OP2 User's Manual (phase 1)

Mike Giles

May 30, 2011

1 Introduction

OP2 is an API with associated libraries and preprocessors to generate parallel executables for applications on unstructured grids. The initial API is for C++, but C99 and FORTRAN 90 will also be supported.

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. 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 platforms:

- single-threaded on a CPU
- parallelised using CUDA for NVIDIA GPUs
- multi-threaded using OpenMP for multicore x86 systems

In the longer-term there will be support for AVX vectorisation for x86 CPUs, and OpenCL for both CPUs and GPUS.

There will also be support for distributed-memory MPI parallelisation in combination with any of the above. This will require parallel file I/O and so there will be routines to handle file I/O for the main datasets, as well as routines to handle terminal I/O.

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 one.

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, the single-threaded executable does not use 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 OpenMP code for x86 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 OP header file `op_seq.h` and is linked to the OP routines in `op_seq.c` 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 routines in `op_lib.cu` 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, both in the single-threaded library and in the CUDA library.

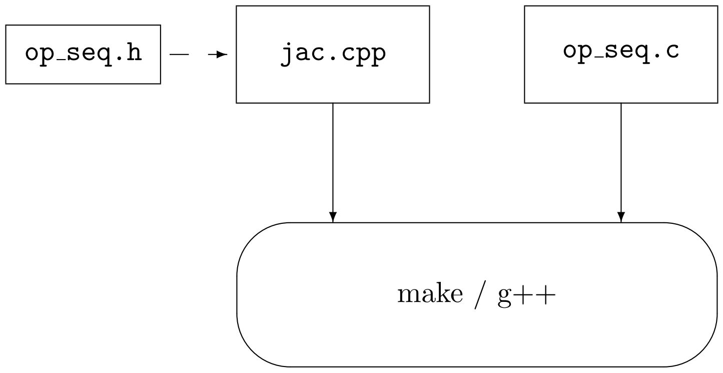

Figure 1: Sequential code build process

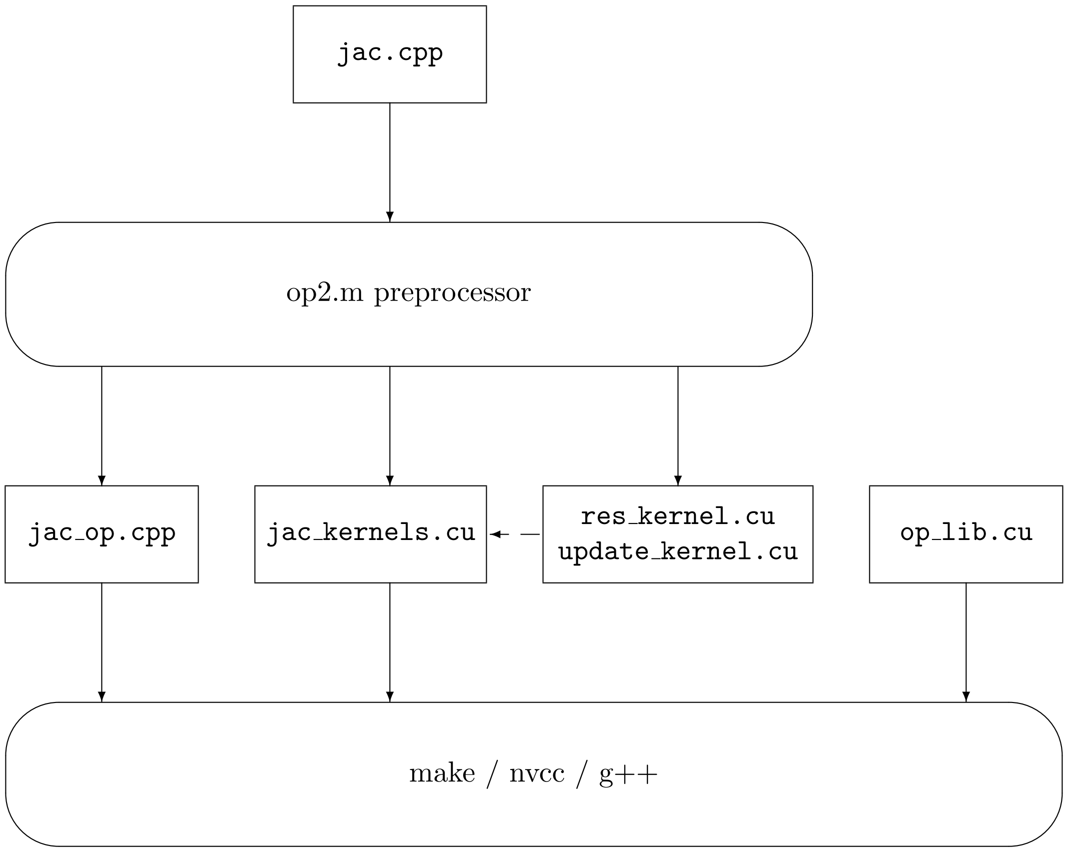

Figure 2: CUDA code build process

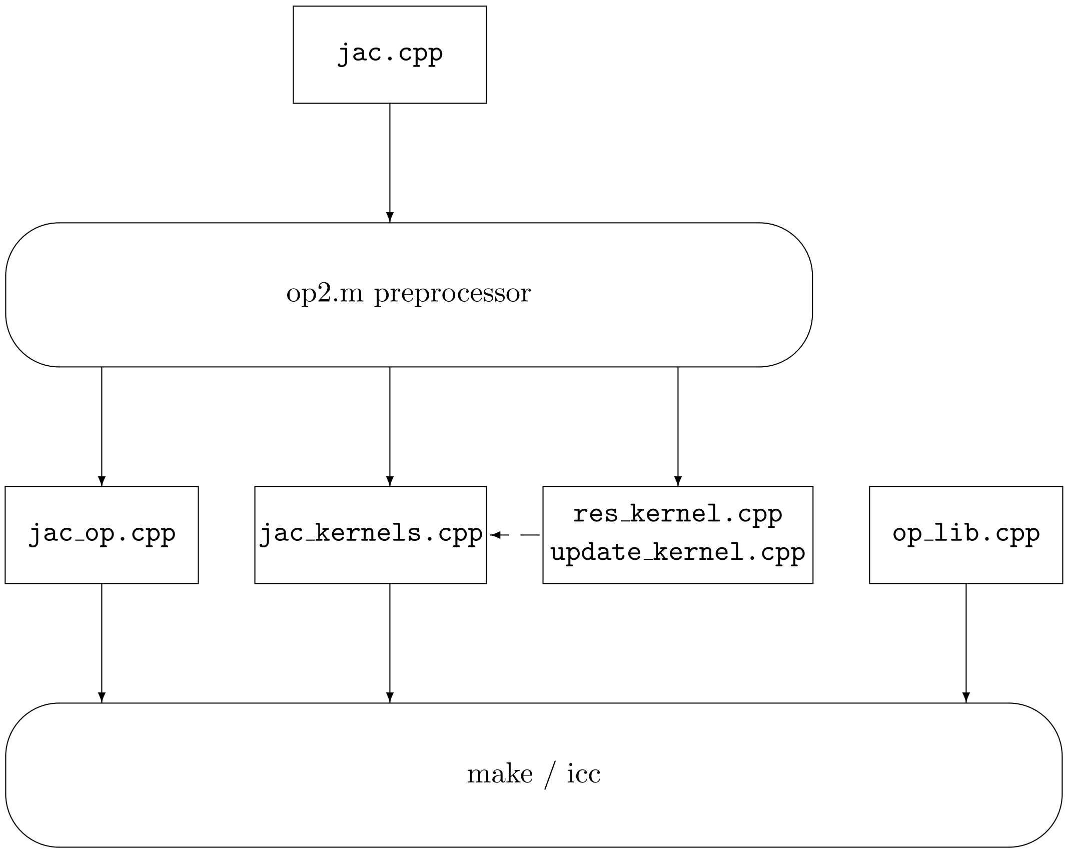

Figure 3: OpenMP code build process

3 Initialisation and termination routines

void op_init(int argc, char **argv, int diags_level)

This routine must be called before all other OP routines.

`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.

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) 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 ("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
`data`	input data of type `T` (checked for consistency with `type` at run-time)
`name`	a name used for output diagnostics

void op_fetch_data(op_dat dat)

This routine transfers data from the GPU back to the CPU.

`dat`	OP dataset ID – data is put back into original input array

void op_diagnostic_output()

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

4 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 (-1 $\equiv$ no mapping indirection)
`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.

5 User-defined datatypes

If the user defines a new datatype `mytype` it must be included in a header file along with

- a type-checking routine:

  ```
  inline int type_error(const mytype *,const char *type)
  {return strcmp(type,"mytype");}
  ```

  which is used at run-time to check the consistency of the user's type declarations in input arguments.

- a "zero element" declaration of the form:

  ```
  #define ZERO_mytype 0;
  ```

  as well as an appropriate overloaded addition operator if there is any `OP_INC` access to the datatype. The zero element and overloaded addition have to be such that $0 + x = x$ where x represents any element of the user's datatype and 0 represents the declared zero element.

