OP2 C++ User's Manual

Mike Giles, Gihan R. Mudalige, István Reguly

May 2019

Contents

1 Introduction

OP2 is a high-level framework with associated libraries and preprocessors to generate parallel executables for applications on unstructured grids. This document describes the C++ API, but FORTRAN 90 is also supported with a very similar API.

The key concept behind OP2 is that unstructured grids can be described by a number of sets. Depending on the application, these sets might be of nodes, edges, faces, cells of a variety of types, far-field boundary nodes, wall boundary faces, etc. Associated with these are data (e.g. coordinate data at nodes) and mappings to other sets (e.g. edge mapping to the two nodes at each end of the edge). All of the numerically-intensive operations can then be described as a loop over all members of a set, carrying out some operations on data associated directly with the set or with another set through a mapping.

OP2 makes the important restriction that the order in which the function is applied to the members of the set must not affect the final result to within the limits of finite precision floating-point arithmetic. This allows the parallel implementation to choose its own ordering to achieve maximum parallel efficiency. Two other restrictions are that the sets and maps are static (i.e. they do not change) and the operands in the set operations are not referenced through a double level of mapping indirection (i.e. through a mapping to another set which in turn uses another mapping to data in a third set).

OP2 currently enables users to write a single program which can be built into three different executables for different single-node platforms:

- single-threaded on a CPU
- parallelised using CUDA for NVIDIA GPUs
- multi-threaded using OpenMP for multicore CPU systems

A current development branch, also supports AVX vectorisation for x86 CPUs, and OpenCL for both CPUs and GPUS. In addition to this, there is support for distributed-memory MPI parallelisation in combination with any of the above. The user can either use OP2's parallel file I/O capabilities for HDF5 files with a specified structure, or perform their own parallel file I/O using custom MPI code.

2 Overview

A computational project can be viewed as involving three steps:

- writing the program
- debugging the program, often using a small testcase
- running the program on increasingly large applications

With OP2 we want to simplify the first two tasks, while providing as much performance as possible for the third.

To achieve the high performance for large applications, a preprocessor is needed to generate the CUDA code for GPUs or OpenMP code for multicore x86 systems. However, to keep the initial development simple, a development single-threaded executable can be created without any special tools; the user's main code is simply linked to a set of library routines, most of which do little more than error-checking to assist the debugging process by checking the correctness of the user's program. Note that this single-threaded version will not execute efficiently. The preprocessor is needed to generate efficient single-threaded and OpenMP code for CPU systems.

Figure 1 shows the build process for a single thread CPU executable. The user's main program (in this case `jac.cpp`) uses the OP2 header file `op_seq.h` and is linked to the appropriate OP2 libraries using `g++`, perhaps controlled by a Makefile.

Figure 2 shows the build process for the corresponding CUDA executable. The preprocessor parses the user's main program and produces a modified main program and a CUDA file which includes a separate file for each of the kernel functions. These are then compiled and linked to the OP libraries using `g++` and the NVIDIA CUDA compiler `nvcc`, again perhaps controlled by a Makefile.

Figure 3 shows the OpenMP build process which is very similar to the CUDA process except that it uses `*.cpp` files produced by the preprocessor instead of `*.cu` files.

In looking at the API specification, users may think it is a little verbose in places. e.g. users have to re-supply information about the datatype of the datasets being used in a parallel loop. This is a deliberate choice to simplify the task of the preprocessor, and therefore hopefully reduce the chance for errors. It is also motivated by the thought that **"programming is easy; it's debugging which is difficult"**. i.e. writing code isn't time-consuming, it's correcting it which takes the time. Therefore, it's not unreasonable to ask the programmer to supply redundant information, but be assured that the preprocessor or library will check that all redundant information is self-consistent. If you declare a dataset as being of type `OP_DOUBLE` and later say that it is of type `OP_FLOAT` this will be flagged up as an error at run-time.

