# OPS C++ User's Manual

Mike Giles, Istvan Reguly, Gihan Mudalige

December 2015

# Contents

# 1 Introduction

OPS is a high-level framework with associated libraries and preprocessors to generate parallel executables for applications on **multi-block structured grids**. Multi-block structured grids consists of an unstructured collection of structured meshes/grids. This document describes the OPS C++ API, which supports the development of single-block and multi-block structured meshes.

Many of the API and library follows the structure of the OP2 high-level library for unstructured mesh applications [1]. However the structured mesh domain is distinct from the unstructured mesh applications domain meshes/grids. The key idea is that operations involve looping over a "rectangular" multi-dimensional set of grid points using one or more "stencils" to access data. In multi-block grids, we have several structured blocks. The connectivity between the faces of different blocks can be quite complex, and in particular they may not be oriented in the same way, i.e. an $i, j$ face of one block may correspond to the $j, k$ face of another block. This is awkward and hard to handle simply.
To clarify some of the important issues in designing the API, we note here some needs connected with a 3D application:

- When looping over the interior with loop indices $i, j, k$, often there are 1D arrays which are referenced using just one of the indices.
- To implement boundary conditions, we often loop over a 2D face, accessing both the 3D dataset and data from a 2D dataset.
- To implement periodic boundary conditions using dummy "halo" points, we sometimes have to copy one plane of boundary data to another. e.g. if the first dimension has size $I$ then we might copy the plane $i = I-2$ to plane $i = 0$, and plane $i = 1$ to plane $i = I-1$.
- In multigrid, we are working with two grids with one having twice as many points as the other in each direction. To handle this we require a stencil with a non-unit stride.
- In multi-block grids, we have several structured blocks. The connectivity between the faces of different blocks can be quite complex, and in particular they may not be oriented in the same way, i.e. an $i, j$ face of one block may correspond to the $j, k$ face of another block. This is awkward and hard to handle simply.

The latest proposal is to handle all of these different requirements through stencil definitions.

# 2 OPS C++ API

## 2.1 Initialisation and termination routines

**void ops_init(int argc, char **argv, int diags_level)**
This routine must be called before all other OPS routines.

| | |
|---|---|
| `argc, argv` | the usual command line arguments |
| `diags_level` | an integer which defines the level of debugging diagnostics and reporting to be performed |

Currently, higher `diags_level`s does the following checks
`diags_level` = 1 : no diagnostics, default to achieve best runtime performance.
`diags_level` > 1 : print block decomposition and `ops_par_loop` timing breakdown.
`diags_level` > 4 : print intra-block halo buffer allocation feedback (for OPS internal development only)
`diags_level` > 5 : check if intra-block halo MPI sends depth match MPI receives depth (for OPS internal development only)

**ops_block ops_decl_block(int dims, char *name)**
This routine defines a structured grid block.

| | |
|---|---|
| `dims` | dimension of the block |
| `name` | a name used for output diagnostics |

**ops_block ops_decl_block_hdf5(int dims, char *name, char *file)**
This routine reads the details of a structured grid block from a named HDF5 file

| | |
|---|---|
| `dims` | dimension of the block |
| `name` | a name used for output diagnostics |
| `file` | hdf5 file to read and obtain the block information from |

Although this routine does not read in any extra information about the block from the named HDF5 file than what is already specified in the arguments, it is included here for error checking (e.g. check if blocks defined in an HDF5 file is matching with the declared arguments in an application) and completeness.

**ops_dat ops_decl_dat(ops_block block, int dim, int* size, int *base,**
**int *d_m, int *d_p, T *data, char *type, char *name)**
This routine defines a dataset.

| | |
|---|---|
| `block` | structured block |
| `dim` | dimension of dataset (number of items per grid element) |
| `size` | size in each dimension of the block |
| `base` | base indices in each dimension of the block |
| `d_m` | padding from the face in the negative direction for each dimension (used for block halo) |
| `d_p` | padding from the face in the positive direction for each dimension (used for block halo) |

| | |
|---|---|
| `data` | input data of type `T` |
| `type` | the name of type used for output diagnostics (e.g. "double", "float") |
| `name` | a name used for output diagnostics |

The `size` allows to declare different sized data arrays on a given `block`. `d_m` and `d_p` are depth of the "block halos" that are used to indicate the offset from the edge of a block (in both the negative and positive directions of each dimension).

