DENDRO Manual

Version 1.0

Rahul S. Sampath, Hari Sundar, Santi S. Adavani,
Ilya Lashuk & George Biros

Computational Science and Engineering Laboratory

University of Pennsylvania, Philadelphia, PA

&

Georgia Institute of Technology, Atlanta, GA

November 18, 2016

Contents

Chapter 1

Introduction

'DENDRO' is a collection of tools for parallel tree-based applications. It is written in C++ using the standard template library (STL) and uses the Message Passing standard, MPI. It supports the PETSc objects 'Mat' and 'Vec' and provides interfaces to PETSc's linear and non-linear solvers.

1.1   Layout of DENDRO

The DENDRO library comprises of 4 main modules, which are listed below:

1. 'par': This is a templated library of parallel functions used for sorting, merging, searching and partitioning data.

2. 'oct': This comprises of the functions used to construct and balance linear octrees.

3. 'oda': This comprises of the functions used to construct and manage the finite element mesh and provides interfaces to create the PETSc

objects 'Mat' and 'Vec'.

4. 'omg': This comprises of the functions used to set up and use the multigrid solver on the octree based finite element discretization.

There is another module, 'fem', which provides some generic classes to help implement FEM matrix-vector multiplications.

1.2   Organisation of the manual

Chapter **??** gives detailed instructions for installing the library on UNIX based systems. Use 'cygwin' to install on Windows machines. Chapter **??** describes how to use DENDRO to solve partial differential equations (PDEs) using the finite element method (FEM). Chapter **??** described how to use the multigrid library. Note, throughout the manual we refer to different functions in the library. However, we do not give the detailed header or description of those functions in this manual. Please refer the 'Doxygen' documentation for the syntax and description of the functions.

Chapter 2

Installation

1. Install PETSc (version 2.3.3) with the option '-with-clanguage=cxx'.
   This will build the PETSc library using a 'C++' compiler. Follow the
   instructions given on the PETSc Homepage (www-unix.mcs.anl.gov/petsc/petsc-
   as/index.html) to do this.

2. Check that the environment variables 'PETSC_DIR' and 'PETSC_ARCH'
   point to the correct PETSc installation. You can do this by typing the
   following command at the command line.

   ```
   echo $PETSC_DIR; echo $PETSC_ARCH
   ```

   The first line of the ouput should point to the directory in which you in-
   stalled PETSc and the second line must be the value of 'PETSC_ARCH'
   that you set while compiling PETSc.

3. Set the environment variable DENDRO_DIR to point to the top level
   directory where you want to install DENDRO.

4. Create a directory named 'lib' (if it is not already present) under the top level directory, 'dendro'.

5. Type 'make' to build the library and the example programs.

6. You can provide additional optimization flags to the compiler by setting the flag 'CFLAGS' in the makefile from the command line.

7. The functions in the library are profiled using PETSc. To profile the major functions in the library, pass the flag '-DPETSC_USE_LOG' to the compiler in the variable 'CFLAGS' in the makefile, while building the library. You can then view the log summary by passing the option '-log_summary' to your executable.

8. By default, DENDRO will print some relevant information during the execution of the program. If you want to suppress this output pass the flag '-D_SILENT_MODE_' to the compiler in the variable 'CFLAGS'.

9. By default, DENDRO will be compiled with 32 bit global indices. If you are solving very large problems that require 64 bit indices to represent the global sizes, then pass the flag '-D__USE_64_BIT_INT__' to the compiler in the variable 'CFLAGS'.

10. You must copy the '*.inp' files from the 'Stencils' folder into the directory containing the executable before running any program that uses the 'omg' module, i.e. multigrid solver.

11. There are few example files that you can try out. Read the file 'EXAMPLES_DOC' for more information about the examples. Section **??**

discusses a few of these.

Chapter 3

Solving partial differential equations (PDEs) using DENDRO

One of the features of the DENDRO framework for FEM simulations is that 'hanging' nodes[1] are not stored explicitly. If the i-th vertex of an element/octant is 'hanging', then the index corresponding to this node will point to the i-th vertex of the parent[2] of this element instead. Thus, if a hanging node is shared between 2 or more elements, then in each element it might point to a different index. A description of this proceedure can be found in the following references:

1. Hari Sundar et al., *Low-constant parallel algorithms for finite element*

---

[1]Nodes that exist at the center of a face of another octant are called face-hanging nodes and the nodes that are located at the centers of an edge of another octant are called edge-hanging nodes.

[2]The 2:1 balance constraint ensures that the nodes of the parent can never be hanging.

simulations using linear octrees. SC'07.

2. Weigang Wang, *Special bilinear quadrilateral elements for locally refined finite element grids.* SIAM Journal on Scientific Computing, 2001.

3.1 Outline of the solution proceedure

Follow the following steps to solve a PDE using the Finite Element Method (FEM) and the DENDRO framework.

1. The first step is to read in the point data from which the octree shall be constructed. This can be done by the user based on whatever format they are most comfortable with. The octree construction function requires the points to be specified as a STL `vector` of `double`. We provide the following routine to read binary point dataset files.

   ```
   ot::readPtsFromFile();
   ```

   The **ot::readPtsFromFile()** function is provided mainly as a template to help the user write IO routines compatible with the DENDRO library.

2. Once the data has been read, we can go ahead and construct the Octree. This can be done using the **ot::points2Octree()** function. This generates a linear complete octree based on the input points. Once the octree is generated, the points are no longer needed and can be cleared.

