OPS C++ User's Manual

Mike Giles, Istvan Reguly, Gihan Mudalige

May 2019

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 due to the implicit connectivity between neighbouring mesh elements (such as vertices, cells) in structured 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 Key concepts and structure

An OPS applicaiton can generally be divided into two key parts: initialisation and parallel execution. During the initialisation phase, one or more blocks (`ops_block`) are defined: these only have a dimensionality (i.e. 1D, 2D, etc.), and serve to group datasets together. Datasets are defined on a block, and have a specific size (in each dimension of the block), which may be slightly different across different datasets (e.g. staggered grids), in some directions they may be degenerate (a size of 1), or they can represent data associated with different multigrid levels (where their size if a multiple or a fraction of other datasets). Datasets can be declared with empty (NULL) pointers, then OPS will allocate the appropriate amount of memory, may be passed non-NULL pointers (currently only supported in non-MPI environments), in which case OPS will assume the memory is large enough for the data and the block halo, and there are HDF5 dataset declaration routines which allow the distributed reading of datasets from HDF5 files. The concept of blocks is necessary to group datasets together, as in a multi-block problem, in a distributed memory environment, OPS needs to be able to determine how to decompose the problem.

The initialisation phase usually also consists of defining the stencils to be used later on (though they can be defined later as well), which describe the data access patterns used in parallel loops. Stencils are always relative to the "current" point; e.g. if at iteration (i, j), we wish to access $(i-1, j)$ and (i, j), then the stencil will have two points: $\{(-1, 0), (0, 0)\}$. To support degenerate datasets (where in one of the dimensions the dataset's size is 1), as well as for multigrid, there are special strided, restriction, and prolongation stencils: they differ from normal stencils in that as one steps through a grid in a parallel loop, the stepping is done with a non-unit stride for these datasets. For example, in a 2D problem, if we have a degenerate dataset called xcoords, size $(N, 1)$, then we will need a stencil with stride $(1, 0)$ to access it in a regular 2D loop.

Finally, the initialisation phase may declare a number of global constants - these are variables in global scope that can be accessed from within user kernels, without having to pass them in explicitly. These may be scalars or small arrays, generally for values that do not change during execution, though they may be updated during execution with repeated calls to `ops_decl_const`.

The initialisation phase is terminated by a call to `ops_partition`.

The bulk of the application consists of parallel loops, implemented using calls to `ops_par_loop`. These constructs work with datasets, passed through the opaque `ops_dat` handles declared during the initialisation phase. The iterations of parallel loops are semantically independent, and it is the responsibility of the user to enforce this: the order in which iterations are executed cannot affect the result (within the limits of floating point precision). Parallel loops are defined on a block, with a prescribed iteration range that is always defined from the perspective of the dataset written/modified (the sizes of datasets, particularly in multigrid situations, may be very different). Datasets are passed in using `ops_arg_dat`, and during execution, values at the current grid point will be passed to the user kernel. These values are passed wrapped in a templated `ACC<>` object (templated on the type of the data), whose parentheses operator is overloaded, which the user must use to specify the relative offset to access the grid point's neighbours (which accesses have to match the the declared stencil). Datasets written may only be accessed with a one-point, zero-offset stencil (otherwise the parallel semantics may be violated).

Other than datasets, one can pass in read-only scalars or small arrays that are iteration space invariant with `ops_arg_gbl` (typically weights, δt, etc. which may be different in different loops). The current iteration index can also be passed in with `ops_arg_idx`, which will pass a globally consistent index to the user kernel (i.e. also under MPI).

Reductions in loops are done using the ops_arg_reduce argument, which takes a reduction handle as an argument. The result of the reduction can then be acquired using a separate call to `ops_reduction_result`. The semantics are the following: a reduction handle after it was declared is in an "uninitialised" state. The first time it is used as an argument to a loop, its type is determined (increment/min/max), and is initialised appropriately $(0, \infty, -\infty)$, and subsequent uses of the handle in parallel loops are combined together, up until the point, where the result is acquired using `ops_reduction_result`, which then sets it back to an uninitialised state. This also implies, that different parallel loops, which all use the same reduction handle, but are otherwise independent, are independent and their partial reduction results can be combined together associatively and commutatively.

OPS takes responsibility for all data, its movement and the execution of parallel loops. With different execution hardware and optimisations, this means OPS will re-organise data as well as execution (potentially across different loops), and therefore any data accesses or manipulation may only be done through the OPS API.

3 OPS C++ API

3.1 Initialisation declaration 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: a variable in global scope. Global constants need 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_exit()
This routine must be called last to cleanly terminate the OPS computation.