**ops_dat ops_decl_dat_hdf5(ops_block block, int dim, char *type, char *name, char *file)**
This routine defines a dataset to be read in from a named hdf5 file

| | |
|---|---|
| `block` | structured block |
| `dim` | dimension of dataset (number of items per grid element) |
| `type` | the name of type used for output diagnostics (e.g. "double", "float") |
| `name` | name of the dat used for output diagnostics |
| `file` | hdf5 file to read and obtain the data from |

**void ops_decl_const(char const * name, int dim, char const * type, T * data )**
This routine defines a global constant. Global constants needs to be declared upfront so that they can be correctly handled for different parallelizations. For e.g CUDA on GPUs. Once defined they remain unchanged throughout the program, unless changed by a call to ops_update_const(..)

| | |
|---|---|
| `name` | a name used to identify the constant |
| `dim` | dimension of dataset (number of items per element) |
| `type` | the name of type used for output diagnostics (e.g. "double", "float") |
| `data` | pointer to input data of type `T` |

**void ops_update_const(char const * name, int dim, char const * type, T * data)**
This routine updates/changes the value of a constant

| | |
|---|---|
| `name` | a name used to identify the constant |
| `dim` | dimension of dataset (number of items per element) |
| `type` | the name of type used for output diagnostics (e.g. "double", "float") |
| `data` | pointer to new values for constant of type `T` |

**ops_halo ops_decl_halo(ops_dat from, ops_dat to, int *iter_size, int* from_base, int *to_base, int *from_dir, int *to_dir)**
This routine defines a halo relationship between two datasets defined on two different blocks.

| | |
|---|---|
| `from` | origin dataset |
| `to` | destination dataset |
| `iter_size` | defines an iteration size (number of indices to iterate over in each direction) |
| `from_base` | indices of starting point in "from" dataset |
| `to_base` | indices of starting point in "to" dataset |
| `from_dir` | direction of incrementing for "from" for each dimension of `iter_size` |

| | |
|---|---|
| `to_dir` | direction of incrementing for "to" for each dimension of `iter_size` |

A from_dir [1,2] and a to_dir [2,1] means that x in the first block goes to y in the second block, and y in first block goes to x in second block. A negative sign indicates that the axis is flipped. (Simple example: a transfer from (1:2,0:99,0:99) to (-1:0,0:99,0:99) would use iter_size = [2,100,100], from_base = [1,0,0], to_base = [-1,0,0], from_dir = [0,1,2], to_dir = [0,1,2]. In more complex case this allows for transfers between blocks with different orientations.)

**ops_halo ops_decl_halo_hdf5(ops_dat from, ops_dat to, char* file)**
This routine reads in a halo relationship between two datasets defined on two different blocks from a named HDF5 file

| | |
|---|---|
| `from` | origin dataset |
| `to` | destination dataset |
| `file` | hdf5 file to read and obtain the data from |

**ops_halo_group ops_decl_halo_group(int nhalos, ops_halo *halos)**
This routine defines a collection of halos. Semantically, when an exchange is triggered for all halos in a group, there is no order defined in which they are carried out.

| | |
|---|---|
| `nhalos` | number of halos in `halos` |
| `halos` | array of halos |

**ops_reduction ops_decl_reduction_handle(int size, char *type, char *name)**
This routine defines a reduction handle to be used in a parallel loop

| | |
|---|---|
| `size` | size of data in bytes |
| `type` | the name of type used for output diagnostics (e.g. "double", "float") |
| `name` | name of the dat used for output diagnostics |

**void ops_reduction_result(ops_reduction handle, T *result)**
This routine returns the reduced value held by a reduction handle

| | |
|---|---|
| `handle` | the `ops_reduction` handle |
| `result` | a pointer to write the results to, memory size has to match the declared |