3. In order to use the octree for FEM calculations, we need to enforce the 2 : 1 balance constraint. This can be done using the **ot::balanceOctree()**

function. It will produce the optimal balanced complete linear octree for a given input octree. The linear octree is no longer needed and can be cleared to free memory.

4. Generate the stencils for the finite element matrices using conforming trilinear lagrange shape functions. This could be done offline using a package like Matlab. Refer sections **??** and **??** before deriving the stencils. The folder Matlab also contains some m-files that will be useful for this step.

5. Construct the finite element mesh. This can be done using the `ot::DA` class' constructor. Refer section **??** to do this. If you are using the multigrid solver, then there is no need to call the constructor of `ot::DA` explicitly. Instead, follow the steps outlined in Chapter **??**.

6. Create the PETSc object 'Mat', which is used to store the finite element matrices. Refer the PETSc manual and section **??** to do this. For matrix-free methods, use the matrix type 'MATSHELL'. This is described in the PETSc manual.

7. Create the PETSc object 'Vec', which is used to store the solution and right hand side vectors. Refer the PETSc manual and section **??** to do this.

8. Create distributed vectors for storing material properties. These vectors could either be STL[3] vectors or PETSc Vec objects. Refer section **??** for STL vectors and section **??** for PETSc Vec objects.

---

[3]C++ standard template library

9. Refer section **??** to learn about the various iterators available in DEN-DRO.

10. Set values into the vectors. This could be for storing material properties or building the right hand side or setting an initial guess. Refer section **??** to do this.

11. Assemble and store the finite element matrices in one of the formats supported by PETSc. Refer section **??** to do this. Creating and destroying matrices can be expensive and hence this must be done only for small matrices. For large dimensional problems, it would probably be better to use a matrix-free method.

12. Solve the equations using PETSc's linear and/or nonlinear solvers. Refer the PETSc manual to do this.

13. Alternatively, PETSc's multigrid solver could be used with the interface provided in DENDRO. Refer Chapter **??** to do this.

3.2 Child numbers

The 'child number' of an octant is very important in deriving and using the finite element stencils. An octant's position with respect to its parent is identified by specifying the node that it shares with its parent. If an octant shares its k-th node with its parent, then it is said to have a 'child number' equal to k. For convenience, we use the Morton ordering[4] to number the

---

[4]Paul M. Campbell et al., *Dynamic octree load balancing using space-filling curves.* Technical Report CS-03-01, Williams College, Department of Computer Science, 2003.

nodes of an octant. Thus, sorting the children of an octant in the Morton order is the same as sorting the children according to their child numbers. Section **??** describes how the 'child number' of an octant is used in deriving the finite element stencils.

The child number of an octant is a function of the coordinates of its 'anchor'[5] and its level in the tree. In DENDRO, the octree is stored in a compressed format and the coordinates of the anchor of an octant are not directly available. Instead, the coordinates are computed on the fly as one loops through all the octants. For this reason, DENDRO does not support random queries into the octree and such access patterns are not necessary for FEM calculations. Hence, one must use the following function to get the 'child number' of an octant.

```
unsigned char ot::DA::getChildNumber()
```

This is an inline member function of the class ot::DA. This must be called only from within a loop over the elements using the iterators provided in the class ot::DA.

3.3   Hanging node configurations

To be able to use the DENDRO framework for FEM, it is important to understand the following properties of balanced linear octrees. Figure **??** illustrates these properties.

1. Every octant has at least 2 non-hanging nodes and they are:

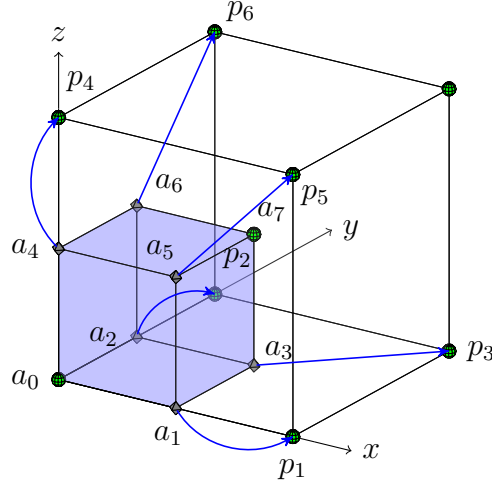[5]The 0-node of an element is called the anchor of the element.

Figure 3.1: Illustration of nodal-connectivities required to perform conforming FEM calculations using a single tree traversal. Every octant has at least 2 non-hanging nodes, one of which is shared with the parent and the other is shared amongst all the siblings. The octant shown in blue (a) is a child 0, since it shares its zero node (a_0) with its parent. It shares node a_7 with its siblings. All other nodes, if hanging, point to the corresponding node of the parent octant instead. Nodes, a_3, a_5, a_6 are face hanging nodes and point to p_3, p_5, p_6, respectively. Similarly a_1, a_2, a_4 are edge hanging nodes and point to p_1, p_2, p_4. All the nodes in this illustration are labelled in the Morton ordering.

    (a) The node that is common to both this octant and its parent.

    (b) The node that is common to this octant and all its siblings.

2. An octant can have a face hanging node only if the remaining nodes on that face are one of the following:

    (a) Edge hanging nodes.

    (b) The node that is common to both this octant and its parent.

In each of the 8 child number based configurations (Figures **??** - **??**), v_0 is the node that this element shares with its parent and v_7 is the node that

this element shares with all its siblings. For an element with child number k, v_0 will be the k-th node and v_7 will be the (7-k)-th node. v_0 and v_7 can never be hanging. If v_3, v_5 or v_6 are hanging, they will be face-hanging and not edge-hanging. If v_3 is hanging, then v_1 and v_2 must be edge-hanging. If v_5 is hanging, then v_1 and v_4 must be edge-hanging. If v_6 is hanging, then v_2 and v_4 must be edge-hanging. After factoring in these constraints, there are only 18 potential hanging node configurations for each of the 8 'child number' configurations.