3.2 Diagnostics and output routines

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, char *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 be written
`file`	hdf5 file to write to

void ops_fetch_stencil_hdf5_file(ops_stencil stencil, char *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 be written
`file`	hdf5 file to write to

void ops_fetch_dat_hdf5_file(ops_dat dat, const char *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 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_NaNcheck(ops_dat dat)
Check if any of the values held in the `dat` is a NaN. If a NaN is found, prints an error message and exits.

`dat`	ops_dat to to be checked

3.3 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

3.4 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)
Passes a scalar or small array that is invariant of the iteration space (not to be confused with ops_decl_const, which facilitates global scope variables).

`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)
Passes a pointer to a variable that needs to be incremented (or swapped for min/max reduction) by the user kernel.

`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)
Passes a pointer wrapped in ac ACC¡¿ object to the value(s) at the current grid point to the user kernel. The ACC object's parentheses operator has to be used for dereferencing the pointer.

`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.

3.5 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 semantics for the index of data to be accessed, for stencil point `p`, in dimension `m` are defined as:

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

where `loop_index[m]` is the iteration index (within the user-defined iteration space) in the different dimensions.
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.

See `OPS/apps/c/CloverLeaf/build_field.cpp` and `OPS/apps/c/CloverLeaf/generate.cpp` for an example `ops_decl_strided_stencil` declaration and its use in a loop, respectively.

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).

3.6 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`: 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. 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. 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.

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.

3.7 Access to OPS data

This section describes APIS that give the user access to internal data structures in OPS and return data to user-space. These should be used cautiously and sparsely, as they can affect performance significantly

int ops_dat_get_local_npartitions(ops_dat dat)
This routine returns the number of chunks of the given dataset held by the current process.

`dat`	the dataset

int ops_dat_get_global_npartitions(ops_dat dat)
This routine returns the number of chunks of the given dataset held by all processes.

`dat`	the dataset

void ops_dat_get_extents(ops_dat dat, int part, int *disp, int *sizes)
This routine returns the MPI displacement and size of a given chunk of the given dataset on the current process.

`dat`	the dataset
`part`	the chunk index (has to be 0)
`disp`	an array populated with the displacement of the chunk within the "global" distributed array
`sizes`	an array populated with the spatial extents

char* ops_dat_get_raw_metadata(ops_dat dat, int part, int *disp, int *size, int *stride, int *d_m, int *d_p)
This routine returns array shape metadata corresponding to the ops_dat. Any of the arguments that are not of interest, may be NULL.

`dat`	the dataset
`part`	the chunk index (has to be 0)
`disp`	an array populated with the displacement of the chunk within the "global" distributed array
`size`	an array populated with the spatial extents
`stride`	an array populated strides in spatial dimensions needed for column-major indexing
`d_m`	an array populated with padding on the left in each dimension. Note that these are negative values
`d_p`	an array populated with padding on the right in each dimension

char* ops_dat_get_raw_pointer(ops_dat dat, int part, ops_stencil stencil, ops_memspace *memspace)
This routine returns a pointer to the internally stored data, with MPI halo regions automatically updated as required by the supplied stencil. The strides required to index into the dataset are also given.

`dat`	the dataset
`part`	the chunk index (has to be 0)
`stencil`	a stencil used to determine required MPI halo exchange depths
`memspace`	when set to OPS_HOST or OPS_DEVICE, returns a pointer to data in that memory space, otherwise must be set to 0, and returns whether data is in the host or on the device

void ops_dat_release_raw_data(ops_dat dat, int part, ops_access acc)
Indicates to OPS that a dataset previously accessed with ops_dat_get_raw_pointer is released by the user, and also tells OPS how it was accessed

`dat`	the dataset
`part`	the chunk index (has to be 0)
`acc`	the kind of access that was used by the user (OPS_READ if it was read only, OPS_WRITE if it was overwritten, OPS_RW if it was read and written)

void ops_dat_fetch_data(ops_dat dat, int part, int *data)
This routine copies the data held by OPS to the user-specified memory location, which needs to be at least as large as indicated by the sizes parameter of ops_dat_get_extents.

`dat`	the dataset
`part`	the chunk index (has to be 0)
`data`	pointer to memory which should be filled by OPS

void ops_dat_set_data(ops_dat dat, int part, int *data)
This routine copies the data given by the user to the internal data structure used by OPS. User data needs to be laid out in column-major order and strided as indicated by the sizes parameter of ops_dat_get_extents.