**ops_partition(char *method)**
Triggers a multi-block partitioning across a distributed memory set of processes. (links to a dummy function for single node parallelizations). This routine should only be called after all the ops_halo ops_decl_block and ops_halo ops_decl_dat statements have been declared

| | |
|---|---|
| `method` | string describing the partitioning method. Currently this string is not used internally, but is simply a place-holder to indicate different partitioning methods in the future. |

**void ops_diagnostic_output()**
This routine prints out various useful bits of diagnostic info about sets, mappings and datasets. Usually used right after an ops_partition() call to print out the details of the decomposition

**void ops_printf(const char * format, ...)**
This routine simply prints a variable number of arguments; it is created is in place of the standard C printf function which would print the same on each MPI process

**void ops_timers(double *cpu, double *et)**
gettimeofday() based timer to start/end timing blocks of code

| | |
|---|---|
| `cpu` | variable to hold the CPU time at the time of invocation |
| `et` | variable to hold the elapsed time at the time of invocation |

**void ops_fetch_block_hdf5_file(ops_block block, chat *file)**
Write the details of an ops_block to a named HDF5 file. Can be used over MPI (puts the data in an ops_dat into an HDF5 file using MPI I/O)

| | |
|---|---|
| `block` | ops_block to to be written |
| `file` | hdf5 file to write to |

**void ops_fetch_stencil_hdf5_file(ops_stencil stencil, chat *file)**
Write the details of an ops_block to a named HDF5 file. Can be used over MPI (puts the data in an ops_dat into an HDF5 file using MPI I/O)

| | |
|---|---|
| `stencil` | ops_stencil to to be written |
| `file` | hdf5 file to write to |

**void ops_fetch_dat_hdf5_file(ops_dat dat, chat *file)**
Write the details of an ops_block to a named HDF5 file. Can be used over MPI (puts the data in an ops_dat into an HDF5 file using MPI I/O)

| | |
|---|---|
| `dat` | ops_dat to to be written |
| `file` | hdf5 file to write to |

**void ops_print_dat_to_txtfile(ops_dat dat, chat *file)**
Write the details of an ops_block to a named text file. When used under an MPI parallelization each MPI process will write its own data set separately to the text file. As such it does not use MPI I/O. The data can be viewed using a simple text editor

| | |
|---|---|
| `dat` | ops_dat to to be written |
| `file` | text file to write to |

**void ops_timing_output(FILE *os)**
Print OPS performance performance details to output stream

| | |
|---|---|
| `os` | output stream, use stdout to print to standard out |

**void ops_exit()**
This routine must be called last to cleanly terminate the OPS computation.

## 2.2 Halo exchange

**void ops_halo_transfer(ops_halo_group group)**
This routine exchanges all halos in a halo group and will block execution of subsequent computations that depend on the exchanged data.

| | |
|---|---|
| `group` | the halo group |

### 2.3 Parallel loop syntax

A parallel loop with N arguments has the following syntax:
**void ops_par_loop( void (*kernel)(...),**
**char *name, ops_blck block, int dims, int *range,**
**ops_arg arg1, ops_arg arg2, . . . , ops_arg argN )**

| | |
|---|---|
| `kernel` | user's kernel function with N arguments |
| `name` | name of kernel function, used for output diagnostics |
| `block` | the ops_block over which this loop executes |
| `dims` | dimension of loop iteration |
| `range` | iteration range array |
| `args` | arguments |

The **ops_arg** arguments in **ops_par_loop** are provided by one of the following routines, one for global constants and reductions, and the other for OPS datasets.

