OP2 Documentation

Release latest

Gihan Mudalige, Istvan Reguly, Mike Giles

Feb 14, 2022

CONTENTS:

OP2 is a high-level embedded domain specific language (eDSL) for writing **unstructured mesh** algorithms with automatic parellelisation on multi-core and many-core architectures. The API is embedded in both C/C++ and Fortran.

These pages provide detailed documentation on using OP2, including an installation guide, an overview of the C++ API, a walkthrough of the development of an example application, and developer documentation.

CHAPTER

ONE

INTRODUCTION

1.1 Overview

OP2 is a high-level embedded domain specific language (eDSL) for writing **unstructured mesh** algorithms with automatic parellelisation on multi-core and many-core architectures. The API is embedded in both C/C++ and Fortran.

The current OP2 eDSL supports generating code targeting multi-core CPUs with SIMD vectorisation and OpenMP threading, many-core GPUs with CUDA or OpenMP offloading, and distributed memory cluster variants of these using MPI. There is also experimental support for targeting a wider range of GPUs using SYCL and AMD HIP.

These pages provide detailed documentation on using OP2, including an installation guide, an overview of the C++ API, a walkthrough of the development of an example application, and developer documentation.

1.2 Licencing

OP2 is released as an open-source project under the BSD 3-Clause License. See the LICENSE file for more information.

1.3 Citing

To cite OP2, please reference the following paper:

G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli and P. H. J. Kelly, "OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," 2012 Innovative Parallel Computing (InPar), 2012, pp. 1-12, doi: 10.1109/InPar.2012.6339594.

```
@INPROCEEDINGS{6339594,
  author={Mudalige, G.R. and Giles, M.B. and Reguly, I. and Bertolli, C. and Kelly, P.H.
↪J},
  booktitle={2012 Innovative Parallel Computing (InPar)},
  title={OP2: An active library framework for solving unstructured mesh-based␣
↪applications on multi-core and many-core architectures},
  year={2012},
  volume={},
  number={},
  pages={1-12},
  doi={10.1109/InPar.2012.6339594}}
```

1.4 Support

The preferred method of reporting bugs and issues with OPS is to submit an issue via the repository's issue tracker. Users can also email the authors directly by contacting the OP-DSL team.

1.5 Funding

Development of the OP-DSL libraries is or has been supported by the Engineering and Physical Sciences Research Council, the Royal Society, the Hungarian Academy of Sciences, the European Commission and Rolls-Royce plc., UK AWE, NAG. We are also grateful for hardware resources during development from the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, ARCHER and ARCHER2 UK National Supercomputing Service, the University of Oxford Advanced Research Computing (ARC) facility and hardware donations/access from Nvidia and Intel.

CHAPTER

TWO

GETTING STARTED

2.1 Spack

Coming soon.

2.2 Manual Build

2.2.1 Toolchain and Build Dependencies

These are likely provided in some form by either your distribution's package manager or pre-installed and loaded via commands such as with Environment Modules:

- GNU Make > 4.2
- A C/C++ compiler: Currently supported compilers are GCC, Clang, Cray, Intel, IBM XL and NVHPC.
- (Optional) A Fortran compiler: Currently supported compilers are GFortran, Cray, Intel, IBM XL and NVHPC.
- (Optional) An MPI implementation: Any implementation with the `mpicc`, `mpicxx`, and `mpif90` wrappers is supported.
- (Optional) NVIDIA CUDA > 9.2

2.2.2 Library Dependencies

These may also be provided from various package managers and modules, however they must be built with a specific configuration and with the same compiler toolchain that you plan on using to build OP2:

- (Optional) (PT-)Scotch: Used for mesh partitioning. You must build both the sequential Scotch and parallel PT-Scotch with 32-bit indicies (`-DIDXSIZE=32`) and without threading support (remove `-DSCOTCH_PTHREAD`).
- (Optional) ParMETIS: Used for mesh partitioning.
- (Optional) HDF5: Used for HDF5 I/O. You may build with and without `--enable-parallel` (depending on if you need MPI), and then specify both builds via the environment variables listed below.

