OP2 Documentation

Release latest

Gihan Mudalige, Istvan Reguly, Mike Giles

Oct 05, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 Overview

OP2 (Oxford Parallel library for Unstructured mesh solvers) 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 OP2 is or has been part supported by the Engineering and Physical Sciences Research Council (EPSRC) grants EP/I006079/1, EP/I00677X/1 on Multi-layered Abstractions for PDEs, EP/JOl0553/1 Algorithms and Software for Emerging Architectures (ASEArch), EP/S005072/1 Strategic Partnership in Computational Science for Advanced Simulation and Modelling of Engineering Systems - ASiMoV, the Royal Society through their Industry Fellowship Scheme (INF/R1/180012), the UK Technology Strategy Board and Rolls-Royce plc. through the SILOET project, The Janos Bolyai Research Scholarship of the Hungarian Academy of Sciences and the European Commission. We are also grateful for hardware resources during development from the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725, 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) KaHIP: 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 PTSCOTCH_INSTALL_PATH=<path/to/ptscotch>
export PARMETIS_INSTALL_PATH=<path/to/parmetis>
export KAHIP_INSTALL_PATH=<path/to/kahip>
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, ...}]
```

Make the build config, verifying that the compilers, libraries and flags are as you expect:

```
make -C op2 config
```

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

DEVELOPING AN OP2 APPLICATION

This page provides a tutorial in the basics of using OP2 for unstructured-mesh application development.

3.1 Example Application

The tutorial will use the Airfoil application, a simple non-linear 2D inviscid airfoil code that uses an unstructured mesh. It is a finite volume application that solves the 2D Euler equations using a scalar numerical dissipation. The algorithm iterates towards the steady state solution, in each iteration using a control volume approach - for example the rate at which the mass changes within a control volume is equal to the net flux of mass into the control volume across the four faces around the cell.

Airfoil consists of five loops, `save_soln` , `adt_calc` , `res_calc` , `bres_calc` and `update`, within a time-marching iterative loop. Out of these, `save_soln` and `update` are what we classify as direct loops where all the data accessed in the loop is defined on the mesh element over which the loop iterates over. Thus for example in a direct loop, a loop over edges will only access data defined on edges. The other three loops are indirect loops. In this case when looping over a given type of elements, data on other types of elements will be accessed indirectly, using mapping tables. Thus for example `res_calc` iterates over edges and increments data on cells, accessing them indirectly via a mapping table that gives the explicit connectivity information between edges and cells.

The standard mesh size solved with Airfoil consists of 1.5M edges. Here the most compute intensive loop is `res_calc`, which is called 2000 times during the total execution of the application and performs about 100 floating-point operations per mesh edge.

- Go to the `OP2/apps/c/airfoil/airfoil/airfoil_tutorial/original` directory and open the `airfoil_orig.cpp` file to view the original application.
- Use the Makefile in the same directory to compile and then run the application. The `new_grid.dat` (downloadable from here) needs to be present in the same directory as the executable.
- The program executes by reporting the rms value of the pressure held on the cells for every 100 iterations and ends by comparing this value to the reference value of the computation at 1000 iterations. If the solution is within machine precision of the reference value, the execution is considered to be validated (PASS). This is the same criteria required to validate any parallel versions of the application, including the paralleizations generated by OP2 as detailed below.

3.2 Original - Load mesh and initialization

The original code begins with allocating memory to hold the mesh data and then initializing them by reading in the mesh data, form the `new_grid.dat` text file:

```
FILE *fp;
if ((fp = fopen(FILE_NAME_PATH, "r")) == NULL) {
  printf("can't open file FILE_NAME_PATH\n");
  exit(-1);
}
if (fscanf(fp, "%d %d %d %d \n", &nnode, &ncell, &nedge, &nbedge) != 4) {
      printf("error reading from FILE_NAME_PATH\n");
      exit(-1);
}

cell   = (int *)malloc(4 * ncell  * sizeof(int));
edge   = (int *)malloc(2 * nedge  * sizeof(int));
ecell  = (int *)malloc(2 * nedge  * sizeof(int));
bedge  = (int *)malloc(2 * nbedge * sizeof(int));
becell = (int *)malloc(1 * nbedge * sizeof(int));
bound  = (int *)malloc(1 * nbedge * sizeof(int));
x      = (double *)malloc(2 * nnode * sizeof(double));
q      = (double *)malloc(4 * ncell * sizeof(double));
qold   = (double *)malloc(4 * ncell * sizeof(double));
res    = (double *)malloc(4 * ncell * sizeof(double));
adt    = (double *)malloc(1 * ncell * sizeof(double));

for (int n = 0; n < nnode; n++) {
      if (fscanf(fp, "%lf %lf \n", &x[2 * n], &x[2 * n + 1]) != 2) {
        printf("error reading from FILE_NAME_PATH\n");
        exit(-1);
      }
}