The hanging configuration of each octant is stored in an 'unsigned char' (8 bits) known as the 'hanging mask'. If the i-th node of the octant is hanging, then this mask has a 1 stored in its i-th bit[6] and 0 otherwise. The following functions can be used to find the hanging configuration of an octant.

```
bool ot::DA::isHanging(unsigned int i)
```

The index, 'i', can be obtained either by using the function **ot :: DA::curr()** or **ot :: DA::getNodeIndices()**.

```
unsigned char ot::DA::getHangingNodeIndex(unsigned int i)
```

DENDRO contains the **class ot :: cNumEtype** (templated on the child number), which has a member enumeration called **eTypes**. The enumeration 'eTypes' has 18 named constants (enumerators) defined within it. The values of these constants are simply the hanging masks corresponding to the 18 hanging configurations. Table **??** lists these 18 constants and the corresponding hanging nodes.

DENDRO also provides the following function, which can be used to map

---

[6]The bits of the mask are numbered in the standard binary format, i.e. the right most bit is bit-0 and the left most bit is bit-7.

a given child number and hanging mask to one of the 18 element types.

```
template<unsigned char cNum>
unsigned char ot::getElemType(unsigned char hnMask)
```

The following macro can also be used to map a given child number and hanging mask to one of the 18 element types.

```
GET_ETYPE_BLOCK()
```

At present, DENDRO only supports trilinear lagrange shape functions. The salient differences between the shape functions for regular grids and those for the DENDRO framework are the following.

1. There are no shape functions rooted at the hanging nodes.

2. The support of a shape function can spread over more than 8 elements.

3. If a node of an element is hanging, then the shape functions rooted at the other regular nodes in that element do not vanish on this hanging node. Instead, they will vanish at the regular node that this hanging node is mapped to. For example, in Figure **??** the shape function rooted at node a_0 will not vanish at nodes a_1, a_2, a_3, a_4, a_5 or a_6. It will vanish at nodes p_1, p_2, p_3, p_4, p_5, p_6 and a_7. It will assume a value equal to 1 at node a_0.

One could pre-compute the finite element stencils for all the 8x18 element types. While doing this, it is best if one uses the same numbering convention as is used within DENDRO (Table **??**). Alternatively, one could just pre-compute the stencils for a specific child number configuration, e.g. child 0 and use rotations to compute the stencils for the other types on the fly inside

Element Type	Named constants (Enumerators)	Hanging nodes
0	ET_N	None
1	ET_Y	v_2
2	ET_X	v_1
3	ET_XY	v_1 and v_2
4	ET_Z	v_4
5	ET_ZY	v_4 and v_2
6	ET_ZX	v_4 and v_1
7	ET_ZXY	v_4, v_1 and v_2
8	ET_XY_XY	v_3, v_1 and v_2
9	ET_XY_ZXY	v_3, v_4, v_1 and v_2
10	ET_YZ_ZY	v_6, v_4 and v_2
11	ET_YZ_ZXY	v_6, v_4, v_1 and v_2
12	ET_YZ_XY_ZXY	v_6, v_3, v_4, v_1 and v_2
13	ET_ZX_ZX	v_5, v_4 and v_1
14	ET_ZX_ZXY	v_5, v_4, v_1 and v_2
15	ET_ZX_XY_ZXY	v_5, v_3, v_4, v_1 and v_2
16	ET_ZX_YZ_ZXY	v_5, v_6, v_4, v_1 and v_2
17	ET_ZX_YZ_XY_ZXY	v_5, v_6, v_3, v_4, v_1 and v_2

Table 3.1: The list of element types and the corresponding enumerators and hanging nodes.

(a) Child 0   (b) Child 1   (c) Child 2

(d) Child 3   (e) Child 4   (f) Child 5
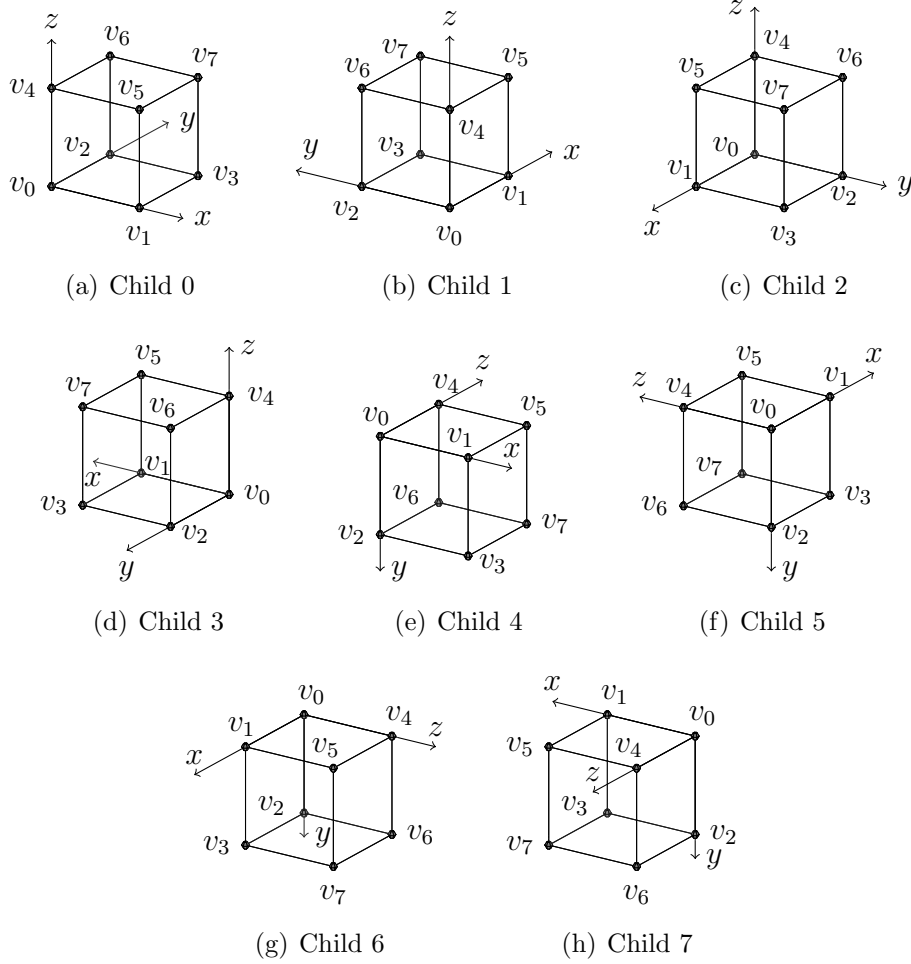
(g) Child 6   (h) Child 7

