OP2 Developers Guide - Distributed Memory (MPI) Parallelisation

Mike Giles, Gihan R. Mudalige and Istvan Reguly

December 2013

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 distributed memory multi-core CPU and/or 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[5]. 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 implementations 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 its key implementation aspects. We also detail the heterogeneous cluster back-end design which facilitates the development and execution of an OP2 application on a cluster of GPUs and a cluster of multi-threaded CPUs. The Airfoil 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 implementation achieve this input (and output) via two approaches:

1. Allow the application developer to handle the file I/O where a minor extension to the OP2 API will makes it possible 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 OP2 routines can read in the sets, data on sets and mappings from a file in a prescribed format.

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. 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 so that 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 and mappings to the correct MPI process and renumber the mapping tables as needed. The current MPI implementation provides partitioning routines (as described in Section 4) 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`

An example implementation of the above is given in the Airfoil application (`airfoil_plain`) where an initial distribution of data on sets and mapping tables are achieved. MPI rank 0 will serially read into its RAM the data on sets and mapping tables from the mesh file (`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 defines an HDF5 file format (described later in Section 3) using which an applications programmer can create a file containing data and mappings to be used in the OP2 application. The OP2 API 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`

An example use of HDF5 file I/O is also given in the Airfoil application (`airfoil_hdf5`).

If the user is responsible for allocating data arrays to pass to `op_decl_map` and `op_decl_dat` then the MPI back-end will make a copy of this data internally as halo creation needs to realloc memory (which might invalidate original pointers held at the user application level). At the end of the program the user is responsible for freeing the allocated memory at the application level, and OP2 will free its internal copy of this data when `op_exit()` is called.

However, if OP2's hdf5 capabilities are used for File I/O then OP2 will be responsible for clean-up of data arrays at the end of the program.

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 because it may need to be executed for the local node to receive all the required contributions.

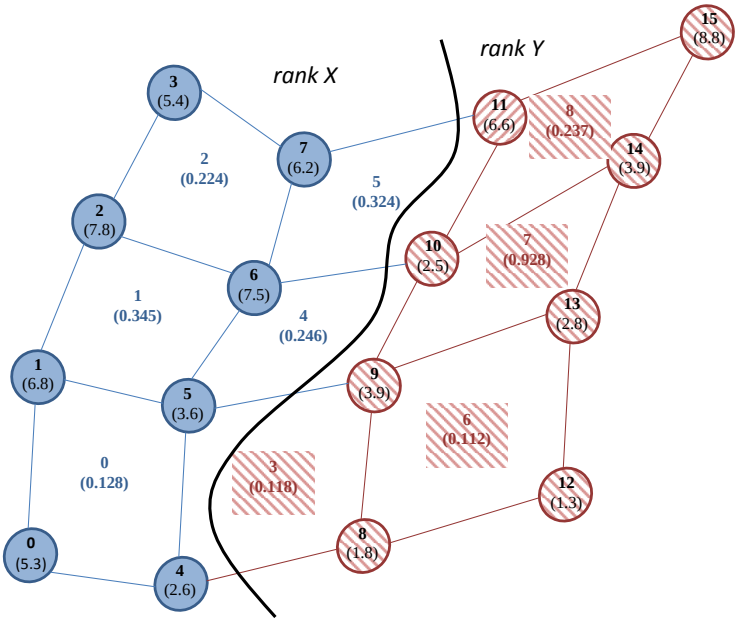

Figure 1: Example mesh with cells and nodes

In the example mesh illustrated in Figure 1 there are 16 nodes and 9 cells partitioned across two MPI processes (rank X and rank Y). Assume that the only mapping available is a cells to node mapping. Rank X holds nodes 0, 1, 2, 3, 4, 5, 6 and 7 and cells 0, 1, 2, 4, and 5. Rank Y holds nodes 8, 9, 10, 11, 12, 13, 14, and 15 and cells 3, 6, 7 and 8. A loop over the cells will need data on nodes 9, 10 and 11 to be imported in to rank X from rank Y. Additionally data on nodes 4 and 5 needs to be imported in to rank Y from rank X. On the other hand, a loop over nodes with contributions from surrounding cells will cause cells 4 and 5 to be imported into rank Y and "kept up to date" in order to receive their contributions to nodes 9, 10 and 11. 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 (see Section 4) 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.

