OP2 Developers Guide - Distributed Memory (MPI) Parallelisation

Gihan R. Mudalige and Mike Giles

June 29, 2011

Abstract

This document explains OP2's distributed memory parallelisation design and implementation based on MPI. It is intended primarily for those who are developing OP2 for multi-CPU and/or multi-GPU clusters and should be read in conjunction with the OP2 developer manual for single node systems. Those who are only using OP2 should instead read the Users Manual.

Contents

1 Introduction

The OP2 design uses hierarchical parallelism with two principal levels. At the highest level, OP2 is parallelised across distributed-memory clusters using MPI message-passing. This uses essentially the same implementation approach as the original OPlus. The domain is partitioned among the compute nodes of the cluster, and import/export halos are constructed for message-passing. Data conflicts when incrementing indirectly referenced datasets are avoided by using an "owner-compute" model, in which each process performs the computations which are required to update data owned by that partition. The second level of parallelisation is achieved within a single multi-core CPU or GPU node. The multi-CPU parallelisation is currently supported by OpenMP threads and in the future will support other techniques such as Intel's AVX. The GPU support is based on NIVIDIA CUDA and will later support OpenCL. The single node design and implementation is the subject of the OP2 developer manual. In this document we detail the design of the distributed memory level based on MPI and describe some of it's key implementation aspects. The Airfoil example application supplied with the OP2 release is used as an example to illustrate the design and implementation.

2 MPI parallelisation strategy

2.1 Parallel Startup

An OP2 application executed under MPI on a cluster of nodes, where a node may consist of a single CPU core, a multi-core CPU (or an SMP node) or a GPU node, will have multiple copies of the same application program executed as separate MPI processes. The starting point of a distributed memory parallel application is the design of how the sets, mappings and data on sets that defines an unstructured mesh application is read in by OP2. The current proposal is to achieve this input (and output) via two approaches:

1. Leave the application developer to handle the file I/O where a minor extension to the OP2 API will allow to define `op_set`s, `op_dat`s and `op_map`s that are distributed across the MPI universe.

2. Provide HDF5 based parallel I/O routines with which an application programmer can read in the sets, data on sets and mappings from a file.

The rationale for the above is to allow developers to make the trade-off between ease-of-use and flexibility. Some will want maximum ease-of-use and are prepared to pay the price of working with HDF5 files with the flat keyword-based hierarchy which we will assume. Others will want the flexibility to manage their data storage

in the way they wish, and will accept the additional programming effort this will entail.

In the first case, we assume that the user I/O has resulted in loading the data on sets and mappings between sets across the distributed memory MPI universe. The number of set elements (and thus data on sets) or the size of the mapping tables held by an MPI process is decided by the application programmer. OP2 assumes that only one partition is held by a single MPI process. For example given P number of processors, g_nnodes number of nodes and g_nedges an application programmer can decide to distribute the nodes and edges where each process holds g_nnodes/P nodes and g_nedges/P. Similarly the edge to node mapping table could be distributed such that process 0 will provide the first g_nedges/P entries, process 1 the second g_nedges/P entries and so on. When distributing mapping table entries we assume that the MPI process that holds some set element X will also hold the mapping table entries (belonging to all the mapping tables) from X. This is effectively a trivial contiguous block partitioning of the data on sets and mappings, but it is important to note that this distribution (or partitioning) will not be used for the parallel computation. OP2 will repartition the data on sets and related mapping tables, migrate all data on sets/mappings to the correct MPI process and renumber the mapping tables as needed. The current MPI implementation provides partitioning routines (as described in Section 3) to support this task.

After the loading in of data and mapping tables is complete OP2 set, map and dat declarations can be invoked on each process. This extends the existing API as follows:

- **op_decl_set**: `size` is the number of elements of the set which will be provided by this MPI process

- **op_decl_map**: `imap` provides the part of the mapping table which corresponds to its share of the `from` set

- **op_decl_dat**: `dat` provides the data which corresponds to its share of `set`

The current implementation of Airfoil over MPI (`airfoil_mpi`) uses the above method to do an initial distribution of data on sets and mapping tables. MPI rank 0 will serially read in to its RAM the data on sets and mapping tables from `new_grid.dat` and then will distribute the part of data and mappings (using MPI_Scatter operations) to other processors.