Figure 3.2: Local coordinate systems for each of the 8 child number config-urations. In each of these configurations, v_0 is the node that this element shares with its parent and v_7 is the node that this element shares with all its siblings. For an element with child number k, v_0 will be the k-th node and v_7 will be the (7-k)-th node. v_0 and v_7 can never be hanging. If v_3, v_5 or v_6 are hanging, they will be face-hanging and not edge-hanging. If v_3 is hanging, then v_1 and v_2 must be edge-hanging. If v_5 is hanging, then v_1 and v_4 must be edge-hanging. If v_6 is hanging, then v_2 and v_4 must be edge-hanging.

16

the matvec. In this case, it will be useful to work with the following two types of coordinate systems:

1. The global coordinate system, which is also the local coordinate system for elements with child number 0. DENDRO consistently uses the global coordinate system in all its functions.

2. The local coordinate system of an element depends on its child number. For an octant with child number equal to 'k', the local coordinate system is simply defined as a right handed coordinate system with origin at node 'k' and oriented so that the node $(7 - k)$ is facing in the +x,+y,+z directions (in the local coordinate system) from the origin (node 'k'). Figures **??** - **??** illustrates the local coordinates systems for each of the 8 'child number' configurations.

The stencils can be compressed further if the differential operator under consideration is invariant under a cyclic rotation of the coordinates, i.e. $(x \leftarrow y, y \leftarrow z, z \leftarrow x)$. In this case, it is not even necessary to pre-compute the stencils for all the 18 different hanging node configurations. For example, the scalar laplacian only needs 8 different types of stencils, the stencils for the remaining types can be derived using simple linear transformations. We will not discuss these transformations here.

3.4 Constructing the finite element mesh

The finite element mesh is constructed by calling the constructor of the class ot::DA. This will create the element-to-node mappings and will set up the

scatter contexts used for communicating ghost values inside the matvecs. If you want to use the multigrid solver, then there is no need to call this constructor explicitly. While setting up the multigrid context, this constructor will be called internally for each level.

3.5   Creating a PETSc Mat object using ot::DA

DENDRO provides the following functions to query for the number of nodes and/or elements owned by each processor. This information is sufficient to create any of the Matrix types supported by PETSc. Refer the PETSc manual to learn about the various matrix types and how to create them.

Use the following functions to get the number of nodes and elements owned by the calling processor.

```
unsigned int ot::DA::getNodeSize()
unsigned int ot::DA:getElementSize()
```

To get the sizes including ghosts as well use the following functions:

```
unsigned int ot::DA::getGhostedNodeSize()
unsigned int ot::DA::getGhostedElementSize()
```

Since, most FEM matrices will have a local size equal to the local node size on each processor, DENDRO provides a wrapper for this type of matrices. This is listed below:

```
int ot::DA::createMatrix(Mat& M, MatType mtype, unsigned int dof = 1)
```

The local size of the matrix will be (dof $\times$ ot::DA::getNodeSize()). Use the function **createActiveMatrix()** instead to create a matrix that is distributed only on the active processors.

One should not set the matrix entries directly using the functions provided in PETSc. This will not be possible as the indexing used by PETSc is different from that used in DENDRO. Section **??** describes how to set values into a matrix using DENDRO.

3.6 Creating a PETSc 'Vec' object using ot::DA

One can query DENDRO for the local sizes using the functions described in Section **??** and use them to create PETSc vectors as described in the PETSc manual. Alternatively, one can use the following wrapper provided in DENDRO.

```
int createVector(Vec & local, bool isElemental,
 bool isGhosted, unsigned int dof = 1)
```

Use the function **createActiveVector()** to create a vector that is distributed only on the active processors. To delete these vectors use PETSc's **VecDestroy()** function.

3.7 Creating a distributed STL vector object using ot::DA

DENDRO provides the following wrapper to create distributed STL vectors.

```
template <typename T>
int createVector(std::vector<T>& local, bool isElemental,
 bool isGhosted, unsigned int dof = 1)
```

The above described function can be used with user-defined datatypes

as well and one can even communicate messages of this datatype across processors. To use the wrappers provided in DENDRO to communicate messages of user-defined datatypes, one must create a template specialization of the following class.

```
template <typename T> class par::Mpi_datatype
```

Note, this is only an abstract class declaration. To use it, one must define a template specialization of this class and one must also define the static member function **value()** as shown in the examples below.