Table 1: Import/Export lists

On X	*core*	*ieh*	*eeh*	*inh*	*enh*
Nodes	0, 1, 2, 3, 4, 5, 6, 7	-	-	8, 9, 10, 11	4, 5, 6, 7
Cells	0, 1, 2	3	4, 5		
On Y	*core*	*ieh*	*eeh*	*inh*	enh
Nodes	8, 9, 10, 11, 12, 13, 14, 15	-	-	4, 5, 6, 7	8, 9, 10, 11
Cells	6, 7, 8	4, 5	3	-	-

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:

- ***core*** : An element of a set is said to be a *core* element to the MPI process it is located at, if all the elements referenced through all the mapping tables from this set element is also in the *core* element set in 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 *core* to this MPI process if all the nodes referenced by this cell is also *core* to this MPI process.

- **export execute halo (*eeh*)**: An element of a set is said to belong to the "export execute halo" if at least one element referenced through any of the mapping tables from this set element is NOT core to 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, because it may need to be executed on that foreign process to update data on that node. 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 compute the correct contributions to the local element. 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, including mappings belonging to the *ieh*) 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 *enh* on the foreign MPI process.

- **export non-execute halo (*enh*)**: If an element of a set is referenced by an element located on a foreign MPI process then the data for the local element needs to be exported on the foreign MPI process (if its not already in the *eeh*). 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. *enh* is a subset of *core*. 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 to that foreign process.

The above classification allows us 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). For the mesh given in Figure 1, the import/export elements can be separated as in Table 1.

The `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 each of the above categories. An export or an import list for an `op_set` has the structure in Figure 2 (defined in `op_mpi_core.h` and `op_mpi_core.c`):

```
1  typedef struct {
2    op_set set;     //set related to this list
3    int size;       //number of elements in this list
4    int *ranks;     //MPI ranks to be exported to or imported from
5    int ranks_size; //number of MPI neighbors to be exported to or imported from
6    int *disps;     //displacements for the starting point of each rank's
7                    //element list
8    int    *sizes;  //number of elements exported to or imported from each ranks
9    int    *list;   //the list of all elements
10 } halo_list_core;
11
12 typedef halo_list_core * halo_list;
13
14 halo_list *OP_export_exec_list;//eeh list
15 halo_list *OP_import_exec_list;//ieh list
16
17 halo_list *OP_import_nonexec_list;//inh list
18 halo_list *OP_export_nonexec_list;//enh list
```

Figure 2: `halo_list_core` struct

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 *core* to 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*.

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

```
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;
```

Figure 3: `op_mpi_buffer` struct

tables that are related to the execute halo. The *eeh* and *ieh* of the "from set" of each mapping table is used to identify which 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 were 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 uses 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 or 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

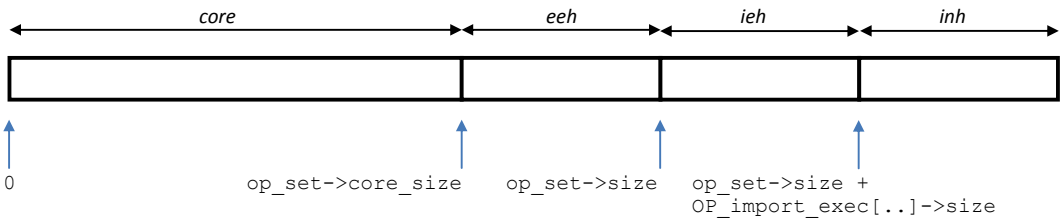

Figure 4: Element order of an `op_set` after halo creation

element indices to point to local indices. All required referenced elements (or a copy of it) should be now available locally on each MPI process.

9. **Create MPI send buffers**
   For each `op_dat`, create buffer space for `MPI_Isend`s. The struct detailed in Figure 3 holds the required buffers and related data.