for (int n = 0; n < ncell; n++) {
  if (fscanf(fp, "%d %d %d %d \n", &cell[4 * n], &cell[4 * n + 1],
      &cell[4 * n + 2], &cell[4 * n + 3]) != 4) {
    printf("error reading from FILE_NAME_PATH\n");
    exit(-1);
  }
}
for (int n = 0; n < nedge; n++) {
      if (fscanf(fp, "%d %d %d %d \n", &edge[2 * n], &edge[2 * n + 1],
      &ecell[2 * n], &ecell[2 * n + 1]) != 4) {
    printf("error reading from FILE_NAME_PATH\n");
        exit(-1);
  }
}
for (int n = 0; n < nbedge; n++) {
  if (fscanf(fp, "%d %d %d %d \n", &bedge[2 * n], &bedge[2 * n + 1],
      &becell[n], &bound[n]) != 4) {
  printf("error reading from FILE_NAME_PATH\n");
  exit(-1);
```

(continues on next page)

(continued from previous page)

```
  }
}
fclose(fp);
```

The code then initialize `q` and `res` data arrays to 0.

3.3 Original - Main iteration and loops over mesh

The main iterative loop is a for loop that iterates for some `NUM_ITERATIONS` which in this case is set to 1000 iterations. Within this main iterative loops there are 5 loops over various mesh elements (as noted above) including direct and indirect loops.

3.4 Build OP2

Build OP2 using instructions in the Getting Started. page.

3.5 Step 1 - Preparing to use OP2

First, include the following header files, then initialize OP2 and finalize it as follows:

```
#include "op_seq.h"
...
...
int main(int argc, char **argv) {
  //Initialise the OP2 library, passing runtime args, and setting diagnostics level to␣
↪low (1)
  op_init(argc, argv, 1);
  ...
  ...
  ...
  free(adt);
  free(res);

  //Finalising the OP2 library
  op_exit();
}
```

By this point you need OP2 set up - take a look at the Makefile in step1, and observe that the include and library paths are added, and we link against `op2_seq` back-end library. Using the `op_seq.h` header file and linking with the sequential back-end OP2 lib produces what we call a *developer sequential* version of the application. This should be used to successively develop the rest of the application by using the OP2 API and validate its numerical output, as we do so in the next steps.

3.6 Step 2 - OP2 Declaration

Declare sets - The Airfoil application consists of four mesh element types (which we call sets): nodes, edges, cells and boundary edges. These needs to be declared using the `op_set` API call together with the number of elements for each of these sets:

```
// declare sets
op_set nodes  = op_decl_set(nnode,  "nodes" );
op_set edges  = op_decl_set(nedge,  "edges" );
op_set bedges = op_decl_set(nbedge, "bedges");
op_set cells  = op_decl_set(ncell,  "cells" );
```

Later, we will see how the number of mesh elements can be read in directly from an hdf5 file using the `op_set_hdf5` call.

When developing your own application with OP2, or indeed converting an application to use OP2, you will need to decide on what mesh element types, i.e. sets will need to be declared to define the full mesh. A good starting point for this design is to see what mesh elements are used the loops over the mesh.

Declare maps - Looking at the original Airfoil application's loops we see that mappings between edges and nodes, edges and cells, boundary edges and nodes, boundary edges and cells, and cells and nodes are required. This can be observed by the indirect access to data in each of the loops in the main iteration loops. These connectivity information needs to be declared via the `op_decl_map` API call:

```
//declare maps
op_map pedge   = op_decl_map(edges,  nodes, 2, edge,   "pedge"  );
op_map pecell  = op_decl_map(edges,  cells, 2, ecell,  "pecell" );
op_map pbedge  = op_decl_map(bedges, nodes, 2, bedge,  "pbedge" );
op_map pbecell = op_decl_map(bedges, cells, 1, becell, "pbecell");
op_map pcell   = op_decl_map(cells,  nodes, 4, cell,   "pcell"  );
```

The `op_decl_map` requires the names of the two sets for which the mapping is declared, its arity, mapping data (as in this case allocated in integer blocks of memory) and a string name.

Declare data - All data declared on sets should be declared using the `op_decl_dat` API call. For Airfoil this consists of the mesh coordinates data `x`, new and old solution `q` and `q_old`, area time step `adt`, flux residual `res` and boundary flag `bound` that indicates if the edge is a boundary edge:

```
//declare data on sets
op_dat p_bound = op_decl_dat(bedges, 1, "int",    bound, "p_bound");
op_dat p_x     = op_decl_dat(nodes,  2, "double", x,     "p_x"    );
op_dat p_q     = op_decl_dat(cells,  4, "double", q,     "p_q"    );
op_dat p_qold  = op_decl_dat(cells,  4, "double", qold,  "p_qold" );
op_dat p_adt   = op_decl_dat(cells,  1, "double", adt,   "p_adt"  );
op_dat p_res   = op_decl_dat(cells,  4, "double", res,   "p_res"  );
```

Declare constants - Finally global constants that are used in any of the computations in the loops needs to be declared. This is required due to the fact that when using code-generation later for parallelizations such as on GPUs (e.g. using CUDA), global constants needs to be copied over to the GPUs before they can be used in a GPU kernel. Declaring them using the `op_decl_const` API call will indicate to the OP2 code-generator that these constants needs to be handled in a special way, generating code for copying them to the GPU for the relevant back-ends.