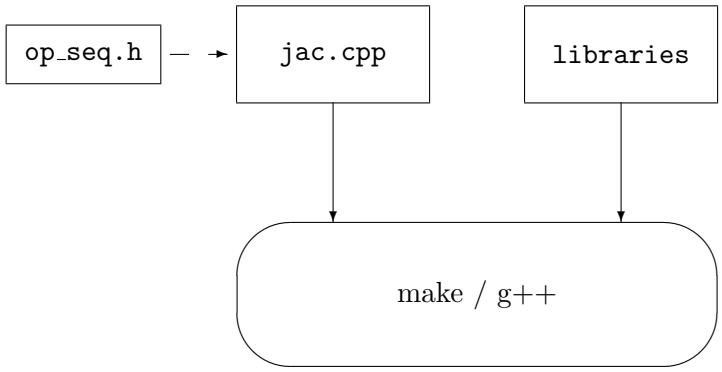

Figure 1: Build process for the development single threaded CPU version

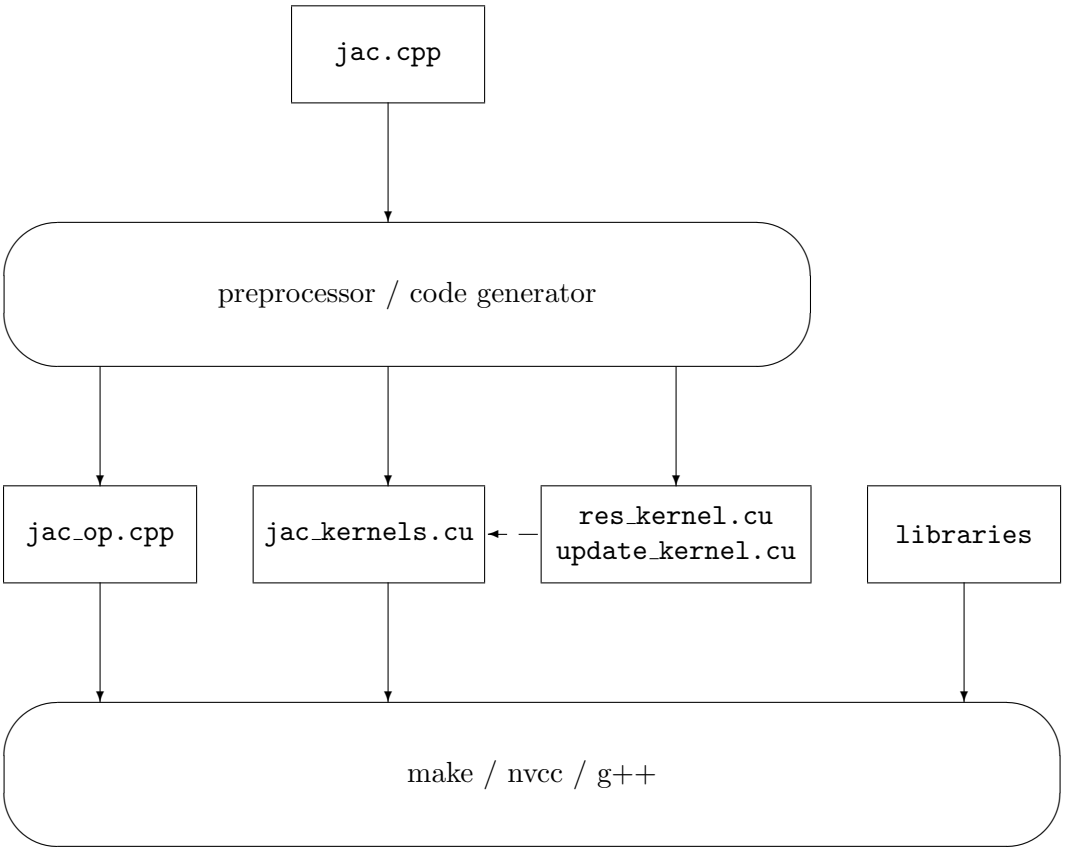

Figure 2: CUDA code build process

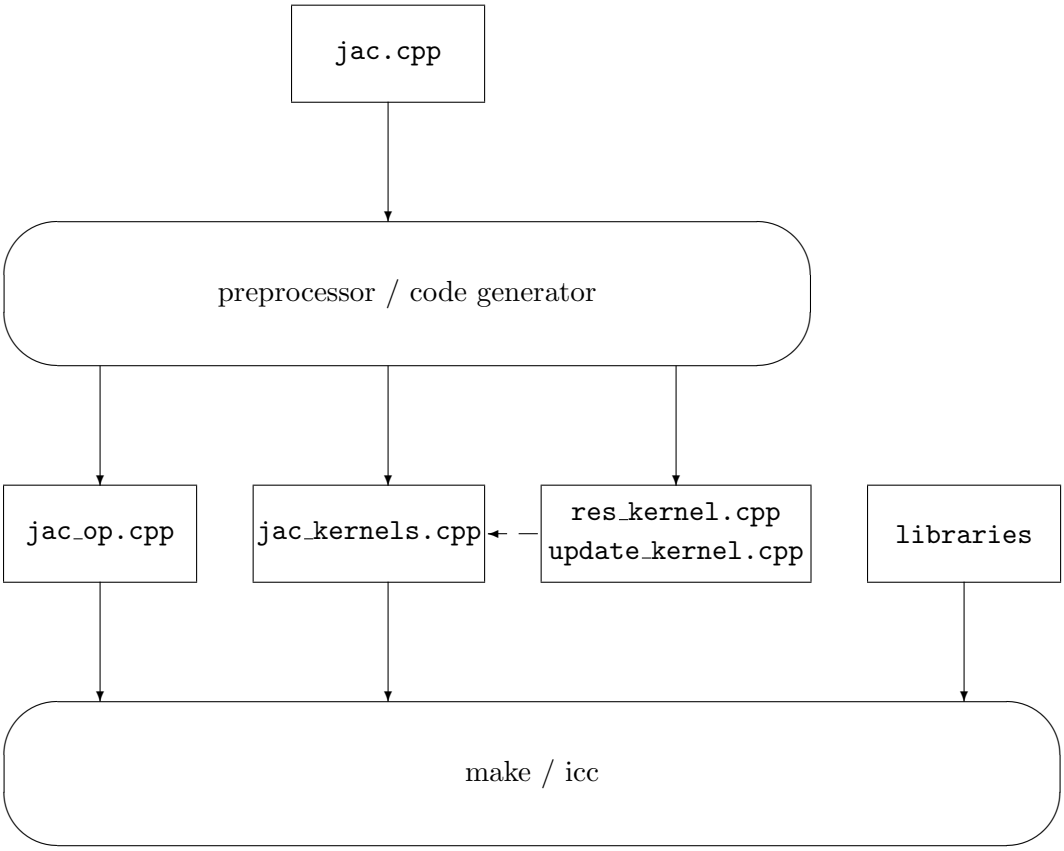

Figure 3: OpenMP code build process

3 OP2 C++ API

3.1 Initialisation and termination routines

void op_init(int argc, char **argv, int diags_level)
This routine must be called before all other OP routines. Under MPI back-ends, this routine also calls `MPI_Init()` unless its already called previously

`argc, argv`	the usual command line arguments
`diags_level`	an integer which defines the level of debugging diagnostics and reporting to be performed; 0 – none; 1 – error-checking; 2 – info on plan construction; 3 – report execution of parallel loops; 4 – report use of old plans; 7 – report positive checks in op_plan_check;

void op_exit()
This routine must be called last to cleanly terminate the OP computation. Under MPI back-ends, this routine also calls `MPI_Finalize()` unless its has been called previously. A runtime error will occur if `MPI_Finalize()` is called after `op_exit()`

op_set op_decl_set(int size, char *name)
This routine defines a set, and returns a set ID.

`size`	number of elements in the set
`name`	a name used for output diagnostics

op_map op_decl_map(op_set from, op_set to, int dim, int *imap, char *name)
This routine defines a mapping from one set to another, and returns a map ID.

`from`	set pointed from
`to`	set pointed to
`dim`	number of mappings per element
`imap`	input mapping table
`name`	a name used for output diagnostics

void op_decl_const(int dim, char *type, T *dat, char *name)
This routine declares constant data with global scope to be used in user's kernel functions. Note: in sequential version, it is the user's responsibility to define the appropriate variable with global scope.

`dim`	dimension of data (i.e. array size) for maximum efficiency, this should be a literal constant (i.e. a number not a variable)
`type`	datatype, either intrinsic ("float", "double", "int", "uint", "ll", "ull" or "bool") or user-defined
`dat`	input data of type `T` (checked for consistency with `type` at run-time)
`name`	global name to be used in user's kernel functions; a scalar variable if `dim=1`, otherwise an array of size `dim`

op_dat op_decl_dat(op_set set, int dim, char *type, T *data, char *name)
This routine defines a dataset, and returns a dataset ID.

