OP2 User's Manual (phase 1)

Mike Giles

June 15, 2010

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 FORTRAN 77 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, triangular faces, quadrilateral faces, cells of a variety of types, far-field boundary nodes, wall boundary faces, etc. Associated with these sets are both data (e.g. coordinate data at nodes) and pointers to other sets (e.g. edge pointers 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 pointer.

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 in the current implementation are that the sets and pointers are static (i.e. they do not change) and the operands in the set operations are not referenced through a double level of pointer indirection (i.e. through a pointer to another set which in turn uses another pointer to data in a third set).

In the long-term, this library will allow users to write a single program which can be built into a variety of different executables for different platforms:

- single-threaded on a CPU
- multi-threaded / vectorised on a CPU using OpenMP and/or SSE/AVX vectors
- parallelised on GPUs using CUDA or OpenCL
- distributed-memory MPI parallelisation in combination with any of the above

The higher-level distributed-memory MPI parallelisation 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.

However, the initial version described in this document is for execution on a shared-memory system, and the initial implementation is for CUDA execution on a single GPU.

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 will be needed to generate the executable for GPUs. 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. A new preprocessor (which has not yet been developed) will be needed to get efficient OpenMP/SSE/AVX execution on CPUs.

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.

As well as the header file `op_seq.h` which is included by the user's main code `jac.cpp`, there is a header file `op_datatypes.h` which is included by all of the files in the CUDA implementation, and by `op_seq.h`.

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 the 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.