In the second case, OP2 will define an HDF5 file format using which an applications programmer can create a file containing data and mappings to be used in the OP2 application. The OP2 API will define the following to support reading from such a file:

- **op_decl_set_hdf5**: similar to **op_decl_set** but with `size` replaced by `file` which defines the HDF5 file from which `size` is read using keyword `name`

- **op_decl_map_hdf5**: similar to **op_decl_map** but with `imap` replaced by `file` from which the mapping table is read using keyword `name`

- **op_decl_dat_hdf5**: similar to **op_decl_dat** but with `dat` replaced by `file` from which the data is read using keyword `name`

2.2 Constructing Halo Lists

The OP2 distributed memory parallelisation uses an "owner-compute" model where each MPI process "owns" the elements of the partitioned sets. In order to ensure that the data associated with these sets are "up-to-date" it is necessary to communicate with "neighbours" of an MPI process, and perform redundant computation on some of the elements imported from these neighbours. The block of data that's exchanged is commonly known as a halo in distributed memory programming.

Consider an example mesh consisting of nodes and cells, with a cell to node mapping. If a cell is located on a MPI process, then all the nodes making up the cell must also be present in this (local) process in order to ensure that when a loop over cells are performed, the owned cell receives all the possible contributions from its nodes. If at least one of the nodes are not present in this local process, then it should be imported in from a foreign MPI process. Conversely, if a node located on an MPI process is part of a cell that resides in a foreign MPI process, then that cell needs to be imported in to this local process for the local node to receive all the required contributions. As part of this, the imported cell may have to be kept "up-to-date" with redundant computation.

Given the above scenario, each MPI process needs to construct a list of elements for each set that needs to be imported from and exported to other "neighbouring" MPI processes. Within an OP2 application, creation of these halos occur immediately after partitioning with a call to `op_halo_create()`. The remainder of this section illustrates the design and implementation of this routine and the data structures used.

In order to determine what elements of a set should be imported or exported (via MPI Send/Receives) to or from another MPI process, we create the following classification:

- **OWNED** : An element of a set is said to be "owned" by the MPI process it is located at, if all the elements referenced through all the mapping tables from this set element is also owned by this MPI process. e.g. In a mesh with nodes and cells (with a mapping of cells to nodes) a cell held within an MPI process is owned by this MPI process if all the nodes referenced by this cell is also owned by this MPI process.

- **Export Execute Halo (EEH)**: An element of a set is said to belong to

4

the "export execute halo" if at least one element referenced through any of the mapping tables from this set element is NOT owned by this (local) MPI process. e.g. In a mesh with nodes and cells (with a mapping of cells to nodes), if a cell references a node owned by a foreign MPI process then this cell needs to be exported to the foreign MPI process. This cell will fall in to the Export Execute Halo (EEH) on the local MPI process and in turn will form part of the Import Execute Halo (IEH) on the foreign MPI process.

- **Import Execute Halo (IEH)**: If an element of a set is referenced by an element located at a foreign MPI process then the foreign element needs to be imported on to this MPI process in order to get the correct contributions to the local element. The imported element may require to be executed over in order to be kept "up-to-date" by that process. This results in redundant computation. The imported element is said to be in the Import Execute Halo (IEH) of the local MPI process. e.g. In a mesh with nodes and cells (with a mapping of cells to nodes), if a node on the local MPI process is referenced by a cell in a foreign MPI process, then the foreign cell needs to be imported and will be part of the Import Execute Halo (IEH) on the local MPI process.

- **Import Non-execute Halo (INH)**: If an element located at an MPI process references (via some mapping) an element that is located on a foreign MPI process, then the element on the foreign MPI process needs to be imported. The imported element will fall in to the Import Non-Execute Halo (INH) if it is not already a part of the Import Execute Halo (IEH). e.g. In a mesh with nodes and cells (with a mapping of cells to nodes), if a cell references a node owned by a foreign MPI process then the referenced node needs to be imported onto this MPI process. The node will fall into the Export Non-Execute Halo on the foreign MPI process.

- **Export Non-Execute Halo (ENH)**: If an element of a set is referenced by an element located at a foreign MPI process then the local element needs to be exported on the foreign MPI process (if its not already in the Export Execute Halo). Any loop over the foreign set element cannot proceed without getting all the contributions from the elements it refers to. The exported element is said to be part of the Export Non-Execute Halo (ENH) on the local MPI process. e.g. In a mesh with nodes and cells (with a mapping of cells to nodes) if a node located on the local MPI process is referenced by a cell in a foreign MPI process, then the local node needs to be exported on to that foreign process.