An example of template specialization for the **struct FourIntsTwoChars** is given below.

```
struct FourIntsTwoChars { int iArr[4]; char cArr[2]; };
namespace par {
template<typename T>
class Mpi_datatype<FourIntsTwoChars>
{
public:
static MPI_Datatype value() {
static bool first = true;
static MPI_Datatype datatype;
if(first) {
        first = false;
  MPI_Type_contiguous(sizeof(FourIntsTwoChars),
        MPI_BYTE, &datatype);
        MPI_Type_commit(&datatype);
}
return datatype;
}
};
}
```

To learn about other ways to create user-defined MPI_Datatypes refer the MPI standard.

3.8 Looping over the octants

Every octant is owned by a single processor. Since all nodes, except for boundary nodes in the positive directions, can be uniquely associated with an element (the element with its anchor at the same coordinate as the node) we use an interleaved representation where a common index is used for both the elements and the nodes. Thus the processor that owns an octant automatically gets to own the node located at this octant's anchor as well[7]. To handle the boundary nodes on the positive faces, we add a layer of 'pseudo-octants' and the boundary nodes will be the anchors of these pseudo-octants. The processors that own these pseudo-octants will own the respective boundary nodes as well. Only the processor that owns an octant/node is allowed to write data to it. Thus, the octants owned by a processor are collectively referred to as 'writables'.

In finite element computations, the values of unknowns associated with octants on inter-processor boundaries need to be shared among several processors. We keep multiple copies of the information related to these octants and we term them 'ghost' octants or simply 'ghosts'. If a processor must write some data to a ghost, then it must send this data to the processor that owns this node and the recieving processor will write this data. The octants a processor recieves from the processors with a rank lower than that of its own are referred to as 'pre-ghost' octants or simply 'pre-ghosts'. The octants a processor receives from the processors with ranks higher than that of its own

---

[7]The partition of the octree is such that if an octant's anchor is hanging then the regular node that this hanging node will map to lies on the same processor as this hanging anchor. Thus, hanging anchors do not pose any special problems while determining ownership.

are called 'post-ghost' octants or simply 'post-ghosts'. Each processor only stores the element-to-nodal mappings of the pre-ghosts and writables. Hence, looping over 'post-ghosts' is not allowed. Also, looping over 'pseudo-octants' is not allowed and therefore boundary nodes must be accessed indirectly using the element-to-nodal mappings of the regular octants touching the boundary.

As mentioned earlier, DENDRO does not support random queries into the octree and one must use the iterators provided in DENDRO to loop over the octants. The enumeration 'loopType', which is a member of the namespace ot::DA_FLAGS, has a list of enumerators defined within it. Table **??** lists these enumerators and the loop type they define. An example of an 'ALL' loop is given below. The 'WRITABLE', 'INDEPENDENT', 'DEPENDENT' and 'W_DEPENDENT' loops are similar to the 'ALL' loop, the only difference being the enumerator used in the loop.

```
ot::DA* = da;
if(da->iAmActive())
{
  for( da->init<ot::DA_FLAGS::ALL>();
      da->curr() < da->end<ot::DA_FLAGS::ALL>();
      da->next<ot::DA_FLAGS::ALL>() )
 {
 }
}
```

In the above example 5 different functions were used. **iAmActive()** returns 'true' if the invoking processor owns any octants and 'false' otherwise. **init()** initializes the iterator, so that the subsequent calls to **curr()** will point to an octant of the specified type, which could be any of the 6 enumerators listed in Table **??**. **curr()** returns the current position of the iterator.

It can be called any number of times within the loop as well. It does not modify the state of the iterator. **end()** return an index greater than that of the last octant on the invoking processor that is of the specified type. **next()** increments the iterator to point to the next octant of the specified type and returns the index of that octant.

An example of a loop using the 'FROM_STORED' enumerator is shown below.

```
for( da->init<ot::DA_FLAGS::FROM_STORED>();

            da->curr() < da->end<ot::DA_FLAGS::ALL>();

            da->next<ot::DA_FLAGS::ALL>() )
{
}
```

Note, the 'FROM_STORED' enumerator is only used in the initialization part of the for loop. This is because, the enumerator 'FROM_STORED' does not define any specific looping pattern in itself. It is to be used only while continuing a loop that was terminated abruptly based on some user specified conditions within the loop. Hence, the conditional and the incrementing portions of the for loops will be the same as those for the loop that was terminated abruptly. Also note that in the loop that was terminated abruptly, the call to the function **curr()** must be replaced by a call to the function **currWithInfo()**. This function will store the current state of the iterator, which is required to initialize the iterator when the enumerator 'FROM_STORED' is used.

Enumerator/Loop type	Use
ALL	Looping over all the pre-ghost and writable octants.
WRITABLE	Looping over writable octants only.
DEPENDENT	Looping over those octants (pre-ghosts and writables) that point to at least one ghost node, i.e. their element-node mappings contains an entry for a ghost node. This will be useful while overlapping communication with computation.
INDEPENDENT	Looping over those writable octants that do not point to any ghost nodes. This will be useful while overlapping communication with computation.
W_DEPENDENT	Looping over those writable octants that point to at least one ghost nodes. This will be useful while overlapping communication with computation.
FROM_STORED	If any of the above loops were terminated in the middle and the last index was stored, then this can be used to continue from that entry. This will be useful while overlapping communication with computation.

Table 3.2: The enumerators defined within the enumeration 'ot::DA_FLAGS::loopType'. These will be used to loop over the octants.