```
//declare global constants
op_decl_const(1, "double", &gam  );
op_decl_const(1, "double", &gm1  );
```

(continues on next page)

(continued from previous page)

```
op_decl_const(1, "double", &cfl  );
op_decl_const(1, "double", &eps  );
op_decl_const(1, "double", &alpha);
op_decl_const(4, "double", qinf  );
```

Finally information about the the declared mesh can be viewed using a diagnostics level of 2 in `op_init` and calling `op_diagnostic_output()` API call:

```
/* main application */
int main(int argc, char **argv) {
//Initialise the OP2 library, passing runtime args, and setting diagnostics level to low␣
↪(1)
op_init(argc, argv, 2);
...
... op_decl_set ...
... op_decl_map ...
... op_decl_dat ...
... op_decl_const  ...
...
//output mesh information
op_diagnostic_output();
```

Finally compile the step2 application and execute using the following runtime flag:

```
./airfoil_step2 OP_NO_REALLOC
```

The `OP_NO_REALLOC` runtime flag instructs to the OP2 back-end to simply use the already allocated memory for sets, maps and data in the declaration, without internally de-allocating them. This helps the developer to gradually build up the application with the conversion to OP2 API (as we are do here), checking for validation on each step. However, this behavior is only specified for the developer sequential version, which we are developing here. None of the parallel versions generated via the code generator nor the code generated sequential version `gen_seq` will work as they de-allocate the initial memory and move the mesh to obtain best parallel performance.

3.7 Step 3 - First parallel loop : direct loop

We can now convert the first loop to use the OP2 API. In this case its a direct loop called `save_soln` that iterates over cells and saves the previous time-iteration's solution, `q` to `q_old`:

```
//save_soln : iterates over cells
for (int iteration = 0; iteration < (ncell * 4); ++iteration) {
  qold[iteration] = q[iteration];
}
```

This is a direct loops due to the fact that all data accessed in the computation are defined on the set that the loop iterates over. In this case the iteration set is cells.

To convert to the OP2 API we first outline the loop body (elemental kernel) to a subroutine:

```
//outlined elemental kernel
inline void save_soln(double *q, double *qold) {
for (int n = 0; n < 4; n++)
  qold[n] = q[n];
```

(continues on next page)

(continued from previous page)

```
}

//save_soln : iterates over cells
for (int iteration = 0; iteration < (ncell * 4); ++iteration) {
  save_soln(&q[iteration], &qold[iteration]);
}
```

Now we can directly declare the loop with the `op_par_loop` API call:

```
op_par_loop(save_soln, "save_soln", cells,
            op_arg_dat(p_q,    -1, OP_ID, 4, "double", OP_READ ),
            op_arg_dat(p_qold, -1, OP_ID, 4, "double", OP_WRITE));
```

Note how we have:

- indicated the elemental kernel `save_soln` in the first argument to `op_par_loop`
- used the `op_dat``s names ``p_q` and `p_qold` in the API call
- noted the iteration set `cells` (3rd argument)
- indicated the direct access of `q` and `q_old` using `OP_ID`
- indicated that `p_q` is read only (`OP_READ`) and `q_old` is written to only (`OP_WRITE`), by looking through the elemental kernel and identifying how they are used/accessed in the kernel.
- given that `p_q` is read only we also indicate this by the key word `const` for `save-soln` elemental kernel.
- The fourth argument of an `op_arg_dat` is the dimension of the data. For `p_q` and `p_qold` there are 4 doubles per mesh point.

Compile and execute (again using `OP_NO_REALLOC`) the modified application (see code in `../step3`) and check if the solution validates.

3.8 Step 4 - Indirect loops

The next loop in the application `adt_calc` calculate area/timstep and iterates over cells. In this case we see that the loop is an indirect loop where the data `x` on the four nodes connected to a cell is accessed indirectly via a cells to nodes mapping. Additionally data `adt` are accessed directly where `adt` is data on the cells:

```
//adt_calc - calculate area/timstep : iterates over cells
for (int iteration = 0; iteration < ncell; ++iteration) {
  int map1idx = cell[iteration * 4 + 0];
  int map2idx = cell[iteration * 4 + 1];
  int map3idx = cell[iteration * 4 + 2];
  int map4idx = cell[iteration * 4 + 3];

  double dx, dy, ri, u, v, c;

  ri = 1.0f / q[4 * iteration + 0];
  u = ri * q[4 * iteration + 1];
  v = ri * q[4 * iteration + 2];
  c = sqrt(gam * gm1 * (ri * q[4 * iteration + 3] - 0.5f * (u * u + v * v)));

  dx = x[2 * map2idx + 0] - x[2 * map1idx + 0];
```

(continues on next page)

(continued from previous page)

```
  dy = x[2 * map2idx + 1] - x[2 * map1idx + 1];
  adt[iteration] = fabs(u * dy - v * dx) + c * sqrt(dx * dx + dy * dy);

  dx = x[2 * map3idx + 0] - x[2 * map2idx + 0];
  dy = x[2 * map3idx + 1] - x[2 * map2idx + 1];
  adt[iteration] += fabs(u * dy - v * dx) + c * sqrt(dx * dx + dy * dy);

  dx = x[2 * map4idx + 0] - x[2 * map3idx + 0];
  dy = x[2 * map4idx + 1] - x[2 * map3idx + 1];
  adt[iteration] += fabs(u * dy - v * dx) + c * sqrt(dx * dx + dy * dy);

  dx = x[2 * map1idx + 0] - x[2 * map4idx + 0];
  dy = x[2 * map1idx + 1] - x[2 * map4idx + 1];
  adt[iteration] += fabs(u * dy - v * dx) + c * sqrt(dx * dx + dy * dy);

  adt[iteration] = (adt[iteration]) / cfl;
}
```

Similar to the direct loop, we outline the loop body and call it within the loop as follows:

```
//outlined elemental kernel - adt_calc
inline void adt_calc(double *x1, double *x2, double *x3,
                     double *x4, double *q, double *adt) {
  double dx, dy, ri, u, v, c;

  ri = 1.0f / q[0];
  u = ri * q[1];
  v = ri * q[2];
  c = sqrt(gam * gm1 * (ri * q[3] - 0.5f * (u * u + v * v)));

  dx = x2[0] - x1[0];
  dy = x2[1] - x1[1];
  *adt = fabs(u * dy - v * dx) + c * sqrt(dx * dx + dy * dy);

  dx = x3[0] - x2[0];
  dy = x3[1] - x2[1];
  *adt += fabs(u * dy - v * dx) + c * sqrt(dx * dx + dy * dy);

  dx = x4[0] - x3[0];
  dy = x4[1] - x3[1];
  *adt += fabs(u * dy - v * dx) + c * sqrt(dx * dx + dy * dy);

  dx = x1[0] - x4[0];
  dy = x1[1] - x4[1];
  *adt += fabs(u * dy - v * dx) + c * sqrt(dx * dx + dy * dy);

  *adt = (*adt) / cfl;
}

//adt_calc - calculate area/timstep : iterates over cells
for (int iteration = 0; iteration < ncell; ++iteration) {
  int map1idx = cell[iteration * 4 + 0];
```

(continues on next page)

(continued from previous page)

```
  int map2idx = cell[iteration * 4 + 1];
  int map3idx = cell[iteration * 4 + 2];
  int map4idx = cell[iteration * 4 + 3];

  adt_calc(&x[2 * map1idx], &x[2 * map2idx], &x[2 * map3idx],
           &x[2 * map4idx], &q[4 * iteration], &adt[iteration]);
}
```

Now, convert the loop to use the `op_par_loop` API:

```
//res_calc - calculate flux residual: iterates over edges
op_par_loop(adt_calc, "adt_calc", cells,
            op_arg_dat(p_x,   0, pcell, 2, "double", OP_READ ),
            op_arg_dat(p_x,   1, pcell, 2, "double", OP_READ ),
            op_arg_dat(p_x,   2, pcell, 2, "double", OP_READ ),
            op_arg_dat(p_x,   3, pcell, 2, "double", OP_READ ),
            op_arg_dat(p_q,  -1, OP_ID, 4, "double", OP_READ ),
            op_arg_dat(p_adt,-1, OP_ID, 1, "double", OP_WRITE));
```

Note in this case how the indirections are specified using the mapping declared as OP2 map `pcell`, indicating the to-set index (2nd argument), and access mode `OP_READ`.

3.9 Step 5 - Global reductions

At this stage almost all the remaining loops can be converted to the OP2 API. Only the final loop *update* that updates the flow field needs special handling due to its global reduction:

```
rms = 0.0f;
for (int iteration = 0; iteration < ncell; ++iteration) {
  double del, adti;

  adti = 1.0f / (adt[iteration]);

  for (int n = 0; n < 4; n++) {
    del = adti * res[iteration * 4 + n];
    q[iteration * 4 + n] = qold[iteration * 4 + n] - del;
    res[iteration * 4 + n] = 0.0f;
    rms += del * del;
  }
}
```

Here, the global variable `rms` is used as a reduction variable to compute the rms value of the residual. The kernel can be outlined as follows:

```
//outlined elemental kernel - update
inline void update(const double *qold, double *q, double *res,
                   const double *adt, double *rms) {
  double del, adti;

  adti = 1.0f / (*adt);

```

(continues on next page)

(continued from previous page)

```
  for (int n = 0; n < 4; n++) {
    del = adti * res[n];
    q[n] = qold[n] - del;
    res[n] = 0.0f;
    *rms += del * del;
  }
}
```

And then call it in the application:

```
//update = update flow field - iterates over cells
rms = 0.0f;
for (int iteration = 0; iteration < ncell; ++iteration) {
  update(&qold[iteration * 4], &q[iteration * 4], &res[iteration * 4],
         &adt[iteration], &rms);
}
```

The global reduction requires the op_arg_gbl API call with OP_INC access mode to indicate that the variable is a global reduction:

```
//update = update flow field - iterates over cells
rms = 0.0f;
op_par_loop(update, "update", cells,
            op_arg_dat(p_qold, -1, OP_ID, 4, "double", OP_READ ),
            op_arg_dat(p_q,    -1, OP_ID, 4, "double", OP_WRITE),
            op_arg_dat(p_res,  -1, OP_ID, 4, "double", OP_RW   ),
            op_arg_dat(p_adt,  -1, OP_ID, 1, "double", OP_READ ),
            op_arg_gbl(&rms,    1,           "double", OP_INC  ));
```

At this point all the loops have been converted to use op_par_loop API and the application should be validating when executed on as sequential, single threaded CPU application. You should now also be able to run without the OP_NO_REALLOC runtime flag and still get a valid result. However, in this case you should note that OP2 will be making internal copies of the data declared for op_map and op_dat . When developing applications for performance, you should consider freeing the initial memory allocated immediately after the relevant op_decl_map and op_decl_dat calls. In the next step we avoid freeing such "application developer allocated" memory by using HDF5 file I/O so that mesh data is directly read from file to OP2 allocated internal memory.

3.10 Step 6 - Handing it all to OP2

Once the developer sequential version has been created and the numerical output validates the application can be prepared to obtain a developer distributed memory parallel version. This step can be completed to obtain a parallel executable, without code-generation if the following steps are implemented.

(1) File I/O needs to be extended to allow distributed memory execution with MPI. The current Airfoil application simply reads the mesh data from a text file and such a simple setup will not be workable on a distributed memory system, such as a cluster and more importantly will not be scalable with MPI. The simplest solution is to use OP2's HDF5 API for declaring the mesh by replacing op_decl_set, op_decl_map, op_decl_dat and op_decl_const by its HDF5 counterparts as follows:

```
// declare sets
op_set nodes  = op_decl_set_hdf5(file,  "nodes" );
```

(continues on next page)

(continued from previous page)

```
op_set edges  = op_decl_set_hdf5(file,  "edges" );
op_set bedges = op_decl_set_hdf5(file, "bedges");
op_set cells  = op_decl_set_hdf5(file,  "cells" );

//declare maps
op_map pedge   = op_decl_map_hdf5(edges,  nodes, 2, file, "pedge"  );
op_map pecell  = op_decl_map_hdf5(edges,  cells, 2, file, "pecell" );
op_map pbedge  = op_decl_map_hdf5(bedges, nodes, 2, file, "pbedge" );
op_map pbecell = op_decl_map_hdf5(bedges, cells, 1, file, "pbecell");
op_map pcell   = op_decl_map_hdf5(cells,  nodes, 4, file, "pcell"  );

//declare data on sets
op_dat p_bound = op_decl_dat_hdf5(bedges, 1, "int",    file, "p_bound");
op_dat p_x     = op_decl_dat_hdf5(nodes,  2, "double", file, "p_x"    );
op_dat p_q     = op_decl_dat_hdf5(cells,  4, "double", file, "p_q"    );
op_dat p_qold  = op_decl_dat_hdf5(cells,  4, "double", file, "p_qold" );
op_dat p_adt   = op_decl_dat_hdf5(cells,  1, "double", file, "p_adt"  );
op_dat p_res   = op_decl_dat_hdf5(cells,  4, "double", file, "p_res"  );