`set`	set
`dim`	dimension of dataset (number of items per set element) at present this must be a literal constant (i.e. a number not a variable); this restriction will be removed in the future but a literal constant will remain more efficient
`type`	datatype, either intrinsic or user-defined – expert users can add a qualifier to control data layout and management within OP2 (see section 3.3)
`data`	input data of type `T` (checked for consistency with `type` at run-time) – for each element in `set`, the `dim` data items muct be contiguous, but OP2 may use a different data layout internally for better performance on certain hardware platforms (see section 3.3)
`name`	a name used for output diagnostics

op_dat op_decl_dat_tmp(op_set set, int dim, char *type, char *name)
This routine defines a temporary dataset, initialises it to zero, and returns a dataset ID.

`set`	set
`dim`	dimension of dataset (number of items per set element)
	at present this must be a literal constant (i.e. a number not a variable); this restriction will be removed in the future but a literal constant will remain more efficient
`type`	datatype, either intrinsic or user-defined – expert users can add a qualifier to control data layout and management within OP2 (see section 3.3)
`name`	a name used for output diagnostics

void op_free_dat_tmp(op_dat dat)
This routine terminates a temporary dataset.

`dat`	OP dataset ID

void op_diagnostic_output()
This routine prints out various useful bits of diagnostic info about sets, mappings and datasets

3.2 Parallel loop syntax

A parallel loop with N arguments has the following syntax:

void op_par_loop(void (*kernel)(...), char *name, op_set set,
op_arg arg1, op_arg arg2, ..., op_arg argN)

`kernel`	user's kernel function with N arguments (this is only used for the single-threaded CPU build)
`name`	name of kernel function, used for output diagnostics
`set`	OP set ID
`args`	arguments

The **op_arg** arguments in **op_par_loop** are provided by one of the following routines, one for global constants and reductions, and the other for OP2 datasets. In the future there will be a third one for sparse matrices to support the needs of finite element calculations.

op_arg op_arg_gbl(T *data, int dim, char *typ, op_access acc)

`data`	data array
`dim`	array dimension
`typ`	datatype (redundant info, checked at run-time for consistency)
`acc`	access type: `OP_READ`: read-only `OP_INC`: global reduction to compute a sum `OP_MAX`: global reduction to compute a maximum `OP_MIN`: global reduction to compute a minimum

op_arg op_arg_dat(op_dat dat, int idx, op_map map,
int dim, char *typ, op_access acc)

`dat`	OP dataset ID
`idx`	index of mapping to be used (ignored if no mapping indirection) – a negative value indicates that a range of indices is to be used (see section 3.3 for additional information)
`map`	OP mapping ID (`OP_ID` for identity mapping, i.e. no mapping indirection)
`dim`	dataset dimension (redundant info, checked at run-time for consistency)
	at present this must be a literal constant (i.e. a number not a variable); this restriction will be removed in the future but a literal constant will remain more efficient
`typ`	dataset datatype (redundant info, checked at run-time for consistency)
`acc`	access type: `OP_READ`: read-only `OP_WRITE`: write-only, but without potential data conflict `OP_RW`: read and write, but without potential data conflict `OP_INC`: increment, or global reduction to compute a sum
	The restriction that `OP_WRITE` and `OP_RW` access must not have any potential data conflict means that two different elements of the set cannot through a mapping indirection reference the same elements of the dataset.
	Furthermore, with `OP_WRITE` the user's kernel function must set the value of all `DIM` components of the dataset. If the user's kernel function does not set all of them, the access should be specified to be `OP_RW` since the kernel function needs to read in the old values of the components which are not being modified.

op_arg op_opt_arg_dat(op_dat dat, int idx, op_map map,
int dim, char *typ, op_access acc, int flag)

This is the same as `op_arg op_arg_dat` except for an extra variable `flag`; the argument is only actually used if `flag` has a non-zero value. This routine is required for large application codes (such as HYDRA) which has lots of different features turned on and off by logical flags.

Note that if the user's kernel needs to know the value of `flag` then this must be passed as an additional `op_arg_gbl` argument.

The pointer corresponding to the optional argument in the user kernel must not be dereferenced when the flag is false or not set

3.3 Expert user capabilities

3.3.1 SoA data layout

At present we have an option to force OP2 to use SoA (struct of arrays) storage internally on GPUs. As illustrated in Figure 4 the user always supplies data in AoS (array of structs) layout, with all of the items associated with one set element stored contiguously. On cache-based CPUs this is almost always the most efficient storage layout because it usually maximises the cache hit ratio and reuse of data. However, when doing vector computing (either on GPUs or in the AVX vector units of CPUs) with no indirect addressing, then the SoA format is more efficient.