3.9 Setting values in vectors

Follow the following steps to set values into a vector.

1. Get a buffer for the vector using one of the following two functions.

    (a) For PETSc vectors use:

    ```
    int ot::DA::vecGetBuffer(Vec v, PetscScalar*& arr,
     bool isElemental, bool isGhosted,
     bool isReadOnly, unsigned int dof = 1)
    ```

    Note, the values for 'isElemental' and 'isGhosted' must be the same as those used for creating the vector. If 'isElemental' and 'isGhosted' are both 'true', then this does not involve any copies. In this case just a pointer to the data is returned. In all other cases, the necessary amount of memory for the buffer is allocated and the data in the vector will be copied to the buffer.

    (b) For distributed STL vectors use:

    ```
    template <typename T>
    int ot::DA::vecGetBuffer(std::vector<T>& v, T*& arr,
     bool isElemental, bool isGhosted,
     bool isReadOnly, unsigned int dof = 1)
    ```

    After you have finished using the buffers, call the respective **vecRestoreBuffer()** functions to free the memory for the buffers and copy values back to the vectors.

2. Read values from other processors into the space allocated for ghost nodes using the following pair of functions. This is a 2 step proceedure

and one can place some code in between these 2 function calls to overlap communication with computation.

```
template <typename T>
 int ot::DA::ReadFromGhostsBegin(T* arr, unsigned int dof = 1)
```

Any exisitng values will be overwritten by those recieved from other processors. The 'assignment' operator (=) must be defined for the type 'T'. The **class par::Mpi_datatype<T>** must be defined as described in Section **??**. Call **ReadFromGhostsEnd()** to finish updating the ghost values. One can overlap communication with computation by placing some code in between **ReadFromGhostsBegin()** and **ReadFromGhostsEnd()**. This function should not be used for elemental buffers. Use the functions **ReadFromGhostElemsBegin()** and **ReadFromGhostElemsEnd()** for elemental buffers.

3. Loop through the octants using one of the loops described in Section **??** and use the following function within the loop to get the indices for the nodes of the octants.

```
int ot::DA::getNodeIndices(unsigned int* nodes)
```

'nodes[i]' will store the index of the i-th node (0-based indexing) of the current element. If a node is hanging, the index of the corresponding node of the element's parent will be returned (Section **??**).

4. DENDRO provides the option to compress the mesh and save on memory at the expense of a small overhead of decompression while using the above function. This option can be turned off and this overhead can be

26

saved. The following function can be used to check whether the mesh is compressed or not. It returns the option passed to the constructor of 'ot::DA'.

```
bool ot::DA::isLUTcompressed()
```

5. If the mesh is compressed, then the function 'getNodeIndices()' must either be called within each iteration of the loop or never be called within that loop. If a call to this function is unnecessary for some iteration then the following function must be called instead.

```
void ot::DA::updateQuotientCounter()
```

6. It is common to query for the levels and locations (anchors) of octants within the loop. The following functions can be used for the same.

ot :: DA::getLevel(unsigned int i) returns the level of the octant in the modified octree that includes 'pseudo-octants' for boundary nodes. This octree has a maximum depth equal to 1 more than that of the input octree used to construct the finite element mesh. Hence, the value returned by this function will be 1 more than the true level of the octant in the input octree. (Section **??**).

ot :: DA::getCurrentOffset() returns the anchor of the current octant in an object of the class 'Point'. The coordinates of the anchor can be extracted using the following member functions of the class Point:

```
int Point::xint()
int Point::yint()
int Point::zint()
```

7. To write values into the nodes owned by other processors use the following pair of functions. This is a 2 step proceedure and one can place some code in between these 2 function calls to overlap communication with computation.

```
template <typename T>
 int ot::DA::WriteToGhostsBegin(T* arr, unsigned int dof = 1)
```

The values are added to the existing values. The 'assignment by sum' operator ($+=$) must be defined for the type 'T'. The **class par::Mpi_datatype<T>** must be defined as described in Section **??**.

Call **WriteToGhostsEnd()** to finish updating the ghost values. One can overlap communication with computation by placing some code in between **WriteToGhostsBegin()** and **WriteToGhostsEnd()**. Do not use this function for elemental buffers. Use **WriteToGhostElemsBegin()** and **WriteToGhostElemsEnd()** for elemental buffers.

8. Update the vectors with the values from the buffers and release the memory for the buffers using the following functions.

   (a) For PETSc vectors use:

   ```
   int ot::DA::vecRestoreBuffer(Vec v, PetscScalar*& arr,
    bool isElemental, bool isGhosted,
    bool isReadOnly, unsigned int dof = 1)
   ```

   (b) For distributed STL vector use:

   ```
   int ot::DA::vecRestoreBuffer(Vec v, PetscScalar*& arr,
    bool isElemental, bool isGhosted,
    bool isReadOnly, unsigned int dof = 1)
   ```

The arguments for these functions must be the same as those used with **ot :: DA::vecGetBuffer**.

3.10 Assembling the finite element matrices

As mentioned earlier (Section **??**), one should not try to set the matrix entries directly using the PETSc functions. In DENDRO the element-to-node mappings are computed and stored in a local numbering scheme, which is an interleaved representation where a common index is used for both the elements and the nodes. In order to be able to set entries into the matrix we must first map the numbering scheme used within DENDRO to the global numbering scheme used within PETSc. This can be done using the following function.

```
int ot::DA::computeLocalToGlobalMappings()
```

One might want to set matrix entries for different matrices using the same ot::DA object. It is not necessary to compute the above described mapping for each matrix separately. Once the mapping is created, it is stored within the **class ot :: DA**. To check whether the mapping has already been computed or not for a particular ot::DA object, use the function **ot :: DA::computedLocalToGlobal()**.