Note: To build the MPI enabled OP2 libraries you will need a parallel HDF5 build, however you only need a sequential HDF5 build if you need HDF5 support for the sequential OP2 libraries.

2.2.3 Building

First, clone the repository:

```
git clone https://github.com/OP-DSL/OP2-Common.git
cd OP2-Common
```

Then, setup toolchain configuration:

```
export OP2_COMPILER={gnu, cray, intel, xl, nvhpc}
```

Alternatively for a greater level of control:

```
export OP2_C_COMPILER={gnu, clang, cray, intel, xl, nvhpc}
export OP2_C_CUDA_COMPILER={nvhpc}
export OP2_F_COMPILER={gnu, cray, intel, xl, nvhpc}
```

Note: In some scenarios you may be able to use a profile rather than specifying an `OP2_COMPILER`. See makefiles/README.md for more information.

Then, specify the paths to the library dependency installation directories:

```
export PT_SCOTCH_INSTALL_PATH=<path/to/ptscotch>
export PARMETIS_INSTALL_PATH=<path/to/parmetis>
export HDF5_{SEQ, PAR}_INSTALL_PATH=<path/to/hdf5>

export CUDA_INSTALL_PATH=<path/to/cuda/toolkit>
```

Note: You may not need to specify the `X_INSTALL_PATH` varaibles if the include paths and library search paths are automatically injected by your package manager or module system.

If you are using CUDA then you may also specify a comma separated list of target architectures for which to generate code for:

```
export NV_ARCH={Fermi, Kepler, ..., Ampere}[,{Fermi, ...}]
```

Verify the compiler and library setup:

```
make -C op2 detect
```

Finally, build OP2 and an example app:

```
make -C op2 -j$(nproc)
make -C apps/c/airfoil/airfoil_plain/dp -j$(nproc)
```

Warning: The MPI variants of the libraries and apps will only be built if an `mpicxx` executable is found. It is up to you to ensure that the MPI wrapper wraps the compiler you specify via `OP2_COMPILER`. To manually set the path to the MPI executables you may use `MPI_INSTALL_PATH`.

CHAPTER

THREE

OP2 C/C++ MANUAL

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.
- Multi-threaded using OpenMP for multicore CPU systems.
- Parallelised using CUDA for NVIDIA GPUs.

Further to these there are also in-development versions that can emit SYCL and AMD HIP for parallelisation on a wider range of 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.

Note: This documentation describes the C++ API, but FORTRAN 90 is also supported with a very similar API.

3.1 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.

Fig. 3.1 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.

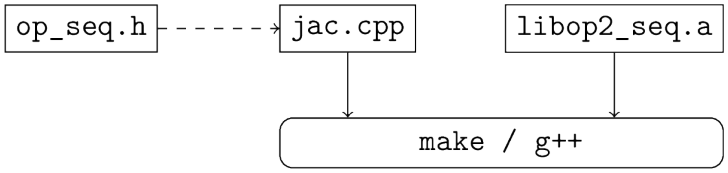

Fig. 3.1: Build process for a single-threaded development executable.

Fig. 3.2 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.

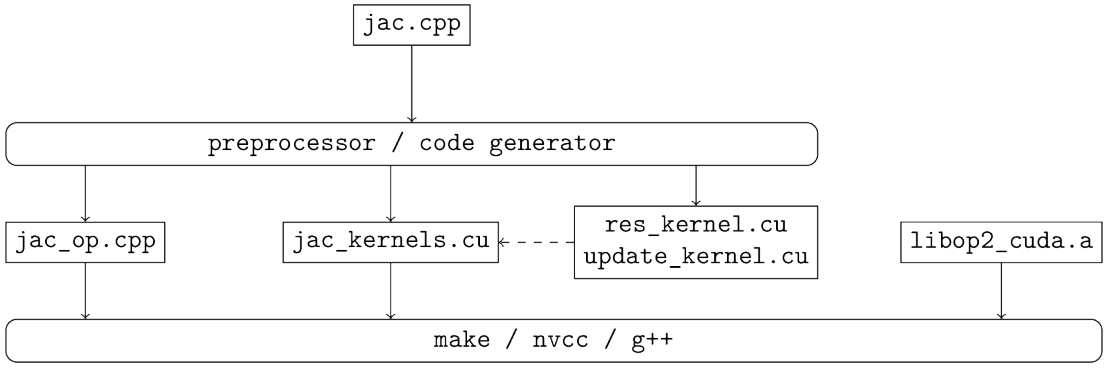

Fig. 3.2: Build process for a CUDA accelerated executable.

