing this to "Surface with Edges" will outline each visible element, actually making the decomposition visible.

Also, the mesh by default is rendered in one "Solid Color". There should be other options corresponding to the fields and numberings that were on this mesh at the time of file writing. There is usually an "apf_part" alternative for files written by PUMI, which allows users to see the parallel partitioning of the mesh in color.

When vertices are numbered, it may be useful to display their numbers. See Section 5.2 for information on creating numberings. Right above the mesh viewing area there is a button to select nodes, you may also press the "D" key.

Click and drag to select all the nodes you want to display. Then go to `View->Selection Display Inspector` in the menu and click on the Point Labels options. There you can choose what to display. If you used `Numbering`

properly, there should be an option with the same name that you gave to the numbering. Note that in some later versions of Paraview, there is a bug which displays all values as floating point numbers by default. If you are trying to show `Numbering` values, you may see strange scientific notation instead. Click the following icon in the Selection Display Inspector:

There you will find a format string option, which you can change to "`%d`" in order to show integers.