OP2 can be directed to use the SoA format by setting the environment variable OP_AUTO_SOA=1 before the Python code generator is used. Note that the data should still be supplied by the user in the standard AoS layout; the transposition to SoA format is handled internally by OP2. No changes need to be made to any other user code.

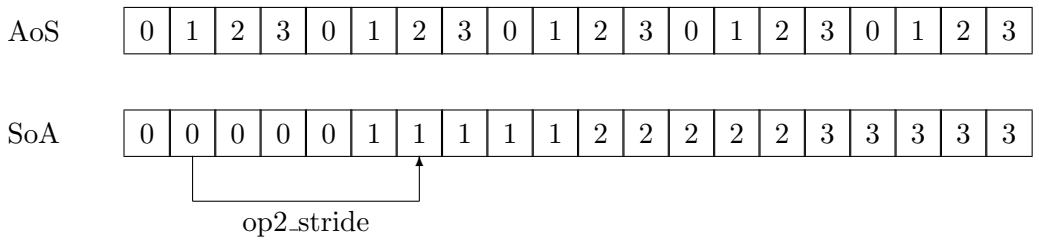

Figure 4: The AoS and SoA layouts for a set with 5 elements, and 4 data items (numbered 0, 1, 2, 3) per element, and the access stride for the SoA storage.

3.3.2 Vector maps

When each of the arguments in a parallel loop uses a single mapping index, the corresponding argument in the user's kernel function is a pointer to an array holding the data items for the set element being pointed to. i.e. the kernel declaration may look something like

```
kernel_routine(float *arg1, float *arg2, float *arg3, float *arg4)
```

If the first 3 arguments correspond to the vertices of a triangle, and the parallel loop is over the set of triangles using a mapping from triangles to vertices, then it may be more natural to combine the first 3 arguments into a single doubly-indexed array as

```
kernel_routine(float *arg1[3], float *arg4)
```

This is obtained by a parallel loop argument having a range of mapping indices (instead of just one) which is accomplished by specifying the mapping index to be `-range` – this means that the set of mapping indices `0 - range-1` is to be used.

3.4 MPI message-passing using HDF5 files

HDF5 has become the *de facto* standard format for parallel file I/O, with various other standards like CGNS layered on top. To make it as easy as possible for users to develop distributed-memory OP2 applications, we provide alternatives to some of the OP2 routines in which the data is read by OP2 from an HDF5 file, instead of being supplied by the user:

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

In addition, there are the following two routines.

op_get_const_hdf5(int dim, char *type, char *file, char *name)
This routine reads a constant (or constant array) from an HDF5 file; if required, the user must then call **op_decl_const** to declare it to OP2.

`dim`	dimension of data (i.e. array size) for maximum efficiency, this should be a literal constant (i.e. a number not a variable)
`type`	datatype, either intrinsic ("float", "double", "int", "uint", "ll", "ull" or "bool") or user-defined; checked at run-time for consistency with `T`
`file`	name of the HDF5 file
`name`	global name to be used in user's kernel functions; a scalar variable if `dim=1`, otherwise an array of size `dim`

void op_partition(char *lib_name, const char* lib_routine,
op_set prime_set, op_map prime_map, op_dat coords)
This routine controls how the various sets are partitioned.

`lib_name`	A string which declares the partitioning library to be used. "PTSCOTCH" - PT-Scotch "PARMETIS" - ParMetis "INERTIAL" - 3D recursive inertial bisection partitioning in OPlus "EXTERNAL" - external partitioning read in from hdf5 file "RANDOM" - select a generic random partitioning (for debugging) If the OP2 library was not built with the specified third-party library, an error message is displayed at runtime and a trivial block-partitioning is used for the remainder of the application.

`lib_routine`	A string which specify the partitioning routine to be used. "KWAY" select the kway graph partitioner in PT-Scotch or ParMetis "GEOM" - select geometric partitioning routine if ParMetis is the `lib_name` "GEOMKWAY" - select geometric partitioning followed by kway partitioning if ParMetis is the `lib_name`
`prime_set`	Specify the primary `op_set` to be partitioned
`prime_map`	Specify the primary `op_map` to be used in the partitioning - to create the adjacency lists for `prime_set` - needed for "KWAY" and "GEOMKWAY"
`prime_set`	Specify the geometric coordinates as an `op_dat` to be used in when using "GEOM" or "GEOMKWAY"

Using the above routines, OP2 will take care of everything, reading in all of the sets, mapping and data, partitoning the sets appropriately, renumbering sets as needed, constructing import/export halo lists, etc., and then performing the parallel computation with halo exchange when needed.

