Fortran OP2 User's Manual

Carlo Bertolli, Adam Betts
Imperial College London, UK

May 19, 2011

Contents

1 Introduction

OP2 is a library targeting parallel Computation Fluid Dynamic (CFD) applications. It currently supports the Finite Volume Method and research work is in progress to extend it to support the Finite Element Method. OP2 supports both structured and unstructured meshes, but it is especially optimised to the unstructured case. The OP2 compiler targets CUDA, OpenCL and AVX/SSE as backends, i.e. it transforms an OP2 program in a semantically corresponding program implemented in the user-selected backend. This documents describes the abstractions provided by the OP2 library and it includes its Fortran API.

2 Overview

In this section we give an overview of the abstractions provided by OP2.

2.1 Sets

An OP2 set is used to model the basic entities forming a mesh. Examples of sets are vertices, edges, etc. In OP2, the only application-related information associated to a set is its size, i.e. the number of elements it includes. In a typical situation, this is an information that is contained in user-defined input files. The actual topology of a mesh is defined in terms of relations between OP2 sets, i.e. maps (see next Subsection).

2.2 Maps

An OP2 map relates two sets A and B, where each element $a \in A$ (A is denoted *source* set) corresponds to one or more elements in the set B (denoted *source* set), i.e. $a \rightarrow \{b_1, \dots, b_N\} \in B$, where N is denoted as the *dimension* of the map and it must be the same constant value for all elements in A. For instance, in a typical situation an element in the edges set corresponds to two elements in the vertices set. This also means that it is not possible to define maps with variable dimension (i.e. a different dimension depending on the source set element), an example of which is the mapping between vertices and edges in a mesh.

Also the mapping information is typically stored in a user-defined input file, which data is used to populate arrays that are passed to the routine initialising a map.

2.3 Datasets

An OP2 dataset is associated to a specific OP2 set, where we relate a set of K scalar values $d_1, \dots, d_K$ to each element of the set. The value K is denoted as the *dimension* of the dataset and it must be the same constant value for all set elements.

Dataset values are typically obtained from an user-defined input file, and the number of scalar values must be equal to the corresponding set size multiplied by the dataset dimension (i.e. to correctly assign the same amount of data to each set element).

2.4 Global Variables

An OP2 global variable is not associated to a specific *op_set*, but it is shared by all parallel loop units[1]. Each parallel loop unit can access all of its fields, without the need of a map.

2.5 Parallel Computation over Sets

The subroutine used to express a parallel computation in OP2 is called op_par_loop . An op_par_loop is defined over a specific *op_set*, called the *target* set. The effect of an op_par_loop is to read and update *op_dat* variables, according to a user-specified subroutine. In general terms, for each parallel computation the programmer has to specify:

- The target *op_set*, over which the parallelism is defined. Semantically, the op_par_loop instantiate a sequential computation for each element in the *op_set* over which it is defined.
- A "user kernel" subroutine, to be evaluated once for each element in the *op_set* over which the op_par_loop is defined. We will always refer to these user-defined, application-level routines as "user kernels", to distinguish them from the notion of a CUDA kernel, which is one of the target programming models of OP2. The general form of a user kernel subroutine is described in Section 7.
- A list of *op_dat* variables that are read and modified by the user kernel subroutine. Each of these variables is also a parameter to the user kernel. It is task of the OP2 run-time to provide each user kernel invocation with the proper section of input op_dat arguments. Notice that, given a *op_set* S over which an op_par_loop is defined, eligible *op_dat* variables for the parallel loop are all those that are defined for S, but also all those that are defined for another *op_set* T if the user has declared an *op_map* between S and T. The former case, in which the passed *op_dat* is defined over S is said to be a *direct* case, as the *op_dat* can be accessed directly given an index of an element in S. The latter case, in which the passed *op_dat* is defined over T, it is said to be an *indirect* case, as the *op_dat* elements can be accessed by: first transforming an element index of S into T (i.e. selecting one of the associated index if the map dimension is greater than 1); then using the transformed index in T to access the *op_dat* .