**ops_arg ops_arg_gbl(T *data, int dim, char *type, ops_access acc)**

| | |
|---|---|
| `data` | data array |
| `dim` | array dimension |
| `type` | string representing the type of data held in data |
| `acc` | access type |

**ops_arg ops_arg_reduce(ops_reduction handle, int dim, char *type, ops_access acc)**

| | |
|---|---|
| `handle` | an `ops_reduction` handle |
| `dim` | array dimension (according to `type`) |
| `type` | string representing the type of data held in data |
| `acc` | access type |

**ops_arg ops_arg_dat(ops_dat dat, ops_stencil stencil, char *type, ops_access acc)**

| | |
|---|---|
| `dat` | dataset |
| `stencil` | stencil for accessing data |
| `type` | string representing the type of data held in dataset |
| `acc` | access type |

**ops_arg ops_arg_idx()**
Give you an array of integers (in the user kernel) that have the index of the current grid point, i.e. idx[0] is the index in x, idx[1] is the index in y, etc. This is a globally consistent index, so even if the block is distributed across different MPI partitions, it gives you the same indexes. Generally used to generate initial geometry.

### 2.4 Stencils

The final ingredient is the stencil specification, for which we have two versions: simple and strided.

**ops_stencil ops_decl_stencil(int dims, int points, int *stencil, char *name)**

| | |
|---|---|
| `dims` | dimension of loop iteration |
| `points` | number of points in the stencil |
| `stencil` | stencil for accessing data |
| `name` | string representing the name of the stencil |

**ops_stencil ops_decl_strided_stencil(int dims, int points, int *stencil, int *stride, char *name)**

| | |
|---|---|
| `dims` | dimension of loop iteration |
| `points` | number of points in the stencil |
| `stencil` | stencil for accessing data |
| `stride` | stride for accessing data |
| `name` | string representing the name of the stencil |

**ops_stencil ops_decl_stencil_hdf5(int dims, int points, char *name, char* file)**

| | |
|---|---|
| `dims` | dimension of loop iteration |
| `points` | number of points in the stencil |
| `name` | string representing the name of the stencil |
| `file` | hdf5 file to write to |

In the strided case, the indices for referencing the data used by point `p` are defined as:

```
stride[m]*loop_index[m] + stencil[p*dims+m]
```

If, for one or more dimensions, both `stride[m]` and `stencil[p*dims+m]` are zero, then one of the following must be true;

- the dataset being referenced has size 1 for these dimensions
- these dimensions are to be omitted and so the dataset has dimension equal to the number of remaining dimensions.

These two stencil definitions probably take care of all of the cases in the Introduction except for multiblock applications with interfaces with different orientations – this will need a third, even more general, stencil specification. The strided stencil will handle both multigrid (with a stride of 2 for example) and the boundary condition and reduced dimension applications (with a stride of 0 for the relevant dimensions).

## 2.5 Checkpointing

OPS supports the automatic checkpointing of applications. Using the API below, the user specifies the file name for the checkpoint and an average time interval between checkpoints, OPS will then automatically save all necessary information periodically that is required to fast-forward to the last checkpoint if a crash occurred. Currently, when re-launching after a crash, the same number of MPI processes have to be used. To enable checkpointing mode, the `OPS_CHECKPOINT` runtime argument has to be used.

**bool ops_checkpointing_init(const char *filename, double interval, int options)**
Initialises the checkpointing system, has to be called after `ops_partition`. Returns true if the application launches in restore mode, false otherwise.

| | |
|---|---|
| `filename` | name of the file for checkpointing. In MPI, this will automatically be post-fixed with the rank ID. |
| `interval` | average time (seconds) between checkpoints |
| `options` | a combinations of flags, listed in `ops_checkpointing.h`:<br>OPS_CHECKPOINT_INITPHASE - indicates that there are a number of parallel loops at the very beginning of the simulations which should be excluded from any checkpoint; mainly because they initialise datasets that do not change during the main body of the execution. During restore mode these loops are executed as usual. An example would be the computation of the mesh geometry, which can be excluded from the checkpoint if it is re-computed when recovering and restoring a checkpoint. The API call void `ops_checkpointing_initphase_done()` indicates the end of this initial phase.<br>OPS_CHECKPOINT_MANUAL_DATLIST - Indicates that the user manually controls the location of the checkpoint, and explicitly specifies the list of `ops_dat`s to be saved.<br>OPS_CHECKPOINT_FASTFW - Indicates that the user manually controls the location of the checkpoint, and it also enables fast-forwarding, by skipping the execution of the application (even though none of the parallel loops would actually execute, there may be significant work outside of those) up to the checkpoint.<br>OPS_CHECKPOINT_MANUAL - Indicates that when the corresponding API function is called, the checkpoint should be created. Assumes the presence of the above two options as well. |