Both MPI and single process executables can be generated, depending on the libraries which are linked in.

3.5 Other I/O and Miscellaneous Routines

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

void op_fetch_data (op_dat dat, T* data)
This routine transfers a copy of the data currently held in an op_dat from the OP2 back-end to a user allocated memory block.

`dat`	OP dataset ID – The op_dat whose data is to be fetched from OP2 space to user space
`data`	pointer to a block of memory of type T – allocated by the user

void op_fetch_data_idx(op_dat dat, T* data, int low, int high)
Transfers a copy of the op_dat's data currently held by OP2 to a user allocated block of memory pointed to by data pointer of type T. The `low` and `high` integers gives the range of elements (or indices) to be fetched. Under MPI (with hdf5) all the processes will hold the same data block(i.e. after an `MPI_Allgather`)

`dat`	OP dataset ID – The op dat whose data is to be fetched from OP2 space to user space
`data`	pointer to a block of memory of type T – allocated by the user
`low`	index of the first element to be fetched
`high`	index of the last element to be fetched

void op_fetch_data_hdf5_file(op_dat dat, char const *file_name)
Write the data in the op_dat to an HDF5 file

`dat`	OP dataset ID – The op dat whose data is to be fetched from OP2 space to user space
`file_name`	the file name to be written to

void op_print_dat_to_binfile(op_dat dat, const char *file_name)
Write the data in the op_dat to a binary file

`dat`	OP dataset ID – The op dat whose data is to be fetched from OP2 space to user space
`file_name`	the file name to be written to

void op_print_dat_to_txtfile(op_dat dat, const char *file_name)
Write the data in the op_dat to a ASCI text file

`dat`	OP dataset ID – The op dat whose data is to be fetched from OP2 space to user space
`file_name`	the file name to be written to

int op_is_root()
A supporting routine that allows to to check for the root process. Intended to be used mainly when the application utilizes HDF5 file I/O and when the user would like to perform some conditional code on the root process. Returns 1 if on MPI_ROOT else 0

int op_get_size(op_set set)
Get the global size of an op_set

`set`	OP set ID

void op_dump_to_hdf5(char const * file_name)
Dump the contents of all the op_sets, op_dats and op_maps to an hdf5 file as held internally by OP2, useful for debugging

`file_name`	the file name to be written to

void op_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 op_timing_output()
Print OP2 performance performance details to STD out

void op_timings_to_csv(char const * file_path)
Write OP2 performance details to csv file. For an MPI code, details are broken down by rank. For an OpenMP code generated with the environment variable OP_TIME_THREADS set, details are broken down by thread. For MPI+OpenMP codes with environment variable OP_TIME_THREADS set, a breakdown of each thread for each MPI rank will be written to the CSV file.

`file_path`	the file to be written to

3.6 MPI message-passing without HDF5 files

Some users will prefer not to use HDF5 files, or at least not to use them in the way prescribed by OP2. To support these users, an application code may do its own file I/O, and then provide the required data to OP2 using the standard routines.

In an MPI application, multiple copies of the same program are executed as separate processes, often on different nodes of a compute cluster. Hence, the OP2 declarations will be invoked on each process. In this case, the behaviour of the OP2 declaration routines is 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`

For example, if an application has 4 processes, $4{\times}10^6$ nodes and $16{\times}10^6$ edges, then each process might be responsible for providing 10^6 nodes and $4{\times}10^6$ edges. Process 0 (the one with MPI rank 0) would be responsible for providing the first 10^6 nodes, process 1 the next 10^6 nodes, and so on, and the same for the edges.

The edge $\rightarrow$ node mapping tables would still contain the same information as in a single process implementation, but process 0 would provide the first 4×10^6 entries, process 1 the next 4×10^6 entries, and so on.

This is effectively using a simple contiguous block partitioning of the datasets, but it is very important to note that this will not be used for the parallel computation. OP2 will re-partition the datasets, re-number the mapping tables as needed (as well as constructing import/export lists for halo data exchange) and will move all data/mappings/datasets to the correct MPI process.