The above classification allows to clearly determine which elements of a set can be computed over without MPI communications, facilitating overlapping of computation with communications for higher performance (see Section 2.3). The

5

op_halo_create() routine (defined in op_mpi_core.c) goes through all the mapping tables and creates lists that hold the indices of the set elements that fall in to the above classification. An export or an import list for an op_set has the following structure (defined in op_mpi_core.h):

```
typedef struct {
 op_set set;         //set related to this list
 int    size;        //number of elements in this list
 int    *ranks;      //MPI ranks to be exported to or imported from
 int    ranks_size;  //number of MPI neighbors
                     //to be exported to or imported from
 int    *disps;      //displacements for the starting point of each
                     //rank's element list
 int    *sizes;      //number of elements exported to or imported
                     //from each ranks
 int    *list;       //the list of all elements
} set_halo_list_core;

typedef halo_list_core* halo_list;


halo_list *OP_export_exec_list;//EEH list
halo_list *OP_import_exec_list;//IEH list

halo_list *OP_import_nonexec_list;//INH list
halo_list *OP_export_nonexec_list;//ENH list
```

The above four arrays are indexed using set->index and is of size OP_set_index. Import and export list creation in op_halo_create() is accomplished in the following steps, by each MPI process:

1. **Create export lists for execute set elements**
   Each MPI process goes through each element of each set. If a set element references (via any of the mapping table from this set) any element that is not OWNED by the local MPI process then we add the referencing element to the EEH list. When creating the EEH list on a given (local) MPI process, we also keep track of the foreign MPI processes that it will be exported to. The list of elements to be sent to each foreign MPI process will be sorted according to its local index.

2. **Create import lists for execute set elements and related mapping table entries**
   Each MPI process exchanges the EEH list with the relevant neighbour processes and use the imported lists to construct the IEH.

6

3. **Exchange mapping table entries using the import/export lists**
The EEH and IEH on each MPI process can now be used to exchange the bits of the mapping tables that is related to the execute halo. The EEH and IEH of the "from set" of each mapping table is used to identify whch mapping table entries are to be exported and imported. For each mapping table, the imported mapping table entries will be appended to the end of the `op_map->map` array.

4. **Create import lists for non-execute set elements**
Each MPI process goes through each element of each set, (now using all the mapping table entries including the additional mapping table entries that was imported), and adds any other element referenced (but not in IEH) to a INH list for each set. The list of elements to be imported from each foreign MPI process will be sorted according to its local index on the foreign process.

5. **Create non-execute set export lists**
Each MPI process exchanges the INH list with the relevant neighbour processes and use the imported lists to construct the ENH. After this step, halo lists are complete. Each MPI process has EEH, ENH, IEH and INH lists.

6. **Exchange data defined on execute set elements using the set import/export lists**
The data defined on the elements belonging to each halo list is exchanged. The execute halos are exchanged first. For each `op_dat` the imported data will be appended to the end of the `op_dat->data` array.

7. **Exchange data defined on non-execute set elements using the set import/export lists**
The non-execute halos are exchanged second. For each `op_dat` the imported data will be appended to the end of the `op_dat->data` array after the IEH data.

8. **Renumber Mapping tables**
Each MPI process goes through all mapping table entries and renumbers the referenced set element indices to point to local indices. All required referenced elements should be now available locally on each MPI process.

9. **Create MPI send buffers**
For each `op_dat`, create some buffer space for `MPI_Isend`s. The following struct holds the required buffers and related data.

```
typedef struct {
 int        dat_index;    //index of the op_dat to which
                          //this buffer belongs
 char       *buf_exec;    //buffer holding exec halo
                          //to be exported;
 char       *buf_nonexec; //buffer holding nonexec halo
                          //to be exported;
 MPI_Request *s_req;      //array of MPI_Reqests for sends
 MPI_Request *r_req;      //array of MPI_Reqests for receives
 int        s_num_req;    //number of sends in flight
                          //at a given time for this op_dat
 int        r_num_req;    //number of receives awaiting
                          //at a given time for this op_dat
} op_mpi_buffer_core;

typedef op_mpi_buffer_core *op_mpi_buffer;

op_mpi_buffer *OP_mpi_buffer_list;
```

10. **Separate owned elements**
    To facilitate overlapping of computation with communication, for each set, the OWNED elements are separated to form a contiguous block of elements. Any element NOT belonging to the EEH is an OWNED element. We rearrange the local set elements and create an `int` array (`owned_num`) holding the index that marks the end of the OWNED elements block for each set. Thus during a loop over a given set, on each MPI process, element indices 0 to `owned_num[set->index]` can be computed over without halo data and elemets from `owned_num[set->index]` to `set->size` will need to be computed over after all the calls to `wait_all()` is completed.