10. **Separate *core* elements**
    To facilitate overlapping of computation with communication, for each set, the *core* elements are separated to form a contiguous block of elements. Any element NOT belonging to the *eeh* is a *core* element. We rearrange the local set elements and initialise `set->core_size` to the number of core elements. Thus during a loop over a given set, on each MPI process, element indices `0` to `set->core_size - 1` can be computed over without halo data and elements from `set->core_size` to `set->size + OP_import_exec[set->index]->size` will need to be computed over after all the calls to `wait_all()` are 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_data()` as well as final outputs can be accurately handled. The `part` struct (see `op_mpi_core.h`) 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 which halos are exchanged during the `op_par_loops` later in the application.

Figure 4 illustrates the element order in which data on a set will be organized after halo creation.

2.3 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 may be needed before computation is performed over any elements that are not `core` and in 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` and `op_mpi_rt_support.c`). The current implementation utilise a separate field in the `op_dat` struct to holds this information. The value `op_dat->dirtybit` is set to 1 to indicate that the halo of `op_dat` has been modified. The rules governing the loop operation are as follows:

```
for each indirect op_arg {
  if ((op_arg.access 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.access == OP_READ)
  execute/loop over set size
else
  execute/loop over set size + ieh
```

Figure 5: Algorithm to determine a halo exchange

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 using local data and no halo exchanges are needed.

3. For indirect loops the algorithm detailed in Figure 5 determines a halo exchange.

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

A halo exchange is triggered by a call to `op_exchange_halo(op_arg* arg)` which is defined in `op_mpi_rt_support.c`. Within this call, the above conditions that determines a halo exchange are 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 code detailed in Figure 6 is for sending the *eeh*.

The `MPI_Isend` operations are immediately followed by `MPI_Irecev` operations (Figure 7), 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.

A call to `op_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 `op_exchange_halo()` calls are done at the beginning of the loop, followed by computation over the *core* elements of the set and then by calls to `op_wait_all()`. This will allow for maximum overlapping of computation with communication as none of the *core* elements reference any halo data. After the calls to the `op_wait_all()` the remaining set elements could be computed. A reference implementation of the above can be found in `op_seq.h`.

2.4 Partial Halo Exchange

The halo exchange for a given `op_set` will trigger a exchange of all the halo elements for this set. The reason is due to OP2 creating the halo for an `op_set` based on all the mapping tables from and to that set (as detailed previously). Therefore when a parallel loop is executed over a boundary set

```
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++)
  {
    set_elem_index = exp_exec_list->list[exp_exec_list->disps[i]+j];
    memcpy(&((op_mpi_buffer)(dat->mpi_buffer))->
    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)(dat->mpi_buffer))->
       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)(dat->mpi_buffer))->
       s_req[((op_mpi_buffer)(dat->mpi_buffer))->s_num_req++]);
}
```

Figure 6: MPI send halo

```
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(&(dat->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)(dat->mpi_buffer))->
       r_req[((op_mpi_buffer)(dat->mpi_buffer))->r_num_req++]);
}
```

Figure 7: MPI receive halo

that has a very sparse connectivity to an internal set, the full internal set's halo will be exchanged. In some 3D mesh applications the connectivity from boundary set to internal set is significantly smaller, pointing to a case where a partial halo exchange would be advantageous to gain better performance. With a partial halo exchange you only need to hide message latency rather than latency plus time to actually transfer the full halo of data.

As a result, a partial halo exchange mechanism has now been implemented, where based on the mapping table that determines the connectivity between the sets, only the halo elements related to this map is exchanged. The same halo structs are used as before, but now a per map halo is created in `op_halo_permap_create()`. This per map halo is exchanged if the total number of (global) mapping table entries (for this map) that references foreign elements over a partition boundary is less than 30% of the total size of the halo for the exchanged set.

Thus for example, assume that a map exists from boundary edges to internal nodes (`bedges_to_nodes`) and a mapping exists between internal edges to internal nodes (`edges_to_nodes`). If

the number of halo elements due to bedges_to_nodes is less than 30% of the halo elements due to both bedges_to_nodes and edges_to_nodes, then a partial halo will be exchanged only using the per map halo of bedges_to_nodes, in a loop over bedges that indirectly accesses internal nodes.

2.5 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, the contributions from executing the *ieh* must not be included. 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 op_mpi_reduce(). This routine checks for the type of the data exchanged and the type of the operation to be performed and calls MPI_Allreduce with the relevant operation and data type.