Fig. 3.3 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. For example, 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": 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.

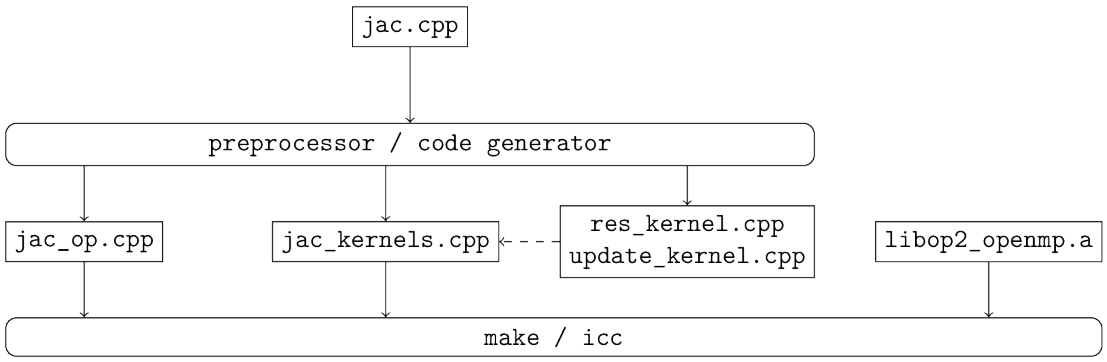

Fig. 3.3: Build process for an OpenMP accelerated executable.

3.2 Initialisation and Termination

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.

Parameters

- **argc** – The number of command line arguments.
- **argv** – The command line arguments, as passed to `main()`.
- **diags_level** – Determines the level of debugging diagnostics and reporting to be performed.

The values for **diags_level** are as follows:

- `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 OP2 runtime. 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 declares a set.

Parameters

- **size** – Number of set elements.
- **name** – A name to be used for output diagnostics.

Returns A set ID.

op_map **op_decl_map**(op_set from, op_set to, int dim, int *imap, char *name)

This routine defines a mapping between sets.

Parameters

- **from** – Source set.
- **to** – Destination set.
- **dim** – Number of mappings per source element.
- **imap** – Mapping table.
- **name** – A name to be used for output diagnostics.

void **op_partition**(char *lib_name, char *lib_routine, op_set prime_set, op_map prime_map, op_dat coords)

This routine controls the partitioning of the sets used for distributed memory parallel execution.

Parameters

- **lib_name** – The partitioning library to use, see below.
- **lib_routine** – The partitioning algorithm to use. Required if using `"PTSCOTCH"` or `"PARMETIS"` as the **lib_name**.
- **prime_set** – Specifies the set to be partitioned.
- **prime_map** – Specifies the map to be used to create adjacency lists for the **prime_set**. Required if using `"KWAY"` or `"GEOMKWAY"`.
- **coords** – Specifies the geometric coordinates of the **prime_set**. Required if using `"GEOM"` or `"GEOMKWAY"`.

The current options for **lib_name** are:

- `"PTSCOTCH"`: The PT-Scotch library.
- `"PARMETIS"`: The ParMETIS library.
- `"INERTIAL"`: Internal 3D recursive inertial bisection partitioning.
- `"EXTERNAL"`: External partitioning optionally read in when using HDF5 I/O.
- `"RANDOM"`: Random partitioning, intended for debugging purposes.

The options for **lib_routine** when using `"PTSCOTCH"` are:

- `"KWAY"`: K-way graph partitioning.