  To express the above behaviour, for each *op_dat* passed to a op_par_loop the programmer needs to specify:

  - the *op_dat* variable;
  - an index that: (i) in case of a directly accessed *op_dat* or a global data, it is not used and it must be set to OP_NONE; (ii) in case of indirectly accessed op_dat, via a given map M (e.g. from set S to T) an integer index in the range $[1, \text{dim}]$, where *dim* is the dimension of M. In this case, the index identifies which of the mapped elements in T is to be considered;
  - an op_map variable: in case of directly accessed op_dat, the programmer must specify the keyword *OP_ID*. In case of a global data the programmer must specify the keyword *OP_GBL*. Finally, in case of an indirectly accessed op_dat, the user must specify the required op_map variable needed to access the op_dat.
  - an accessing operation code, which must be one of the following named constants: *OP_READ*, *OP_WRITE*, *OP_RW*, and *OP_INC*. These accessing modes must be followed in the implementation of the user kernel. The *OP_RW* operation denotes that the corresponding op_dat variable will be both read and updated in the user kernel. The *OP_INC* operation denotes that the related op_dat elements will be incremented. In case of *OP_INC* for a global variable, the effect is a reduction of the values computed by each kernel invocation.

Semantically, the OP2 run-time will instantiate a number of *parallel loop units* equal to the size of the iteration set. Each parallel loop unit, historically denoted with the term Virtual Processor (or VP, in short) in the parallel computing terminology, has the task of applying the user-kernel subroutine for a specific set element, by properly accessing the input datasets with their corresponding set element index.

[1]See below for the definition of parallel loop unit.

3 OP2 Data Types

op_set

Description : this data type is used to express sets defining a mesh. OP2 require users to specify, for each set, its size, corresponding to the number of elements it is formed of, and a string of characters indicating its name. Figure 1 shows the declaration of multiple op_set variables. As it can be noticed from the example, op_set is implemented as a Fortran derived type (see *type* Fortran keyword used in the example).

```
type ( op_set ) :: vertices , edges , bedges , cells
```

Figure 1: Example of declaration of op_set variables.

op_map

Description : this data type is used to express a relation between two OP2 sets, defining the actual topology of an unstructured mesh, i.e. how its elements are related/connected to each other. If A and B are two op_set variables, a map from A to B associates to each element $a \in A$ a subset of elements $b_1, \ldots, b_n \in B$. The number n must be equal for all elements $a \in A$, i.e. it is not possible that two elements in A are associated to a different number of elements in B. The number n is said to be the *dimension* of the op_map, and it must be declared when initialising an op_map as a constant value (see *_map* call). Figure 2 shows the declaration of multiple op_map variables, again as Fortran structures.

```
type ( op_map ) :: verts2Edges , edges2Verts , cells2Verts , cells2Bedges
```

Figure 2: Example of declaration of op_map variables.

op_dat

Description : this data type is used to express application-related data associated to a OP2 set, i.e. it is used to associate to each element of an op_set a fixed number of data items. The number of data items associated to each op_set element is denoted the op_dat *dimension* and it must be a constant value (see *op_decl_dat* call). An op_dat variable can encapsulate any primitive and structured Fortran type (i.e. defined with the *type* keyword). Figure 3 shows the declaration of multiple op_dat variables, as Fortran derived types.

```
type ( op_dat ) :: p_q , p_qold , bound
```

Figure 3: Example of declaration of op_dat variables.

4 Initialisation and Termination Routines

op_init (debug_level)

Description : this routine initialises the OP2 environment and it must be called before any other OP2 subroutine.

Dummy arguments :

- integer(4) :: **debug_level**: required debug level. It can assume the following values:
  - 0: none
  - 1: error checking

  - 2: information on plan construction
  - 3: report execution of parallel loops
  - 4: report use of old plans
  - 5: report positive checks in op_plan_check function

op_exit()

Description : this routine terminates the OP2 environment and must be called after all OP2 subroutines.