**void ops_checkpointing_manual_datlist(int ndats, ops_dat *datlist)**
A use can call this routine at a point in the code to mark the location of a checkpoint. At this point, the list of datasets specified will be saved. The validity of what is saved is not checked by the checkpointing algorithm assuming that the user knows what data sets to be saved for full recovery. This routine should be called frequently (compared to check-pointing frequency) and it will trigger the creation of the checkpoint the first time it is called after the timeout occurs.

| | |
|---|---|
| `ndats` | number of datasets to be saved |
| `datlist` | arrays of `ops_dat` handles to be saved |

**bool ops_checkpointing_fastfw(int nbytes, char *payload)**
A use can call this routine at a point in the code to mark the location of a checkpoint. At this point, the specified payload (e.g. iteration count, simulation time, etc.) along with the necessary datasets, as determined by the checkpointing algorithm will be saved. This routine should be called frequently (compared to checkpointing frequency), will trigger the creation of the checkpoint the first time it is called after the timeout occurs. In restore mode, will restore all datasets the first time it is called, and returns true indicating that the saved payload is returned in payload. Does not save reduction data.

| | |
|---|---|
| `nbytes` | size of the payload in bytes |
| `payload` | pointer to memory into which the payload is packed |

**bool ops_checkpointing_manual_datlist_fastfw(int ndats, op_dat *datlist, int nbytes, char *payload)**
Combines the manual datlist and fastfw calls.

| | |
|---|---|
| `ndats` | number of datasets to be saved |
| `datlist` | arrays of `ops_dat` handles to be saved |
| `nbytes` | size of the payload in bytes |
| `payload` | pointer to memory into which the payload is packed |

**bool ops_checkpointing_manual_datlist_fastfw_trigger(int ndats, opa_dat *datlist, int nbytes, char *payload)**
With this routine it is possible to manually trigger checkpointing, instead of relying on the timeout process. as such it combines the manual datlist and fastfw calls, and triggers the creation of a checkpoint when called.

| | |
|---|---|
| `ndats` | number of datasets to be saved |
| `datlist` | arrays of `ops_dat` handles to be saved |
| `nbytes` | size of the payload in bytes |
| `payload` | pointer to memory into which the payload is packed |

The suggested use of these **manual** functions is of course when the optimal location for checkpointing is known - one of the ways to determine that is to use the built-in algorithm. More details of this will be reported in a tech-report on checkpointing, to be published later.