The options for **lib_routine** when using `"PARMETIS"` are:

- `"KWAY"`: K-way graph partitioning.
- `"GEOM"`: Geometric graph partitioning.
- `"GEOMKWAY"`: Geometric followed by k-way graph partitioning.

void **op_decl_const**(int dim, char *type, T *dat)

This routine defines constant data with global scope that can be used in kernel functions.

Parameters

- **dim** – Number of data elements. For maximum efficiency this should be an integer literal.
- **type** – The type of the data as a string. This can be either intrinsic (`"float"`, `"double"`, `"int"`, `"uint"`, `"ll"`, `"ull"`, or `"bool"`) or user-defined.
- **dat** – A pointer to the data, checked for type consistency at run-time.

Note: If **dim** is 1 then the variable is available in the kernel functions with type `T`, otherwise it will be available with type `T*`.

Warning: If the executable is not preprocessed, as is the case with the development sequential build, then you must define an equivalent global scope variable to use the data within the kernels.

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

Parameters

- **set** – The set the data is associated with.
- **dim** – Number of data elements per set element.
- **type** – The datatype as a string, as with *op_decl_const()*. A qualifier may be added to control data layout - see *Dataset Layout*.
- **data** – Input data of type `T` (checked for consistency with **type** at run-time). The data must be provided in AoS form with each of the **dim** elements per set element contiguous in memory.
- **name** – A name to be used for output diagnostics.

Note: At present **dim** must be an integer literal. This restriction will be removed in the future but an integer literal will remain more efficient.

op_dat **op_decl_dat_temp**(op_set set, int dim, char *type, T *data, char *name)
Equivalent to *op_decl_dat()* but the dataset may be released early with *op_free_dat_temp()*.

void **op_free_dat_temp**(op_dat dat)
This routine releases a temporary dataset defined with *op_decl_dat_temp()*

Parameters

- **dat** – The dataset to free.

3.2.1 Dataset Layout

The dataset storage in OP2 can be configured to use either AoS (Array of Structs) or SoA (Struct of Arrays) layouts. As a default the AoS layout is used, matching what is supplied to *op_decl_dat()*, however depending on the access patterns of the kernels and the target hardware platform the SoA layout may perform favourably.

OP2 can be directed to ues SoA layout storage by setting the environment variable `OP_AUTO_SOA=1` prior to code translation, or by appending `:soa` to the type strings in the *op_decl_dat()* calls. The data supplied by the user should remain in the AoS layout.

3.3 Parallel Loops

void **op_par_loop**(void (*kernel)(...), char *name, op_set set, ...)

This routine executes a parallelised loop over the given **set**, with arguments provided by the *op_arg_gbl()*, *op_arg_dat()*, and *op_opt_arg_dat()* routines.

Parameters

- **kernel** – The kernel function to execute. The number of arguments to the kernel should match the number of op_arg arguments provided to this routine.
- **name** – A name to be used for output diagnostics.
- **set** – The set to loop over.
- **...** – The op_arg arguments passed to each invocation of the kernel.

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

This routine defines an op_arg that may be used either to pass non-constant read-only data or to compute a global sum, maximum or minimum.

Parameters

- **data** – Source or destination data array.
- **dim** – Number of data elements.
- **type** – The datatype as a string. This is checked for consistency with **data** at run-time.
- **acc** – The access type.

Valid access types for this routine are:

- 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 *type, op_access acc)

This routine defines an op_arg that can be used to pass a dataset either directly attached to the target op_set or attached to an op_set reachable through a mapping.

Parameters

- **dat** – The dataset.
- **idx** – The per-set-element index into the map to use. You may pass a negative value here to use a range of indicies - see below. This argument is ignored if the identity mapping is used.
- **map** – The mapping to use. Pass OP_ID for the identity mapping if no mapping indirection is required.
- **dim** – The dimension of the dataset, checked for consistency at run-time.
- **type** – The datatype of the dataset as a string, checked for consistency at run-time.
- **acc** – The access type.

Valid access types for this routine are:

- OP_READ: Read-only.
- OP_WRITE: Write-only.