5 Data Structure Declaration Routines

op_decl_set (size, setVariable, setName)

Description : this routine is used to initialise a op_set variable. Its parameters are:

- integer(4) :: *size*: the number of elements in the op_set.
- type(op_set) :: *setVariable*: the op_set variable to be initialised.
- character(len=*) :: *setName*: a Fortran string specifying the name of the set (for debug purposes). An example of how a set name can be declared is shown in Figure 4.

```
    character(len=5) :: nodesName ='nodes'
```

Figure 4: Example of declaration of an op_set name, to be passed to the op_decl_set routine.

op_decl_map (sourceSet, targetSet, dimension, inputData, mapVariable, mapName)

Description : this routine is used to initialise an op_map variable. According to the description above, an op_map relates the elements in a op_set (called *source* set) to the elements of another op_set (called *target* set). The parameters of this call are:

- type(op_set) :: *sourceSet*: the variable representing the source set of the declared map.
- type(op_set) :: *targetSet*: the variable representing the target set of the declared map.
- integer(4) :: *dimension*: the dimension of the map (i.e. the number of elements in toSet associated to each element in fromSet).
- integer(4), dimension(*) :: *inputData*: an array of integers, including the data needed to instantiate the mapping between fromSet and toSet. The length of this array must be equal to the size of the fromSet multiplied by the dimension of the map (see previous parameter).
- type(op_map) :: *mapVariable*: the op_map variable to be initialised.
- character(len=*) :: *mapName*: a Fortran string specifying the name of the map (for debug purposes).

op_decl_dat (associateSet, dimension, inputData, datVariable, datName)

Description : this routine is used to initialise an op_dat variable. Its parameters are:

- type(op_set) :: *associatedSet*: the op_set variable to which the declared op_dat variable is associated.
- integer(4) :: *dimension*: the dimension of the declared op_dat, i.e. the number of data items associated to each element in op_set.
- "type", dimension(:) :: *inputData*: an array of arbitrary type (here denoted with the "type" string, which can be integer(4), real(4), real(8), etc..) including the actual data to be associated to the op_set. The length of this array must be equal to the size of the associated set multiplied by the dimension of the declared op_dat (see previous parameter). Therefore, the **:** character must be replaced by the actual array size.
- character(len=*) :: *mapName*: a Fortran string specifying the name of the dataset (for debug purposes).

op_decl_gbl (inputData, dimension, datVariable, gblName)

Description : this routine is used to initialise a global OP2 data variable. This kind of data is represented, in terms of data types, as an op_dat, but semantically with a special accessing policy (see Overview Section). The parameters of this call are:

- "type", dimension(:) :: *inputData*: an array of arbitrary type including the actual data to be associated to the declared global variable. Currently, also scalar data must be expressed in terms of a single-sized array.
- integer(4) :: *dimension*: the size of the input data array, which is denoted as the dimension of the global data.
- type(op_dat) :: *datVariable* the op_dat variable to which the declared global variable is associated.
- character(len=*) :: *mapName*: a Fortran string specifying the name of the global data (for debug purposes).

6 Parallel Computation Routines

The prototype for an op_par_loop call with two op_dat input variables is shown below:

```
op_par_loop  (  userSubroutine ,  iterationSet ,    &
             &  datVar1 ,  index1 ,  map1,  access1 ,  &
             &  datVar2 ,  index2 ,  map2,  access2   &
           &  )
```

Description : as described in the Overview section, this routine is used to express a parallel computation over a specific op_set, reading and modifying passed op_dat variables. The op_par_loop call can take an arbitrary number of op_dat variables (i.e. an instance of the above specified parameter list), which must correspond to the number of parameters to the user kernel passed as first argument (see below). The first two parameters of this call are:

- subroutine *userSubroutine*: this is the user kernel subroutine, that is instantiated in a number of parallel units equal to the size of the passed op_set (see next parameter).
- type(op_set) :: *iterationSet*: this is the set over which the op_par_loop is defined, and its size defines the number of parallel units that will be instantiated by the OP2 run-time when executing this call.

These parameters are followed by an arbitrary number of "lines", each related to an input dataset. For each line, the programmer must specify:

- type(op_dat) :: *datVar1*: the related dataset variable.
- integer(4) :: *index1*: in case of direct mapping or global data, this parameter is not used, and it must be set to the named constant OP_NONE. Consider the case of indirectly accessed op_dat, with a map from the iteration set S to another set T with dimension N. *index1* is a value in the range $[1, N]$, identifying one of the N elements in T to which each S element is mapped to.
- type(op_map) :: *map1*: in case of directly accessed op_dat this value must be set to the OP2 reserved name **OP_ID**. In case of global data, this value must be set to the OP2 reserved name **OP_GBL**. In case of an indirectly accessed op_dat, this must be the reference to an op_map variable mapping the iteration set to the set over which the op_dat is defined.
- op_access :: *access1*: this parameter can assume the values OP_READ, OP_WRITE, OP_RW or OP_INC (see Overview Section).

```
subroutine userKernel ( in1 , in2 , ... , inN )

  ! currently supported data types are integer(4), real(4) and real(8)
  "type1", dimension(size1) :: in1
  "type2", dimension(size2) :: in2
    ...
  "typeN", dimension(sizeN) :: inN

  ! use of input variables

end subroutine userKernel
```

Figure 5: General form of a user kernel.