//read and declare global constants
op_get_const_hdf5("gam",   1, "double", (char *)&gam,  file);
op_get_const_hdf5("gm1",   1, "double", (char *)&gm1,  file);
op_get_const_hdf5("cfl",   1, "double", (char *)&cfl,  file);
op_get_const_hdf5("eps",   1, "double", (char *)&eps,  file);
op_get_const_hdf5("alpha", 1, "double", (char *)&alpha,file);
op_get_const_hdf5("qinf",  4, "double", (char *)&qinf, file);

op_decl_const(1, "double", &gam  );
op_decl_const(1, "double", &gm1  );
op_decl_const(1, "double", &cfl  );
op_decl_const(1, "double", &eps  );
op_decl_const(1, "double", &alpha);
op_decl_const(4, "double", qinf  );
```

Note here that we assume that the mesh is already available as an HDF5 file named `new_grid.h5`. (See the `convert_mesh.cpp` utility application in `OP2-Common/apps/c/airfoil/airfoil_hdf5/dp` to understand how we can create an HDF5 file to be compatible with the OP2 API for Airfoil starting from mesh data defined in a text file.)

When the application has been switched to use the HDF5 API calls, manually allocated memory for the mesh elements can be removed. Additionally all `printf` statements should use `op_printf` so that output to terminal will only be done by the ROOT mpi process. We can also replace the timer routines with OP2's `op_timers` which times the execution of the code the ROOT.

Given that the mesh was read via HDF5, to obtain the global sizes of the mesh, OP2's `op_get_size()` API call need to be used. This is required for the Airfoil application to obtain the number of cells to compute the rms value for every 100 iterations to validate the application:

```
//get global number of cells
ncell = op_get_size(cells);
```

(2) Add the OP2 partitioner call `op_partition` to the code in order to signal to the MPI back-end, the point in the program that all mesh data have been defined and mesh can be partitioned and MPI halos can be created:

```
...
...
op_decl_const(1, "double", &alpha);
op_decl_const(4, "double", qinf  );

//output mesh information
op_diagnostic_output();

//partition mesh and create mpi halos
op_partition("BLOCK", "ANY", edges, pecell, p_x);

...
...
```

See the API documentation for practitioner options. In this case no special partitioner is used leaving the initial block partitioning of data at the time of file I/O through HDF5.

Take a look at the code in the `/step6` for the full code changes done to the Airfoil application. The application can now be compiled to obtain a developer distributed-memory (MPI) parallel executable using the Makefile in the same directory. Note how the executable is created by linking with the OP2 MPI back-end, `libop2_mpi` together with the HDF5 library `libhdf5`. You will need to have had HDF5 library installed on your system to carry out this step.

The resulting executable is called a `developer MPI` version of the application, which should again be used to verify validity of the application by running with `mpirun` in the usual way of executing an MPI application.

3.11 Step 7 - Code generation

Now that both the sequential and MPI developer versions work and validate, its time to generate other parallel versions. However, first we should move the elemental kernels to header files so that after the code generation the modified main application will not have the same elemental kernel definitions. This is currently a limitation of the code-generator, which will be remedied in future versions.

We move the elemental kernels to header files, each with the name of the kernel and include them in the `airfoil_step7.cpp` main file:

```
...
...
/* Global Constants */
double gam, gm1, cfl, eps, mach, alpha, qinf[4];

//
// kernel routines for parallel loops
//
#include "adt_calc.h"
#include "bres_calc.h"
#include "res_calc.h"
#include "save_soln.h"
#include "update.h"
...
```

Now the code is ready for code-generation, go to the `/step7` directory and on the terminal type :

```
python $OP2_INSTALL_PATH/../translator/c/op2.py airfoil_step7.cpp
```

Note that the `python` command assumes that it will point to Python 3.* or above. This will then generate (in the same directory) parallel code under sub-directories - `cuda, openacc, openmp, openmp4, seq` and `vec`. They correspond to on-node parallel version based on CUDA, OpenACC, OpenMP, OpenMP4.0 (and higher) and SIMD vectorized, respectively. Each of them can also run in combination with distributed memory parallelization using MPI across nodes.

The Makefile in `/step7` simply uses the OP2 supplied makefiles:

```
APP_NAME := airfoil_step7

APP_ENTRY := $(APP_NAME).cpp
APP_ENTRY_MPI := $(APP_ENTRY)

OP2_LIBS_WITH_HDF5 := true

include ../../../../../makefiles/common.mk
include ../../../../../makefiles/c_app.mk
```

This will build all the currently supported parallel versions, provided that the supporting libraries are available on your system and the relevant OP2 back-end libraries have been built.

3.12 Final - Code generated versions and execution

The following parallel versions will be generated from the code generator. These and the previously developed *developer* versions and can be executed as follows (see the /final directory in the OP2-APPS repository for all the generated code) :

(1) Developer sequential and developer mpi - no code-generation required.