- `OP_RW`: Read and write.
- `OP_INC`: Increment or global reduction to compute a sum.

The **idx** parameter accepts both positive values to specify a single per-element map index, where the kernel is passed a single dimension array of data, or negative values to specify a range of mapping indicies leading to the kernel being passed a two-dimensional array of data. If a negative index is provided the first **-idx** mapping indicies are provided to the kernel.

Consider the example of a kernel that is executed over a set of triangles, and is supplied the verticies via arguments. Using positive **idx** you would need one `op_arg` per vertex, leading to a kernel declaration similar to:

```
void kernel(float *v1, float *v2, float *v3, ...);
```

Alternatively, using a negative **idx** of `-3` allows a more succinct declaration:

```
void kernel(float **v[3], ...);
```

> **Warning:** `OP_WRITE` and `OP_RW` accesses *must not* have any potential data conflicts. This means that two different elements of the set cannot, through a map, reference the same elements of the dataset.
>
> Furthermore with `OP_WRITE` the kernel function *must* set the value of all **dim** components of the dataset. If this is not possible then `OP_RW` access should be specified.

Note: At present **dim** must be an integer literal. This restriction will be removed in the future but an integer literal will remain more efficient.

---

op_arg **op_opt_arg_dat**(op_dat dat, int idx, op_map map, int dim, char *type, op_access acc, int flag)
: This routine is equivalent to *op_arg_dat()* except for an extra **flag** parameter that governs whether the argument will be used (non-zero) or not (zero). This is intended to ease development of large application codes where many features may be enabled or disabled based on flags.

  The argument must not be dereferenced in the user kernel if **flag** is set to zero. If the value of the flag needs to be passed to the kernel then use an additional *op_arg_gbl()* argument.

3.4 HDF5 I/O

HDF5 has become the *de facto* 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. This is particularly useful for distributed memory MPI systems where the user would otherwise have to manually scatter data arrays over nodes prior to initialisation.

op_set **op_decl_set_hdf5**(char *file, char *name)
: Equivalent to *op_decl_set()* but takes a **file** instead of **size**, reading in the set size from the HDF5 file using the keyword **name**.

op_map **op_decl_map_hdf5**(op_set from, op_set to, int dim, char *file, char *name)
: Equivalent to *op_decl_map()* but takes a **file** instead of **imap**, reading in the mappiing table from the HDF5 file using the keyword **name**.

op_dat **op_decl_dat_hdf5**(op_set set, int dim, char *type, char *file, char *name)
Equivalent to *op_decl_dat()* but takes a **file** instead of **data**, reading in the dataset from the HDF5 file using the keyword **name**.

void **op_get_const_hdf5**(char *name, int dim, char *type, char *data, char *file)
This routine reads constant data from an HDF5 file.

Parameters

- **name** – The name of the dataset in the HDF5 file.
- **dim** – The number of data elements in the dataset.
- **type** – The string type of the data.
- **data** – A user-supplied array of at least **dim** capacity to read the data into.
- **file** – The HDF5 file to read the data from.

Note: To use the read data from within a kernel function you must declare it with *op_decl_const()*

Warning: The number of data elements specified by the **dim** parameter must match the number of data elements present in the HDF5 file.

3.5 MPI without HDF5 I/O

If you wish to use the MPI executables but don't want to use the OP2 HDF5 support, you may perform your own file I/O and then provide the data to OP2 using the normal routines. The behaviour of these routines under MPI is as follows:

- *op_decl_set()*: The **size** parameter is the number of elements provided by this MPI process.
- *op_decl_map()*: The **imap** parameter provides the part of the mapping table corresponding to the processes share of the **from** set.
- *op_decl_dat()*: The **data** parameter provides the part of the dataset corresponding to the processes share of the **set** set.

For example if an application has 4 processes, 4M nodes and 16M edges, then each process might be responsible for providing 1M nodes and 4M edges.

Note: This is effectively using simple contiguous block partitioning of the datasets, but it is important to note that this is strictly for I/O and this partitioning will not be used for the parallel computation. OP2 will re-partition the datasets, re-number the mapping tables and then shuffle the data between the MPI processes as required.