Use the following function to set the matrix entries.

```
int ot::DA::setValuesInMatrix(Mat mat, std::vector<ot::MatRecord>& records,
unsigned int dof, InsertMode mode)
```

This function is simply a wrapper for the PETSc function **MatSetValues**. Call PETSc's **MatAssembly()** routines after setting all the values.

29

Examples of matvecs for matrix-free methods and functions to create full matrices can be found in 'dendro/examples/odaJac.C', 'dendro/examples/omgJac.C' and 'dendro/examples/elasticityJac.C'.

Chapter 4

Using the octree based

multigrid solver

Follow the following steps to use PETSc's multigrid solver through the DEN-
DRO framework. DENDRO's ot::DAMG object is similar to PETSc's DMMG
object. Refer to the PETSc manual for a description of the DMMG object.

1. Call the function **ot :: DAMG_Initialize()** at the beginning of your
   main program, just after calling the function **PetscInitialize ()**.

2. Create the multigrid object, coarser octrees, and the finite element
   meshes for all the levels by calling the following function. The inter-
   grid transfer operators between the levels will also be created within
   the function.

   ```
   PetscErrorCode  ot :: DAMGCreateAndSetDA ()
   ```

3. If you want to set a different user context for each level, loop though the
   multigrid levels and set the respective pointers in the member variable

'user'.

4. Set the following global function pointer. It will be called while setting up the solver for the coarsest grid, provided not all the available processors are active on the coarsest grid.

```
void (∗getPrivateMatricesForKSP_Shell)()
```

5. If the matrix used for constructing the preconditioner is different from the finite element matrix, the finite element matrix for all the levels must be set separately by calling the following function. This is not the typical case and so this step can be skipped in most cases.

```
PetscErrorCode ot :: DAMGCreateJMatrix()
```

6. Set up the preconditioning matrices for all the levels by calling the following function. If the previous step was skipped for any level, then it will be assumed that the finite element matrices and the preconditioning matrices are the same for that level.

```
PetscErrorCode ot :: DAMGSetKSP()
```

7. Solve the equations by calling the following function.

```
PetscErrorCode ot :: DAMGSolve()
```

8. The solution will be stored in the 'Vec x' object at the finest level. This can be obtained using the macro 'DAMGGetx'.

9. Destroy the multigrid object by calling the following function.

```
PetscErrorCode ot :: DAMGDestroy()
```

32

10. Call the function **ot :: DAMG_Finalize()** at the end of your main program, just before calling the function **PetscFinalize ()**.

4.1 Examples using the octree based multi-grid solver

We provide several examples illustrating the usage of the multigrid solver described above. They can be found in the files `dendro/examples/omgNeumann_ex1.C`, `dendro/examples/omgNeumann_ex2.C`, `dendro/examples/omgNeumann_ex3.C`, `dendro/examples/omgNeumann_2spheres.C`. The first three examples are similar, so we will describe here only `dendro/examples/omgNeumann_ex3.C` and `dendro/examples/omgNeumann_2spheres.C`.

`dendro/examples/omgNeumann_ex3.C` approximately solves the scalar PDE

$$-\nabla \cdot (\kappa \nabla u) + \alpha u = f$$

in the 3D unit cube subject to homogeneous Neumann boundary conditions. The coefficients and right hand side are

$$
\begin{aligned}
\kappa &= 1 + x^2 + y^2 + z^2 \\
\alpha &= x^2 + y^2 + z^2 \\
f &= (x^2 + y^2 + z^2 + 3\omega^2\pi^2(1 + x^2 + y^2 + z^2))\cos(\omega\pi x)\cos(\omega\pi y)\cos(\omega\pi z) \\
&\quad + 2x\sin(\omega\pi x)\cos(\omega\pi y)\cos(\omega\pi z) + 2y\cos(\omega\pi x)\sin(\omega\pi y)\cos(\omega\pi z) \\
&\quad + 2z\cos(\omega\pi x)\cos(\omega\pi y)\sin(\omega\pi z)
\end{aligned}
$$

thus the analytic solution is

$$u = \cos(\omega \pi x) \cos(\omega \pi y) \cos(\omega \pi z).$$

The example executable is launched using the command line

```
mpirun -np N omgNeumann_ex3 <filePrefix>
```

where the `<filePrefix>` is the common prefix of the files that contain points to be read by different MPI processes. The `options` file (which contains different PETSc options) must be present in current directory. An example `options` file can be found in `dendro/bin/`. The `*.inp` files must also be present in current directory, as discussed in Chapter **??**.

To be specific, the files with the points should have the names of the form `<filePrefix>X_N.pts` where X is $0, \ldots, $N-1. The file `<filePrefix>X_N.pts` is read by the process with MPI rank X. The files with the points are binary files and their format is as follows. The first several bytes (binary representation of an "unsigned int" C type for your architecture, typically 4 bytes) contain the number of points. The rest of the file contains the coordinates of the points. Each coordinate is stored using the binary representation of the C type "double" on your architecture (typically 8 bytes). The coordinates are stored in the order $x_1, y_1, z_1, x_2, y_2, z_2, \ldots$

There is a useful utility `dendro/scripts/splitPoints.C` which splits the points for use by several MPI processes. This utility is invoked as follows:

```
splitPoints file numProcs outFilePrefix
```

where `file` is the name of the file to split, `numProcs` is number of MPI pro-

cesses, and `outFilePrefix` is the common prefix of the files to be produced. The utility produces `numProcs` files, each file contains approximately equal part of the points, the filenames are of the form `<outFilePrefix>X_N.pts` described above.