7 User Kernel Interface

In this section we show how to express a user kernel to be used in a op_par_loop. The general form of a user kernel is shown in Figure 5.

Each input data corresponds to a op_dat argument line in the caller op_par_loop. As above, "type" can be an arbitrary data type. The size of the passed input data corresponds to the *dimension* parameter passed to the *op_decl_dat* call when declaring the corresponding op_dat variable.

A full example including the declaration of op_dat variables, the invocation of an op_par_loop and the corresponding user kernel declaration is shown in Figures 6 and 7.

```
program main

  real(8), dimension(:), allocatable :: x, q, adt, res

  ! allocation and filling of x, q, adt and res, declaration of names, etc..

  ! name example
  character(len=5) :: itSetName = 'nodes'

  ! declaration of iteration set
  type(op_set) :: iterationSet

  ! declaration of another set
  type(op_set) :: anotherSet

  ! mapping between iterationSet and anotherSet
  type(op_map) :: iter2another

  ! declaration of op_dat variables
  type(op_dat) :: p_x, p_q, p_adt, p_res

  ! initialisation of sets
  call op_decl_set ( itSetSize, iterationSet, itSetName )
  call op_decl_set ( anothSetSize, anotherSet, anothSetName )

  ! initialisation of map
  call op_decl_map ( iterationSet, anotherSet, 2, mappingData, iter2another,
     mapName)

  ! initialisation of op_dat variables
  call op_decl_dat ( iterationSet, xDimension, x, p_x, xName )
  call op_decl_dat ( anotherSet, qDimension, q, p_q, qName )
  call op_decl_dat ( anotherSet, adtDimension, adt, p_adt, adtName )
  call op_decl_dat ( anotherSet, resDimension, res, p_res, resName )

  ! parallel computation
  op_par_loop ( res_calc, iterationSet, &
              & p_x,   -1,          OP_ID, OP_READ, &
              & p_q,    1, iter2Another, OP_READ, &
              & p_q,    2, iter2Another, OP_READ, &
              & p_adt,  1, iter2Another, OP_READ, &
              & p_adt,  2, iter2Another, OP_READ, &
              & p_res,  1, iter2Another,  OP_INC, &
              & p_res,  2, iter2Another,  OP_INC  &
            & )

end program main
```

Figure 6: Full example to show how a user kernel is defined. In this figure we show the code of the main program calling the op_par_loop.

```
subroutine res_calc ( x1, q1, q2, adt1, adt2, res1, res2 )

  ! formal parameters
  real(8), dimension(2) :: x1
  real(8), dimension(4) :: q1
  real(8), dimension(4) :: q2
  real(8), dimension(1) :: adt1
  real(8), dimension(1) :: adt2
  real(8), dimension(4) :: res1
  real(8), dimension(4) :: res2

  res1(1) = res1(1) + 0.5 * ( x1(1) + q1(1) + adt1(1) )
  res1(2) = res1(2) + 0.5 * ( x1(2) + q1(2) + adt1(1) )
  res1(3) = res1(3) + 0.5 * ( x1(1) + q1(3) + adt1(1) )
  res1(4) = res1(4) + 0.5 * ( x1(2) + q1(4) + adt1(1) )

  res2(1) = res2(1) + 0.5 * ( x1(1) + q2(1) + adt2(1) )
  res2(2) = res2(2) + 0.5 * ( x1(2) + q2(2) + adt2(1) )
  res2(3) = res2(3) + 0.5 * ( x1(1) + q2(3) + adt2(1) )
  res2(4) = res2(4) + 0.5 * ( x1(2) + q2(4) + adt2(1) )

end subroutine res_calc
```

Figure 7: Full example to show how a user kernel is defined. In this figure we show the code of the user kernel.