11. **Save the original set element indices**
    As the set elements are now rearranged, we need to keep track of the original order in which they appeared so that calls to `op_fetch()` and `op_put()` as well as final outputs can be accurately handled. The `part` struct (see Section 3) is used to hold the original global indices of each set element.

12. **Clean up and compute rough estimate of average worst-case halo size**
    Temporary arrays are freed and a rough estimate of the average size of the worst case import halos on each MPI process is computed. This takes in to account both the IEH and INH and accounts for the data sizes held per set element. The calculation does NOT take in to account what halos are exchanged during the `op_par_loops` later in the application.

2.3 `op_par_loop` and Halo Exchanges

A call to `op_par_loop` in a OP2 application executed under MPI will result in the loop being executed over the local elements of the set on each MPI process. Additionally if the loop is an indirect loop, then computation should be done over the IEH as well. Depending on the loop (indirect or direct) and the access type of each `op_arg`, halo exchanges (with a call to `exchange_halo()`)may be needed before computation is performed over any elements that are not OWNED and the IEH. After loop computation is performed, depending on the access and argtype of the `op_arg` the halos must be marked as "dirty" so that the next iteration of the loop can make the decision to update the halos as required (defined in `op_mpi_core.c`). The current implementation maintains a separate global `int` array that holds this information. The value `dirtybit[op_dat->index]` is set to 1 to indicate that the halo of `op_dat` with index `op_dat->index` has been modified. The rules governing the loop operation is as follows:

1. If the `op_par_loop` consists of at least one `op_arg` that is indirectly accessed then the whole loop is classified as an indirect loop. Else it is a direct loop.

2. Direct loops will only need to loop over the local set size and no halo exchanges are needed.

3. For indirect loops the following algorithm determines a halo exchange.

   ```
   for each indirect op_arg {
     if (op_arg.acc is OP_READ or OP_RW) and (dirty bit is set)
     then do halo exchange for op_arg.dat and clear dirty bit
   }
   if(all indirect op_arg.acc == OP_READ)
       execute/loop over set size
   else
       execute/looop over set size + IEH
   ```

4. After the loop computation we set the dirty bit for each `op_arg.dat` with `op_arg.acc` equal to `OP_INC`, `OP_WRITE` or `OP_RW`.

A halo exchange is triggered by a call to `exchange_halo(op_set set, op_arg arg)` which is defined in `op_mpi_core.c`. Within this call, the above conditions that determines a halo exchange is checked and if satisfied will pack the relevant halo data to the pre defined send buffers, make a call to MPI non-blocking operations (`MPI_Isend` and `MPI_Irecev`) and will return 1 to indicate that a non-blocking communication is *in-flight*. As detailed in Section 2.2, the EEH and the ENH of an MPI process provides the indices of the elements that needs to be exported as well as the MPI ranks that will be exported to. Using these lists an MPI process will pack the data to be set into the send buffers and then will send them using `MPI_Isend` operations. The following is for sending the EEH:

```
halo_list exp_exec_list = OP_export_exec_list[dat->set->index];

for(int i=0; i<exp_exec_list->ranks_size; i++) {
   for(int j = 0; j < exp_exec_list->sizes[i]; j++)
   {
      int set_elem_index = exp_exec_list->
                           list[exp_exec_list->disps[i]+j];

      memcpy(&OP_mpi_buffer_list[dat->index]->
      buf_exec[exp_exec_list->disps[i]*dat->size+j*dat->size],
      (void *)&dat->data[dat->size*(set_elem_index)],dat->size);
   }
   MPI_Isend(&OP_mpi_buffer_list[dat->index]->
         buf_exec[exp_exec_list->disps[i]*dat->size],
         dat->size*exp_exec_list->sizes[i],
         MPI_CHAR, exp_exec_list->ranks[i],
         dat->index, OP_MPI_WORLD,
         &OP_mpi_buffer_list[dat->index]->
         s_req[OP_mpi_buffer_list[dat->index]->s_num_req++]);
}
```