- an overloaded implementation of the inequality operators $<$ and $>$ if there are any `OP_MIN, OP_MAX` accesses to the datatype.

In addition, the user must specify the name of the new header file using the environment variable `OP_USER_DATATYPES` so that this header file is included into the OP2 header file `op_datatypes.h`.

6 Preprocessor

The prototype preprocessor has been written in MATLAB. It 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`, and 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.

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` and the individual kernel files.

The preprocessor cannot currently handle cases in which the same user kernel is used in more than one parallel loop, or when global constant data is set/updated in more than one place within the code. This will be addressed in the future.

7 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.

8 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.

9 Phase 2 proposal

As explained in the introduction, phase 2 of the OP2 project will handle distributed-memory parallelisation using MPI. Because this links into other work by Leigh Lapworth and others at Rolls-Royce, discussions have begun about how this will be handled within OP2, and this has led to the following proposal.

My starting point is that we anticipate dealing with extremely large datasets and so we need to support parallel file I/O. There also seems to be general agreement that HDF5 has become the *de facto* standard underlying file format, with various other standards like CGNS layered on top.

Originally, my idea was to modify the OP2 set, mapping and dataset declarations so that these were read in by OP2 from a specified HDF5 file using specified keywords. Thus the OP2 library would have been entirely responsible for the parallel file I/O.

However, my new proposal is to adopt a layered approach:

- a minor extension to the existing API, leaving the parallel file I/O to the developer
- an example implementation of the parallel file I/O for HDF files, which some developers may choose to use unaltered, and others may modify to suit their needs

The rationale for this is to allow developers to make the tradeoff between ease-of-use and flexibility. Some will want maximum ease-of-use and are prepared to pay the price of working with HDF5 files with the flat keyword-based hierarchy which we will assume. Others will want the flexibility to manage their data storage in the way they wish, and will accept the additional programming effort this will entail.

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. The extensions to the existing API are 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, 4×10^6 nodes and 16×10^6 edges, then each process might be responsible for providing 10^6 nodes and 4×10^6 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 (in parallel, probably using `parmetis` or `PT-Scotch`), will 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.

The second layer would look similar to the existing API:

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