```
#developer sequential
./aifroil_seq

#developer distributed memory with mpi, on 4 mpi procs
$MPI_INSTAL_PATH/bin/mpirun -np 4 ./airfoil_mpi_seq
```

(2) Code-gen sequential and MPI + Code-gen sequential

```
# code-gen sequential
./aifroil_genseq
#On 4 mpi procs
$MPI_INSTAL_PATH/bin/mpirun -np 4 ./aifroil_mpi_genseq
```

(3) Code-gen OpenMP, on 4 OpenMP threads, with mini-partition size of 256 and MPI + Code-gen OpenMP, on 4 MPI x 8 OpenMP with with mini-partition size of 256

```
# on 4 OMP threads
export OMP_NUM_THREADS=4; ./aifroil_openmp OP_PART_SIZE=256
#On 4 mpi procs with each proc running 8 OpenMP threads
export OMP_NUM_THREADS=8; $MPI_INSTAL_PATH/bin/mpirun -np 4 ./aifroil_mpi_openmp with␣
↪mini-partition size of 256
```

(4) Code-gen SIMD vectorized and MPI + Code-gen SIMD vectorized, on 4 MPI

```
#SIMD vec
./aifroil_vec
#On 4 mpi procs with each proc running SIMD vec
$MPI_INSTAL_PATH/bin/mpirun -np 4 ./aifroil_mpi_vec
```

(5) Code-gen CUDA with mini-partition size of 128 and CUDA thread-block size of 192 and MPI + Code-gen CUDA

```
#On a single GPU
./airfoil_cuda OP_PART_SIZE=128 OP_BLOCK_SIZE=192
#On 4 mpi procs, each proc having a GPU
$MPI_INSTALL_PATH/bin/mpirun -np 4 ./airfoil_mpi_cuda OP_PART_SIZE=128 OP_BLOCK_SIZE=192
```

(6) MPI + Code-gen hybrid CUDA with mini-partition size of 128 and CUDA thread-block size of 192

The hybrid version can be run on both CPUs and GPUs at the same time. If there is only 4 GPUs are available the following execution will allocated 4 MPI procs to be run on 4 GPUs and 8 MPI procs allocated to the remaining CPU cores.

```
#On 12 mpi procs
$MPI_INSTALL_PATH/bin/mpirun -np 12 ./airfoil_mpi_cuda_hyb OP_PART_SIZE=128 OP_BLOCK_
↪SIZE=192
```

(7) Code-gen OpenACC with mini-partition size of 128 and thread-block size of 192 and MPI + Code-gen OpenACC