2.6 Fetching Data

The operation of op_fetch_data() 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. The implementation first makes 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. A similar operation is carried out by op_fetch_data_hdf5(op_dat dat, T* data, int low, int high) and op_fetch_data_hdf5_file(op_dat dat, char const *file_name). See user documentation for more details.

Conversely an op_put_data() routine may also be implemented later (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.

2.7 Performance Measurements

For measuring the execution time of code, two timer routines are implemented. Firstly, op_timers_core() (in op_lib_core.c) measures the elapsed time on a single MPI process while op_timers() (in op_mpi_decl.c) has an implicit MPI_Barrier() so that time across the whole MPI universe can be measured. The time spent in the op_par_loop() calls is measured and accumulated. The setup costs due to halo creation and partitioning are 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 times the op_par_loop() routine is called, (3) the indices of the op_dats 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.

Currently, the only way to identify a loop is by its name. Thus we use a hash function to compute a key corresponding to the op_mpi_kernel struct (Figure 8) for each loop and store it in a hash table. 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

```
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;
```

Figure 8: MPI performance measurement collection struct

cause some performance degradation, we allow the MPI message monitoring to be enabled at compile time using the `-DCOMM_PERF` switch.

2.8 Garbage Collection

At the end of the OP2 application a call to `op_exit()` will free all halo lists, MPI send buffers and the table holding performance measures. Also any `dat`s and `map`s held internally by OP2 is freed.

3 HDF5 File I/O

The current hdf5 file format follows the ASCI file format generated by the naca0012.m Airfoil mesh generator. The generated hdf5 file structure and contents has can be viewed through the `h5dump` utility. The hdf5 I/O routines that allows to read and write `op_set`s, `op_map`s, `op_dat`s and constants are detailed in the user documentation.

4 Partitioning

Given the unstructured mesh in an OP2 application, distributing the data on sets and mapping tables across the MPI universe is achieved by a mesh partitioner in order to avoid building large halos. OP2's aim is to achieve good partitions without the intervention of the application programmer. Once the OP2 declaration routines are executed, invoking `op_partition()` from the application, with the appropriate arguments (see user guide) will partition the sets and maps and migrate the data to new MPI processes as required. There are a number of grid/mesh partitioners that can be used for this task. The best partitioner for a given application can be selected as requred. The current distributed memory implementation gives the option of using the following partitioning routines:

- a geometric partitioning with ParMetis [1]
- k-way graph partitioning with ParMetis [1]
- k-way graph partitioning with PT-Scotch [2]
- Inertial coordinate bisection partitioning (for 3D meshes only) based on the original OPlus partitioning in Hydra
- A user defined partitioning that can be read from an hdf5 file

OP2 also provides a number of supporting functions and data migration routines to facilitate the above goals. OP2 should partition the mesh immediately after all the calls to `op_decl_*`. For this OP2 assumes that an initial parallel distribution of the sets and mapping tables has been performed during input, either by user defined I/O routines or using the HDF5 parallel I/O routines. For example in the `airfoil_plain` application the data and mappings are distributed in a block partitioning fashion. The partitioning of the sets are performed by calls to wrapper functions: `op_partition_geom()`, `op_partition_kway()` or `op_partition_ptscotch()` defined in `op_mpi_ part_core.c`. A wrapper function is required to organize the data and/or mesh elements into a format that is acceptable to the ParMetis and PT-Scotch partitioning routines. We anticipate that supporting other partitioners may require new wrapper functions to be developed into the MPI back-end.

For example in the airfoil application, the xy coordinates of the nodes are supplied in `p_x`. Thus `op_partition()` can be utilized with `p_x`. This will utilize ParMETIS's geometric partitioning by calling the wrapper function `op_partition_geom()`. 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 partitioned (or 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.

The logic of secondary sets *inheriting* the partitioning from the primary set is as follows. We first compute a cost associated with using each mapping table to partition some secondary set using a partitioned set starting from the primary set. Partitioning a set using a mapping *from* a partitioned set costs more than partitioning a set using a mapping *to* a partitioned set. Each secondary set is partitioned using the map identified as the one that gives the smallest cost. We assign some integer value to indicate the cost.