3.6 Other I/O and Utilities

void **op_printf**(const char *format, ...)
This routine wraps the standard `printf()` but only prints on the `MPI_ROOT` process.

void **op_fetch_data**(op_dat dat, T *data)
This routine copies data held in an `op_dat` from the OP2 backend into a user allocated memory buffer.

Parameters

- **dat** – The dataset to copy from.
- **data** – The user allocated buffer to copy into.

> **Warning:** The memory buffer provided by the user must be large enough to hold all elements in the `op_dat`.

void **op_fetch_data_idx**(op_dat dat, T *data, int low, int high)
This routine is equivalent to *op_fetch_data()* but with extra parameters to specify the range of data elements to fetch from the `op_dat`.

Parameters

- **dat** – The dataset to copy from.
- **data** – The user allocated buffer to copy into.
- **low** – The index of the first element to be fetched.
- **high** – The index of the last element to be fetched.

void **op_fetch_data_hdf5_file**(op_dat dat, const char *file_name)
This routine writes the data held in an `op_dat` from the OP2 backend into an HDF5 file.

Parameters

- **dat** – The source dataset.
- **file** – The name of the HDF5 file to write the dataset into.

void **op_print_dat_to_binfile**(op_dat dat, const char *file_name)
This routine writes the data held in an `op_dat` from the OP2 backend into a binary file.

Parameters

- **dat** – The source dataset.
- **file** – The name of the binary file to write the dataset into.

void **op_print_dat_to_txtfile**(op_dat dat, const char *file_name)
This routine writes the data held in an `op_dat` from the OP2 backend into a text file.

Parameters

- **dat** – The source dataset.
- **file** – The name of the text file to write the dataset into.

int **op_is_root**()
This routine allows a convenient way to test if the current process is the MPI root process.

Return values

- **1** – Process is the MPI root.

- **0** – Process is *not* the MPI root.

int **op_get_size**(op_set set)
This routine gets the global size of an op_set.

Parameters

- **set** – The set to query.

Returns The number of elements in the set across all processes.

void **op_dump_to_hdf5**(const char *file_name)
This routine dumps the contents of all op_sets, op_dats and op_maps to an HDF5 file *as held internally by OP2*, intended for debugging purposes.

Parameters

- **file_name** – The name of the HDF5 file to write the data into.

void **op_timers**(double *cpu, double *et)
This routine provides the current wall-clock time in seconds since the Epoch using gettimeofday().

Parameters

- **cpu** – Unused.
- **et** – A variable to hold the time.

void **op_timing_output**()
This routine prints OP2 performance details.

void **op_timings_to_csv**(const char *file_name)
This routine writes OP2 performance details to the specified CSV file. For MPI executables the timings are broken down by rank. For OpenMP executables with the OP_TIME_THREADS environment variable set, the timings are broken down by thread. For MPI + OpenMP executables with OP_TIME_THREADS set the timings are broken down per thread per rank.

Parameters

- **file_name** – The name of the CSV file to write.

void **op_diagnostic_output**()
This routine prints diagnostics relating to sets, mappings and datasets.

3.7 Executing with GPUDirect

OP2 supports execution with GPU direct MPI when using the MPI + CUDA builds. To enable this, simply pass -gpudirect as a command line argument when running the executable.

You may also have to user certain environment variables depending on MPI implementation, so check your cluster's user-guide.

CHAPTER

FOUR

IMPLEMENTATION EXAMPLE - AIRFOIL

The airfoil implementation example is currently available in PDF form here.

Warning: This document has not been updated for a significant amount of time; beware that the information contained may be out-of-date.

CHAPTER

FIVE

DEVELOPER GUIDE

The developer guide is currently available in PDF form here, with an extension document detailing the MPI implementation here.

These documents are intended for anyone looking to develop OP2, or looking for a deeper insight into the operational details. If you just wish to use OP2 in your project then the *OP2 C/C++ Manual* should suffice.

Warning: These documents have not been updated for a significant amount of time; beware that the information contained may be out-of-date.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search

INDEX

O