The function **main** in the file `dendro/examples/omgNeumann_ex3.C` calls the function **solve_neumann** in the file `dendro/examples/omgNeumann.C` to perform most of the tasks. Please see doxygen documentation for the complete description of the function **solve_neumann**. In particular, **solve_neumann** requires the caller to provide two callback functions (function pointers). These callback functions are called by **solve_neumann** to set the PDE coefficients and the right hand side. That is, the callback functions are given a list of the centers of all octants, and must return the values of κ, α and f at the center of each octant. The functions **CalcVarCoef** and **CalcVarRHS** in the file `dendro/examples/omgNeumann_ex3.C` are examples of such callback functions.

Another parameter to the function **solve_neumann** is the sequence of points. The octree is built using the condition that each octant should contain at most one point.

After calling **solve_neumann** to obtain an approximate solution, the **main** function in `dendro/examples/omgNeumann_ex3.C` loops over all the octants to find the maximum difference between the analytic and the numerical solutions.

A different example can be found in the file `dendro/examples/omgNeumann_2spheres.C`. This example calculates an approximate solution to the Poisson equation

$-\Delta u = f$ in the "shell" domain

$$p = (x, y, z) \ : \ 0.2 < ||p - c|| < 0.4,$$

where $c = (0.5, 0.5, 0.5)$. The boundary conditions at the spheres $||p - c|| = 0.2$ and $||p - c|| = 0.4$ are zero Dirichlet. The right hand side is chosen to be

$$f = \frac{\pi^2}{(0.4 - 0.2)^2} \frac{\sin(\frac{\pi(r-0.2)}{0.4-0.2})}{r},$$

where $r = ||p - c||$. This gives rise to the analytic solution

$$u = \frac{\sin(\frac{\pi(r-0.2)}{0.4-0.2})}{r}.$$

The example executable is launched using the command line

```
mpirun -np N omgNeumann_2spheres <filePrefix>
```

where the `<filePrefix>` is the common prefix of the files that contain octants to be read by different MPI processes. Note that unlike other examples, this one uses octants and not points as input. As with other examples, the `options` file and the `*.inp` files must be present in current directory.

To be specific, the files with the octants should have the names of the form `<filePrefix>X_N.ot` where X is 0,...,N-1. The file `<filePrefix>X_N.ot` is read by the process with MPI rank X. The files with the octants are text files, and their format is as follows. The first line contains the dimension of a problem (3 in our case) and the maximum level ("depth") of the octants that follow. The numbers on this and all subsequent lines are separated by spaces.

Next line contains total number of the octants in the file. All the following lines correspond to the octants, one line per octant. Every line corresponding to an octant contains 4 numbers. The first three are the integer coordinates of the octant's anchor (based on the maximal depth specified in the beginning of the file), and the fourth one is the octant's level.

The input octants are expected to accurately resolve the geometry of the shell domain. There is an utility `dendro/scripts/gen2spheres.C` which produces such a set of octants. The utility is launched using the command line

```
gen2spheres <fileName> <maxDepth> <regularDepth> [<innerRaduis>]
[<outerRadius>]
```

where `<fileName>` is the output file name, `<maxDepth>` is desired level of the octants which intersect the sphere and `<regularDepth>` is the desired level of the octants in the shell area between the two spheres. The last two arguments, `<innerRaduis` and `<outerRadius>` are the radii of the two spheres. These two parameters are optional, their default values are 0.2 and 0.4, respectively.

To be more specific, `dendro/scripts/gen2spheres.C` produces the set of octants using the following algorithm. We start with subdividing the unit cube into eight octants. Each obtained octant is examined. If the octant intersects with any of the two spheres, and the level of the octant is less than `<maxDepth>`, we subdivide the octant further, i.e., we apply this algorithm recursively. If the octant intersects with any of the two spheres, but is already at the level `<maxDepth>`, we add it to the output. If the

octant lies between the two spheres, it is either subdivided (if its level is less than `<regularDepth>`) or added to the output (if its level equals `<regularDepth>`). Otherwise the octant is discarded.

The obtained file with the octants needs then to be split into several files, so each MPI process loads its own file. This is done using the python script `dendro/scripts/splitOct.py`:

```
splitOct.py  <octFile> <N> <splitPrefix>
```

where `<octFile>` is the name of the file to split, `<N>` is number of MPI processes, and `<splitPrefix>` is the common prefix of the files to be produced. The script produces `<N>` files, each file contains approximately equal part of octants, the filenames are of the form `<splitPrefix>X_N.ot` described above.

The function **main** in the file `dendro/examples/omgNeumann_2spheres.C` calls the function **solve_neumann_oct** in the file `dendro/examples/omgNeumann.C` to perform most of the tasks. Please see doxygen documentation for the complete description of the function **solve_neumann_oct**. Just like **solve_neumann**, **solve_neumann_oct** requires the caller to provide two callback functions to set the PDE coefficients and the right hand side.

In order to simulate the homogeneous Dirichlet boundary conditions at the two spheres, we set the adsorption coefficient α to some large value (say, 10^8) for each octant which center lies outside our shell domain. We also set the right hand side to be zero on all such octants.

The function **solve_neumann_oct** requires the sequence of octants as one of the input parameters. The octree is then built by completing the

provided sequence of octants.

After calling **solve_neumann_oct** to obtain an approximate solution, the **main** function in the file `dendro/examples/omgNeumann_2spheres.C` loops over all the octants to find the maximum difference between the analytic and the numerical solutions.