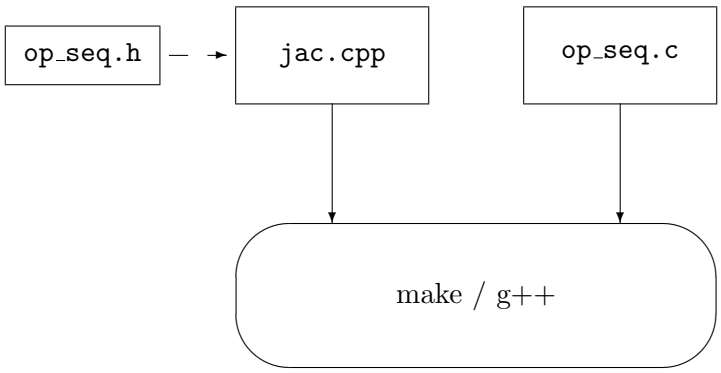

Figure 1: Sequential code build process

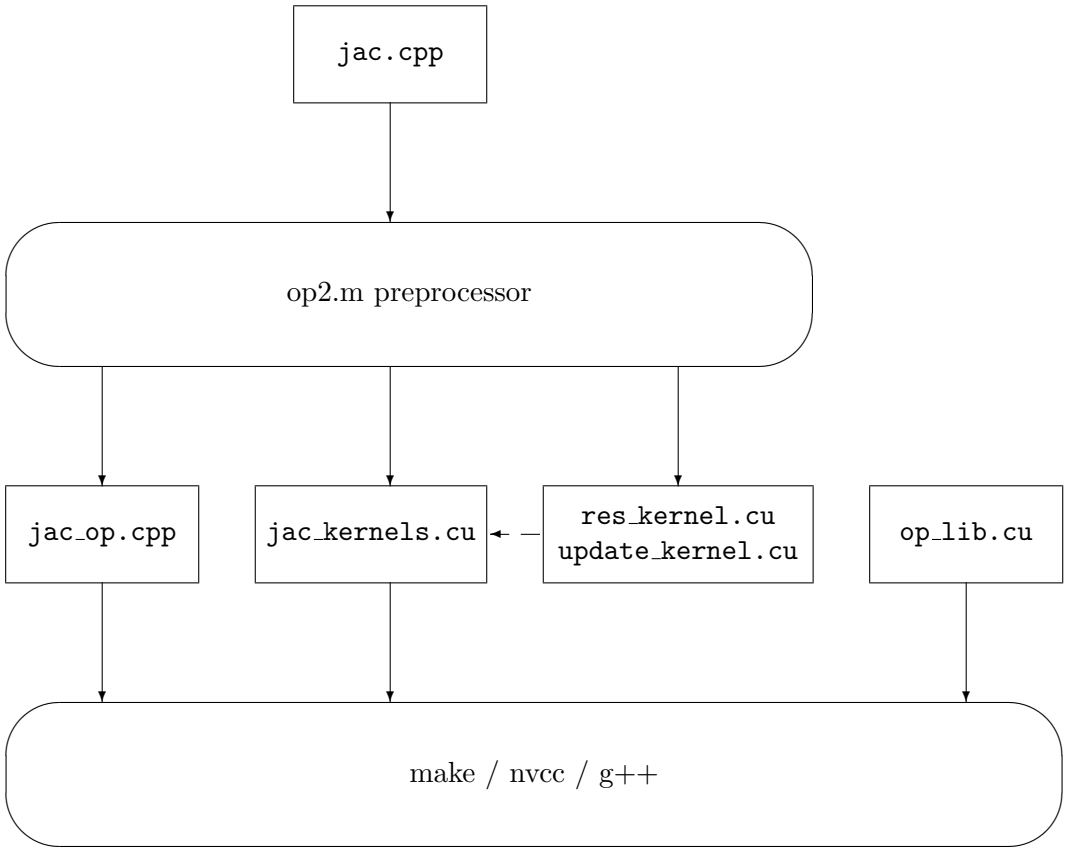

Figure 2: CUDA code build process

3 Initialisation and termination routines

op_init(int argc, char **argv)

This routine must be called before all other OP routines.

op_decl_set(int size, op_set *set, char *name)

This routine declares information about a set.

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

op_decl_ptr(op_set from, op_set to, int dim, int *iptr, op_ptr *ptr, char *name)

This routine declares information about a pointer from one set to another.

`from`	set pointed from
`to`	set pointed to
`dim`	number of pointers per element
`iptr`	input pointer table
`ptr`	output OP pointer ID
`name`	a name used for output diagnostics

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 global variable.

`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_decl_dat(op_set set, int dim, char *type, T *dat, op_dat *data, char *name)

This routine declares information about data associated with a set.

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

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

op_diagnostic_output()

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

op_exit()

This routine must be called last to cleanly terminate the OP computation.

4 Parallel execution routine

As an example, the parallel loop syntax when the user's kernel function has 3 arguments, with the third being a local constant or global reduction array, is:

op_par_loop_3(void (*kernel)(T0 *, T1 *, T2 *), char *name, op_set set,
op_dat arg0, int idx0, op_ptr ptr0, int dim0, char *typ0, op_access acc0,
op_dat arg1, int idx1, op_ptr ptr1, int dim1, char *typ1, op_access acc1,
T2 *arg2, int idx2, op_ptr ptr2, int dim2, char *typ2, op_access acc2)

`kernel`	user's kernel function with 3 arguments of arbitrary type (this is only used for the single-threaded CPU build)
`name`	name of kernel function, used for output diagnostics
`set`	OP set ID, giving set over which the parallel computation is performed
`arg`	OP dataset ID, or pointer to constant or global reduction array
`idx`	index of pointer to be used (-1 ≡ no pointer indirection)
`ptr`	OP pointer ID (`OP_ID` for identity mapping, i.e. no pointer indirection, `OP_GBL` for constant or global reduction array)
`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 `OP_MAX`: global reduction to compute a maximum `OP_MIN`: global reduction to compute a minimum

In this example, `kernel` is a function with 3 arguments of arbitrary type which performs a calculation for a single set element. This will get converted by a preprocessor into a routine called by the CUDA kernel function. The preprocessor will also take the specification of the arguments and turn this into the CUDA kernel function which loads in indirect data (i.e. data addressed indirectly through pointers) from the device main memory into the

shared storage, then calls the converted `kernel` function for each element for each line in the above specification. Indirect data is incremented in shared memory (with thread coloring to avoid possible data conflicts) before being updated at the end of the CUDA kernel call.

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 pointer indirection reference the same elements of the dataset.

Different numbers of arguments are handled similarly by routines with names of the form `op_par_loop_n` where `n` is the number of arguments. Each argument can be either a dataset or a local constant or global reduction array, following the syntax shown above.

5 User-defined datatypes

If the user defines a new datatype `mytype` then this 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_uint 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.

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 Future changes

There will be a new function **op_partition** which will re-number all of the elements in each set, to maximise the data reuse within each mini-partition. This is likely to use the same partitioning algorithm which will be employed for the higher-level distributed-memory partitioning for the MPI implementation in phase 2.

8 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, pointer 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 (including Rolls-Royce?) 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_ptr**: `iptr` provides the part of the pointer 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 → node pointer 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`), will re-number the pointer tables as needed (as well as constructing import/export lists for halo data exchange) and will move all data/pointers/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_ptr_hdf5**: similar to **op_decl_ptr** but with `iptr` replaced by `file` from which the pointer 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`