For example if we have a cells to nodes mapping and the primary set is nodes (and has been partitioned) using the map then we can determine where each cell should reside. Thus if a majority

of the nodes that is pointed to by a cell resides in some partition X (i.e. MPI rank X) then the cell is also best placed on partition X (if not already on X). A similar reasoning is used when partitioning the nodes, given the primary set is cells. In this case a temporary reverse map is created (i.e. a nodes to cells mapping) to determine the partition of nodes. After all the set elements have been assigned a partition a call to, `migrate_all()` will migrates 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 data structures used for partitioning is freed as part of garbage collection. For debugging purposes, we have also implemented a wrapper function: `op_partition_random()` that performs a random partitioning of a given set. Currently partitioning and halo creation is achieved by a call to `op_partition()` with the appropriate arguments (to select the specific library and `op_set`, `op_map` and `op_dat`) as detailed in the user guide. The parallel partitioning only occurs when distributed memory parallelism (i.e. MPI back-end) is used. Otherwise, dummy (null operations) routines are substituted in place of the actual partitioner calls.

4.1 Mesh Renumbering

OP2 allows the ordering or numbering of mesh elements in an unstructured mesh to be optimized. The renumbering of the execution set and related sets that are accessed through indirections has an important effect on performance [6]: cache locality can be improved by making sure that data accessed by elements which are executed consecutively are close, so that data and cache lines are reused. OP2, implements a renumbering routine that can be called to convert the input data meshes based on the Gibbs-Poole-Stockmeyer algorithm in Scotch [2]. The renumbering is implemented in `../op2/c/src/externlib/op_renumber.cpp`.

However this renumbering is currently only works on a single node (i.e. no MPI support) and the usual method of utilizing its benefits is to read in an unoptimized mesh in an OP2 HDF5 file, use the mesh renumbering to optimize the numbering of this mesh and then write the resulting mesh back to a new hdf5 file. The new hdf5 file can then be used on both single node and distributed memory versions of the application.

```
for each op dat requiring a halo exchange {
  execute CUDA kernel to gather export halo data
  copy export halo data from GPU to host
  start non-blocking MPI communication
}
for each color (i) {
  if color ! = core colors {
    wait for all MPI communications to complete
    for each op dat requiring a halo exchange
      copy import halo data from host to GPU
  }
  execute CUDA kernel for color (i) mini-partitions
}
```

Figure 9: MPI+CUDA halo exchange

5 Heterogeneous Back-ends

For Heterogeneous systems (such as distributed memory clusters of GPUs) at least two layers of parallelization needs to be utilized simultaneously (1) distributed memory (process level parallelism) and (2) single-node/shared-memory (thread level parallelism). As such the design for heterogeneous platforms involve two primary considerations; (1) combining the owner compute strategy across nodes and coloring strategy within a node and (2) implementing overlapping of computation with communication within the "plan" construction phase of OP2. For distributed memory clusters of GPUs, the OP2 design assumes that one MPI process will have access to only one GPU. Thus MPI will be used across nodes (where each node is interconnected by a communication network such as InfiniBand) and CUDA within each GPU node. For clusters with each node consisting of multiple GPUs, OP2 assigns one MPI process per GPU. This simplifies the execution on heterogeneous cluster systems by allowing separate processes (and not threads) to manage any multiple GPUs on a single node. At runtime, on each node, each MPI process will select any available GPU device. Code generation with such a strategy reuses the single node code generation with only a few minor modifications as there is no extra level of thread management/partitioning within a node for multiple GPUs. As the MPI back-end achieves overlapping of computation with communication by separating the set-elements into two groups, the *core* elements can be computed over without accessing any halo data. To achieve the same objective on a cluster of GPUs, for each `op_par_loop` that does halo exchanges, OP2 assigns mini-partitions such that each will consists only either *core* elements or non-*core* element (including execute halo, *ieh* elements). This will allow to assign coloring to mini-partitions such that one set of colors are exclusively for mini-partitions containing only *core* element's while a different set will be assigned for the others. As such the pseudo-code for executing an `op_par_loop` on a single GPU within a GPU cluster is detailed in Figure 9.