The `MPI_Isend` operations are immediately followed by `MPI_Irecev` operations, which sets up the non-blocking communications to directly copy the incoming data in to the relevant op_dat, using the IEH and INH lists.

```
halo_list imp_exec_list = OP_import_exec_list[dat->set->index];

int init = dat->set->size*dat->size;
for(int i=0; i < imp_exec_list->ranks_size; i++) {
   MPI_Irecv(&(OP_dat_list[dat->index]->
         data[init+imp_exec_list->disps[i]*dat->size]),
         dat->size*imp_exec_list->sizes[i],
         MPI_CHAR, imp_exec_list->ranks[i],
         dat->index, OP_MPI_WORLD,
         &OP_mpi_buffer_list[dat->index]->
         r_req[OP_mpi_buffer_list[dat->index]->r_num_req++]);
}
```

A call to `wait_all(op_arg arg)` routine needs to be performed in order to complete the MPI communications. The `op_par_loop` is structured so that all the `exchange_halo()` calls are done at the beginning of the loop, followed by computation over the OWNED elements of the set and then by calls to `wait_all()`. This will allow for maximum overlapping of computation with communication as none of the OWNED elements reference any halo data. After the calls to the

`wait_all()` the remaining set elements could be computed. A reference implementation of the above can be found in `op_mpi_seq.h`.

2.4 Global Operations

If an `op_arg` is of type `OP_ARG_GBL` then a global operation needs to be performed for that argument. The operation to be performed is one of `OP_INC` (global reduction), `OP_MAX` (global maximum), `OP_MIN` (global minimum). For an `op_arg` of type `OP_ARG_GBL`, looping over the IEH should not be performed as the IEH is a redundant block of computation that should not be "double counted" during a global operation. Thus the reference implementation passes in a dummy value in place of any `op_arg` with type `OP_ARG_GBL`. After the loop over the elements are performed on each MPI process, the global operation should be done across all the MPI processes by a call to `global_reduce()` which is also defined in `op_mpi_core.c`. This routine checks for the type of the data exchanged and the type of the operation to be performed and calls `MPI_Reduce` with the relevant operation and data type.

2.5 `op_fetch_data`

The proposed operation of `op_fetch_data (op_dat dat)` within an OP2 application executing over MPI will be to present the current values of the `op_dat`'s data array in the order of the elements that was originally handed to OP2. It should be noted that the data array presented to the user level application is a copy of the current state of the internal `op_dat`. This routine is currently not implemented. A valid implementation will first make a copy of the current data values in the `op_dat` requested and will reorder them according to the original global index of the set elements on which this data is defined on.

Conversely an `op_put_data()` routine may also be implemented (as required) so that the user level application can modify the internal values of an `op_dat`. In this case the user submitted data values will replace the internal `op_dat`'s data values. A valid implementation will need to translate the original set element index to the current set element index.

2.6 Debug Routines

Currently only one debug routine is used, `reset_halo(op_arg arg)`, which initialise import halo data to NaN - for diagnostics purposes. If `OP_diags` is > 2 then this routine is called to reset a halo for each `op_arg`.

2.7   Performance Measurements

The time spent in the `op_par_loop()` calls are measured and accumulated. The setup costs due to halo creation and partitioning is also measured and the maximum on all the processors is printed to standard out by rank 0. Additionally information about the amount of MPI communications performed is also collected. For each `op_par_loop()` we maintain a struct that holds (1) the accumulated time spent in the loop (2) the number of time the `op_par_loop()` routine is called, (3) the indices of the `op_dat`s that requires halo exchanges during the loop, (4) the total number of times halo exchanges are done for each `op_dat` and (5) the total number of bytes exported for each `op_dat`.

```
typedef struct
{
  char const  *name;    // name of kernel
  double      time;     //total time spent in this
                        //kernel (compute+comm-overlapping)
  int         count;    //number of times this kernel is called
  int*        op_dat_indices;  //array to hold op_dat index of
                               //each op_dat used in MPI halo
                               //exports for this kernel
  int         num_indices; //number of op_dat indices
  int*        tot_count;   //total number of times this op_dat was
                           //halo exported within this kernel
  int*        tot_bytes;   //total number of bytes halo exported
                           //for this op_dat in this kernel
} op_mpi_kernel;
```

Currently, the only way to identify a loop is by its name. Thus we use a simple hash function on the name string to index into a hash table (`op_mpi_kernel_tab[]`) that holds an `op_mpi_kernel` struct for each loop. Monitoring the halo exchanges require calls to the `op_mpi_perf_comm()` (defined in `op_mpi_core.c`) for each `op_arg` that has had a halo exchanged during each call to an `op_par_loop()`. As this may cause some performance degradation, we allow the MPI message monitoring to be enabled at compile time using the `-DCOMM_PERF` switch.