# 3 OPS User Kernels

In OPS, the elemental operation carried out per mesh/grid point is specified as an outlined function called a *user kernel*. An example taken from the Cloverleaf application is given in Figure 1.

```
void accelerate_kernel( const double *density0, const double *volume,
             double *stepbymass, const double *xvel0, double *xvel1,
             const double *xarea, const double *pressure,
             const double *yvel0, double *yvel1,
             const double *yarea, const double *viscosity) {

double nodal_mass;
//{0,0, -1,0, 0,-1, -1,-1};
nodal_mass = ( density0[OPS_ACC0(-1,-1)] * volume[OPS_ACC1(-1,-1)]
+ density0[OPS_ACC0(0,-1)] * volume[OPS_ACC1(0,-1)]
+ density0[OPS_ACC0(0,0)] * volume[OPS_ACC1(0,0)]
+ density0[OPS_ACC0(-1,0)] * volume[OPS_ACC1(-1,0)] ) * 0.25;

stepbymass[OPS_ACC2(0,0)] = 0.5*dt/ nodal_mass;

//{0,0, -1,0, 0,-1, -1,-1};
//{0,0, 0,-1};
xvel1[OPS_ACC4(0,0)] = xvel0[OPS_ACC3(0,0)] - stepbymass[OPS_ACC2(0,0)] *
( xarea[OPS_ACC5(0,0)]  * ( pressure[OPS_ACC6(0,0)] - pressure[OPS_ACC6(-1,0)] ) +
xarea[OPS_ACC5(0,-1)] * ( pressure[OPS_ACC6(0,-1)] - pressure[OPS_ACC6(-1,-1)] ) );

//{0,0, -1,0, 0,-1, -1,-1};
//{0,0, -1,0};
yvel1[OPS_ACC8(0,0)] = yvel0[OPS_ACC7(0,0)] - stepbymass[OPS_ACC2(0,0)] *
( yarea[OPS_ACC9(0,0)]  * ( pressure[OPS_ACC6(0,0)] - pressure[OPS_ACC6(0,-1)] ) +
yarea[OPS_ACC9(-1,0)] * ( pressure[OPS_ACC6(-1,0)] - pressure[OPS_ACC6(-1,-1)] ) );

//{0,0, -1,0, 0,-1, -1,-1};
//{0,0, 0,-1};
xvel1[OPS_ACC4(0,0)] = xvel1[OPS_ACC4(0,0)] - stepbymass[OPS_ACC2(0,0)] *
( xarea[OPS_ACC5(0,0)] * ( viscosity[OPS_ACC10(0,0)] - viscosity[OPS_ACC10(-1,0)] ) +
xarea[OPS_ACC5(0,-1)] * ( viscosity[OPS_ACC10(0,-1)] - viscosity[OPS_ACC10(-1,-1)] ) );

//{0,0, -1,0, 0,-1, -1,-1};
//{0,0, -1,0};
yvel1[OPS_ACC8(0,0)] = yvel1[OPS_ACC8(0,0)] - stepbymass[OPS_ACC2(0,0)] *
( yarea[OPS_ACC9(0,0)] * ( viscosity[OPS_ACC10(0,0)] - viscosity[OPS_ACC10(0,-1)] ) +
yarea[OPS_ACC9(-1,0)] * ( viscosity[OPS_ACC10(-1,0)] - viscosity[OPS_ACC10(-1,-1)] ) );
}
```

Figure 1: example user kernel

This user kernel is then used in an `ops_par_loop` (Figure 2). The key aspect to note in the user kernel in Figure 1 is the use of the macros `OPS_ACC0, OPS_ACC1, OPS_ACC2` etc. These specifies the stencil in accessing the elements of the respective data arrays. For example in OPS_ACC2, 2 denotes the third function argument (argument number starts from 0) and (0, 0) denotes stencil. At compile time these macros will be expanded to give the correct array index

```
int rangexy_inner_plus1[] = {x_min,x_max+1,y_min,y_max+1};

ops_par_loop(accelerate_kernel, "accelerate_kernel", clover_grid, 2, rangexy_inner_plus1,
 ops_arg_dat(density0, 1, S2D_00_M10_0M1_M1M1, "double", OPS_READ),
 ops_arg_dat(volume, 1, S2D_00_M10_0M1_M1M1, "double", OPS_READ),
 ops_arg_dat(work_array1, 1, S2D_00, "double", OPS_WRITE),
 ops_arg_dat(xvel0, 1, S2D_00, "double", OPS_READ),
 ops_arg_dat(xvel1, 1, S2D_00, "double", OPS_INC),
 ops_arg_dat(xarea, 1, S2D_00_0M1, "double", OPS_READ),
 ops_arg_dat(pressure, 1, S2D_00_M10_0M1_M1M1, "double", OPS_READ),
 ops_arg_dat(yvel0, 1, S2D_00, "double", OPS_READ),
 ops_arg_dat(yvel1, 1, S2D_00, "double", OPS_INC),
 ops_arg_dat(yarea, 1, S2D_00_M10, "double", OPS_READ),
 ops_arg_dat(viscosity, 1, S2D_00_M10_0M1_M1M1, "double", OPS_READ));
```

Figure 2: example `ops_par_loop`

## References

[1] OP2 for Many-Core Platforms, 2013. http://www.oerc.ox.ac.uk/projects/op2