The *core* elements will be computed while non-blocking communications are in-flight. The coloring of mini-partitions is ordered such that the mini-partitions with the non-*core* elements will be computed after all the *core* elements are computed. This allows for an MPI wait_all to be placed before non-`core` colors are reached. Each `op_plan` consists of a mini-partitioning and coloring strategy optimized for their respective loop and number of elements. In the above pseudo-code the halos are transfered via MPI by first copying it to the host over the PCIe bus. As such its an implementation that does not utilize NVIDIA's new GPUDirect [3] technology for transferring data directly between GPUs. However, OP2's latest release has an implementation that utilize GPUDi-

rect (see user guide on how to enable this mode). With GPUDirect the host copy statements in the above code is not required where simply calling the MPI send and receives will result in the required communications between two GPUs.

The multi-threaded CPU cluster implementation is based on MPI and OpenMP and follows a similar design to the GPU cluster design except that there is no data transfer to and from a discretely attached accelerator; all the data resides in CPU main memory.

Currently, for simplicity the OP2 design does not utilize both the host (CPU) and the accelerator (GPU) simultaneously for the problem solution. However, such a design is a possible avenue for future work. One possibility is to assign an MPI process that performs computations on the host CPU and another MPI process that "manages" the computations on the GPU attached to the host. The managing MPI process will utilize MPI and CUDA in exactly the same way described above, while the MPI process computing on the host will either use the single threaded implementation (MPI only) or multi-threaded (MPI and OpenMP) implementation. The key issue in this case is on assigning and managing the load on the different processors depending on their relative speeds for solving a given mesh computation.

5.1 Hybrid CPU/GPU Execution

The OP2 back-end also supports the execution of the problem on both the CPUs and the GPUs on a node. This is called a fully hybrid execution. Currently only applications written with the Fortran back-end can utilize this feature, as this was developed for Rolls Royce's Hydra. See [7] for initial performance results.

The natural approach to enable hybrid CPU-GPU execution in OP2 is to assign some processes to execute on the GPU and others to execute on the CPU. This hardware selection happens at runtime: on a node with N GPUs, the first N processes assigned to it pick up a GPU and the rest become CPU processes. To enable hybrid execution, the generated kernel files include code for execution with both MPI+CUDA and MPI+OpenMP, thus at runtime the different MPI processes assigned to different hardware can call the appropriate one.

The most important challenge with hybrid execution in general is to appropriately load balance between different hardware so that both are utilized as much as possible. Finding such a balance for simple applications where one computational phase (such as a single loop) dominates the runtime may not be difficult. One only needs to compare execution times on the CPU and the GPU separately and assign proportionally sized partitions to the two. However, for an application such as Hydra consisting of several phases of computations, such a load balancing is not trivial: the performance difference between the CPU and the GPU varies widely for different loops.

Currently the partitioning for a fully hybrid execution can only be carried out using the heterogeneous load balancing feature of ParMetis. This is set via run-time arguments:

```
export OP_HYBRID_BALANCE=2.5
mpirun -np 3 ./OP2_application
```

Figure 10: MPI+CUDA fully hybrid execution

If the above is executed on a 2 CPU node with a single GPU, a partition size balance of 2.5 implies that the GPU executes a partition that is 2.5 times larger than a single CPU (i.e. 1.25 times larger than the combined size of the partitions assigned to the two CPUs).

6 To do list

- Implement automatic check-pointing over MPI

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/

[3] NVIDIA GPUDirect,
http://developer.nvidia.com/gpudirect

[4] METIS
http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/manual.pdf

[5] P. I. Crumpton and M. B. Giles, *Multigrid Aircraft Computations Using the OPlus Parallel Library*, Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers, 339-346, A. Ecer, J. Periaux, N. Satofuka, and S. Taylor, (eds.), North-Holland, 1996.

[6] D. A. Burgess and M. B. Giles. 1997. *Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines.* Adv. Eng. Softw. 28 (April 1997), 189–201. Issue 3. ISSN 0965-9978.

[7] István Z. Reguly and Gihan R. Mudalige and Carlo Bertolli and Michael B. Giles and Adam Betts and Paul H.J. Kelly and David Radford. *Acceleration of a Full-scale Industrial CFD Application with OP2* (Under Review)