2.8   Output Routines

A number of output routines are provided currently, including routines to output the performance measures, `op_mpi_timing_output()` as well as file writes that directly writes an `op_dat` to an ASCI file, `gatherprint_tofile()` or a binary file, `gatherprint_bin_tofile()`. These routines gathers a specified `op_dat`, (which is distributed across the MPI universe) on to MPI rank 0 and prints the results

to a user specified file. The `op_dat` will be written to file in the same order in which the set elements were handed to OP2 during the initial input of data and mapping tables. This requires calls to `op_fetch()`.

2.9   Garbage Collection

At the end of the OP2 application a call to `op_halo_destroy()` will free all halo lists, MPI send buffers and the table holding performance measures.

3 Partitioning

Given the unstructured mesh in an OP2 application, distributing the data on sets and mapping tables across the MPI universe should be done by utilising a mesh partitioner in order to avoid building large halos. The OP2 proposal is to achieve good partitions without the intervention of the application programmer. The idea is that once the OP2 declaration routines are called the MPI back-end should call a partitioning routine to partition the sets and maps and migrate the data on sets to new MPI processes as required. There are a number of grid/mesh partitioners that can be used, and at the moment it is not clear which one will provide the best performance. The current reference implementation uses the geometric partitioning routine in ParMetis [1] and provides a number of partitioning and data migration routines to facilitate the above goals. However, a number of extensions needs to be implemented to obtain high quality partitions as well as to make the partitioning truly seamless with no user intervention at the application program level.

The OP2 proposal is to partition the mesh immediately after all the calls to `op_decl_*`. Thus we assume that an initial parallel distribution of the sets and mapping tables have been performed during input, either by user defined I/O routines or using the HDF5 parallel I/O routines. For example in the `airfoil_mpi` application the data and mappings are distributed in a block partitioning fashion.

The OP2 development will in the future provide the application programmer with a choice of third party partitioning routines. At the moment only the geometric partitioning routine in ParMetis (`ParMETIS_V3_PartGeom()`) is available/used. A call to the wrapper function `op_partition_geom(op_dat coords, int g_nnode)` defined in `op_mpi_part_core.c` will utilise this routine to partition an op_dat which contains the xyz coordinates of some set. A wrapper function is required to format the mesh/gird such that it is in a format acceptable to the partitioning routine. In the airfoil application, the xy coordinates of the nodes are supplied in `p_x`. We anticipate that supporting different partitioners may require other wrapper functions to be developed into the MPI back-end. A further proposal is to provide decision logic that selects the appropriate partitioner routine (by calling the appropriate wrapper function) depending on the available `op_set`s, `op_map`s and `op_dat`s without the intervention of the application programmer.

Considering the airfoil application, after a call to `op_partition_geom()`, on each MPI process, the ParMetis routine returns an array that gives the new MPI rank of each set element (in this case for each node). At this point of the application we consider nodes as the <u>partitioned</u> (primary) set. The primary set and the available mapping tables will now allow to partition all other sets. These secondary sets will *inherit* the primary set's partitioning. Partitioning secondary sets is achieved by a call to `partition_all()` from within a wrapper function. Next, a call to `migrate_all()` migrate the data and mappings to the new MPI process and will sort the elements on the new MPI ranks. Finally `renumber_maps()` will

renumber mapping table entries with new indices.

At the end of an OP2 application, the partitioning needs to be reversed in order to revert back to the original set element order in which the user application supplied the data on sets and mapping tables to OP2. This is achieved in `op_partition_reverse()` where the original partition information (which was saved during the partition creation and halo creation routines) are used to reverse the mapping table renumbering and migrate the data on sets to their original MPI rank and sort them in the original element index order.

4   To do list

- Design/implement OP2 to use CUDA/OpenMP within a node on top of the current MPI implementation

- Implement wrapper functions to use other ParMetis routines (especially `ParMETIS_V3_PartKway()`)

- Provide decision logic that selects the appropriate partitioner routine depending on the available `op_set`s, `op_map`s and `op_dat`s without the intervention of the application programmer.

- Implement wrapper functions to use other partitioners (PTScotch [2]). Investigate performance of each package.

- Design and implement HDF5 file IO - define input file format

- Implement automatic check-pointing

References

[1] ParMETIS user manual,
http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/manual.pdf

[2] Scotch and PTScotch,
http://www.labri.fr/perso/pelegrin/scotch/