`dat`	the dataset
`part`	the chunk index (has to be 0)
`data`	pointer to memory which should be copied to OPS

4 Tiling for Cache-blocking

OPS has a code generation (ops_gen_mpi_lazy) and build target for tiling. Once compiled, to enable, use the `OPS_TILING` runtime parameter - this will look at the L3 cache size of your CPU and guess the correct tile size. If you want to alter the amount of cache to be used for the guess, use the `OPS_CACHE_SIZE=XX` runtime parameter, where the value is in Megabytes. When MPI is combined with OpenMP tiling can be extended to the MPI halos. Set `OPS_TILING_MAXDEPTH` to increase the the halo depths so that halos for multiple `ops_par_loops` can be exchanged with a single MPI message (see [2] for more details)

To test, compile CloverLeaf under `apps/c/CloverLeaf`, modify clover.in to use a 6144^2 mesh, then run as follows:

For OpenMP with tiling:
```
export OMP_NUM_THREADS=xx;
numactl -physnodebind=0 ./cloverleaf_tiled OPS_TILING
```

For MPI+OpenMP with tiling:
```
export OMP_NUM_THREADS=xx;
mpirun -np xx ./cloverleaf_mpi_tiled OPS_TILING OPS_TILING_MAXDEPTH=6
```

To manually specify the tile sizes (in MB), use the T1, T2, and T3 environment variables:
```
export T1=600; export T2=200;
export OMP_NUM_THREADS=xx;
numactl -physnodebind=0 ./cloverleaf_tiled OPS_TILING
```

5 CUDA and OpenCL Runtime Arguments

The CUDA (and OpenCL) thread block sizes can be controlled by setting the `OPS_BLOCK_SIZE_X`, `OPS_BLOCK_SIZE_Y` and `OPS_BLOCK_SIZE_Z` runtime arguments. For example :

```
./cloverleaf_cuda OPS_BLOCK_SIZE_X=64 OPS_BLOCK_SIZE_Y=4
```

`OPS_CL_DEVICE=XX` runtime flag sets the OpenCL device to execute the code on.
Usually `OPS_CL_DEVICE=0` selects the CPU and `OPS_CL_DEVICE=1` selects GPUs.

6 Executing with GPUDirect

GPU direct support for MPI+CUDA, to enable (on the OPS side) add **-gpudirect** when running the executable. You may also have to use certain environmental flags when using different MPI distributions. For an example of the required flags and environmental settings on the Cambridge Wilkes2 GPU cluster see:
https://docs.hpc.cam.ac.uk/hpc/user-guide/performance-tips.html

7 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.

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 ACC¡¿ objects and their parentheses operator. These specify the stencil in accessing the elements of the respective data arrays.

References

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

[2] Istvan Z. Reguly, G.R. Mudalige, Mike B. Giles. Loop Tiling in Large-Scale Stencil Codes at Run-time with OPS. (2017) IEEE Transactions on Parallel and Distributed Systems. http://dx.doi.org/10.1109/TPDS.2017.2778161

```
id accelerate_kernel( const ACC<double> &density0, const ACC<double> &volume,
              ACC<double> &stepbymass, const ACC<double> &xvel0, ACC<double> &xvel1,
              const ACC<double> &xarea, const ACC<double> &pressure,
              const ACC<double> &yvel0, ACC<double> &yvel1,
              const ACC<double> &yarea, const ACC<double> &viscosity) {

double nodal_mass;

//{0,0, -1,0, 0,-1, -1,-1};
nodal_mass = ( density0(-1,-1) * volume(-1,-1)
  + density0(0,-1) * volume(0,-1)
  + density0(0,0) * volume(0,0)
  + density0(-1,0) * volume(-1,0) ) * 0.25;

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

//{0,0, -1,0, 0,-1, -1,-1};
//{0,0, 0,-1};

xvel1(0,0) = xvel0(0,0) - stepbymass(0,0) *
          ( xarea(0,0)  * ( pressure(0,0) - pressure(-1,0) ) +
            xarea(0,-1) * ( pressure(0,-1) - pressure(-1,-1) ) );

//{0,0, -1,0, 0,-1, -1,-1};
//{0,0, -1,0};

yvel1(0,0) = yvel0(0,0) - stepbymass(0,0) *
          ( yarea(0,0)  * ( pressure(0,0) - pressure(0,-1) ) +
            yarea(-1,0) * ( pressure(-1,0) - pressure(-1,-1) ) );

//{0,0, -1,0, 0,-1, -1,-1};
//{0,0, 0,-1};

xvel1(0,0) = xvel1(0,0) - stepbymass(0,0) *
          ( xarea(0,0) * ( viscosity(0,0) - viscosity(-1,0) ) +
            xarea(0,-1) * ( viscosity(0,-1) - viscosity(-1,-1) ) );

//{0,0, -1,0, 0,-1, -1,-1};
//{0,0, -1,0};

yvel1(0,0) = yvel1(0,0) - stepbymass(0,0) *
          ( yarea(0,0) * ( viscosity(0,0) - viscosity(0,-1) ) +
            yarea(-1,0) * ( viscosity(-1,0) - viscosity(-1,-1) ) );
```

Figure 1: example user kernel

```
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