4 Executing with GPUDirect

GPU direct support for MPI+CUDA, to enable (on the OP2 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

5 OP2 Preprocessor/ Code generator

There are three preprocessors for OP2, one developed at Imperial College using ROSE (currently not maintained), a second one developed at Oxford using MATLAB and finally a Python parser/generator also developed at Oxford.

5.1 MATLAB preprocessor

The MATLAB preprocessor is run by the command

```
op2('main')
```

where `main.cpp` is the user's main program. It produces as output

- a modified main program `main_op.cpp` which is used for both the CUDA and OpenMP executables;
- for the CUDA executable, a new CUDA file `main_kernels.cu` which includes one or more files of the form `xxx_kernel.cu` containing the CUDA implementations of the user's kernel functions;
- for the OpenMP executable, a new C++ file `main_kernels.cpp` which includes one or more files of the form `xxx_kernel.cpp` containing the OpenMP implementations of the user's kernel functions.

If the user's application is split over several files it is run by a command such as

```
op2('main','sub1','sub2','sub3')
```

where `sub1.cpp, sub2.cpp, sub3.cpp` are the additional input files which will lead to the generation of output files `sub1_op.cpp, sub2_op.cpp, sub3_op.cpp` in addition to `main_op.cpp, main_kernels.cu, main_kernels.cpp` and the individual kernel files.

The MATLAB preprocessor was the first prototype source-to-source translator developed in the OP2 project. This has been now superseded by the Python code generator.

5.2 Python code generator

The Python preprocessor is run on the command-line with the command

```
./op2.py main.cpp sub1.cpp sub2.cpp sub3.cpp
```

Assuming that the user's application is split over several files. This will lead to the generation of output files `sub1_op.cpp, sub2_op.cpp, sub3_op.cpp` in addition to `main_op.cpp, main_kernels.cu, main_kernels.cpp` and the individual kernel files.

- The modified main program `main_op.cpp` is used for the efficient single threaded CPU (also called as generated sequential or Gen_Seq) OpenMP and CUDA executables;

- For the Gen_Seq and OpenMP executable, `main_kernels.cpp` is a new C++ file which includes one or more files of the form `xxx_kernel.cpp` containing the OpenMP implementations of the user's kernel functions.
- For the CUDA executable, `main_kernels.cu` is a new CUDA file which includes one or more files of the form `xxx_kernel.cu` containing the CUDA implementations of the user's kernel functions. If the OP_AUTO_SOA environmental variable is set, it will generate code that transposes multi-dimensional datasets for faster execution on the GPU.

6 Error-checking

At compile-time, there is a check to ensure that CUDA 3.2 or later is used when compiling the CUDA executable; this is because of compiler bugs in previous versions of CUDA. At run-time, OP2 checks the user-supplied data in various ways:

- checks that a set has a strictly positive number of elements
- checks that a map has legitimate mapping indices, i.e. they map to elements within the range of the target set
- checks that variables have the correct declared type

It would be great to get feedback from users on suggestions for additional error-checking.

7 32-bit and 64-bit CUDA

Section 3.1.6 of the CUDA 3.2 Programming Guide says:

> The 64-bit version of `nvcc` compiles device code in 64-bit mode (i.e. pointers are 64-bit). Device code compiled in 64-bit mode is only supported with host code compiled in 64-bit mode.
>
> Similarly, the 32-bit version of `nvcc` compiles device code in 32-bit mode and device code compiled in 32-bit mode is only supported with host code compiled in 32-bit mode.
>
> The 32-bit version of `nvcc` can compile device code in 64-bit mode also using the `-m64` compiler option.
>
> The 64-bit version of `nvcc` can compile device code in 32-bit mode also using the `-m32` compiler option.

On Windows and Linux systems, there are separate CUDA download files for 32-bit and 64-bit operating systems, so the version of CUDA which is installed matches the operating system. i.e. the 64-bit version is installed on a 64-bit operating system.

Mac OS X can handle both 32-bit and 64-bit executables, and it appears that it is the 32-bit version of `nvcc` which is installed. Therefore the Makefiles in the OP2 distribution may need the `-m64` flag added to `NVCCFLAGS` to produce 64-bit object code.

The Makefiles in the OP2 distribution assume 64-bit compilation and therefore they link to the 64-bit CUDA runtime libraries in `/lib64` within the CUDA toolkit distribution. This will need to be changed to `/lib` for 32-bit code.