```
#On a single GPU
./airfoil_openacc OP_PART_SIZE=128 OP_BLOCK_SIZE=192
#On 4 mpi procs, each proc having a GPU
$MPI_INSTALL_PATH/bin/mpirun -np 4 ./airfoil_mpi_openacc OP_PART_SIZE=128 OP_BLOCK_
↪SIZE=192
```

3.13 Optimizations

See the *Performance Tuning* section for a number of specific compile-time and runtime flags to obtain better performance.

CHAPTER

FOUR

OP2 API

4.1 Overview

The key characteristic of unstructured-mesh applications is the explicit connectivity required to specify the mesh and access data (indirectly) on neighboring mesh elements during computation over mesh elements. As such OP2 allows to describe an unstructured mesh by a number of sets, where 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 (i.e. explicit connectivity, 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.

4.2 Key Concepts and Structure

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 at a high-level API declaring the unstructured-mesh problem. Then the OP2 code-generator can be used to automatically generate a number of different paralleizations, using different parallel programing models, targeting modern multi-core and many-core hardware. The most mature parallel code generated by OP2 are :

(1) Single-threaded on a CPU including code parallelized using SIMD

(2) Multi-threaded using OpenMP for multicore CPU systems

(3) Parallelized using CUDA for NVIDIA GPUs.

Further to these there are also in-development versions that can emit SYCL and AMD HIP for parallelization on a wider range of GPUs. All of the above can be combined with distributed-memory MPI parallelization to run of clusters of CPU or GPU nodes. 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.

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 code-generator 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 code-generator is needed to generate efficient single-threaded (see 1. above) , SIMD and OpenMP code for CPU systems.

Fig. 4.1 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.

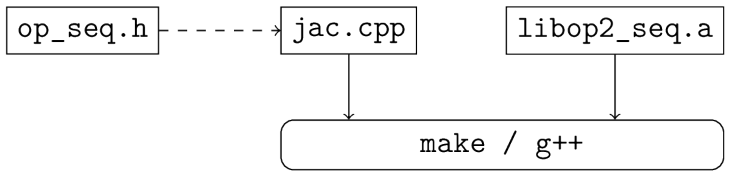

Fig. 4.1: Build process for a single-threaded development executable.

Fig. 4.2 shows the build process for the corresponding CUDA executable. The code-generator 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.

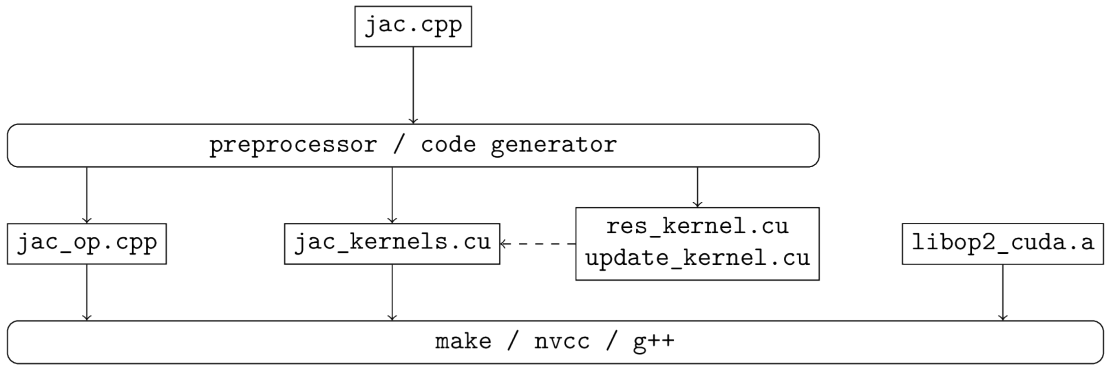

Fig. 4.2: Build process for a CUDA accelerated executable.

Fig. 4.3 shows the OpenMP build process which is very similar to the CUDA process except that it uses `*.cpp` files produced by the code-generator instead of `*.cu` files.

In looking at the API specification, users may think it is somewhat 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 code-generator, 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 code-generator or library will check that all redundant information is self-consistent. If you declare

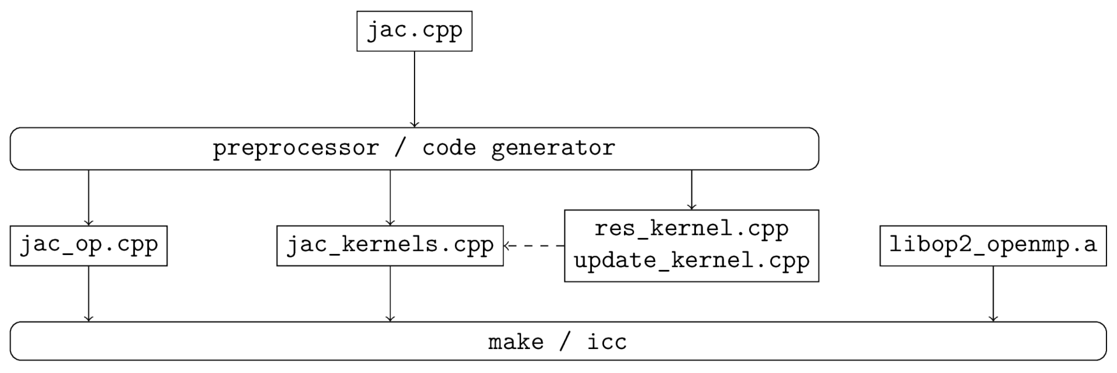

Fig. 4.3: Build process for an OpenMP accelerated executable.

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.

4.3 OP2 C/C++ API

4.3.1 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"`, `"PARMETIS"` or https://kahip.github.io/ 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.
- `"KAHIP"`: The KaHIP 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"` or `"KAHIP"` 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.

4.3.2 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.

4.3.3 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.

4.3.4 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.

4.3.5 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.

4.3.6 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.

CHAPTER FIVE

EXAMPLES

See the OP2-APPS repository to see the latest generated parallel code for each application.

5.1 Airfoil

- The Airfoil mesh can be downloaded from here. or generated from Matlab using the mesh generators in `OP2-Common/apps/mesh_generators/naca0012.m`
- An older version of the airfoil implementation example is archived 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

SIX

PERFORMANCE TUNING

6.1 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.

6.2 OpenMP and OpenMP+MPI

It is recommended that you assign one MPI rank per NUMA region when executing MPI+OpenMP parallel code. Usually for a multi-CPU system a single CPU socket is a single NUMA region. Thus, for a 4 socket system, OP2's MPI+OpenMP code should be executed with 4 MPI processes with each MPI process having multiple OpenMP threads (typically specified by the `OMP_NUM_THREAD` flag). Additionally on some systems using `numactl` to bind threads to cores could give performance improvements.

CHAPTER

SEVEN

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++ API* 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.

7.1 Contributing

To contribute to OP2 please use the following steps :

1. Clone the OP2 repository (on your local system).
2. Create a new branch in your cloned repository
3. Make changes / contributions in your new branch
4. Submit your changes by creating a Pull Request to the `master` branch of the OPS repository

Cumulated contributions in the `master` branch will be included in a new release.

CHAPTER EIGHT

PUBLICATIONS

See OP-DSL publications page.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- search

INDEX