A simple user manual for OpenArray
Version 1.0

Xiaomeng Huang, Xing Huang, Dong Wang, Qi Wu,

Shixun Zhang, and Yi Li

August 2019

Table of Contents

1. Overview

OpenArray is a simple operator library for the decoupling of ocean modelling and parallel computing. The library is promoted as a development tool for the future numerical models by making complex parallel programming transparent.

OpenArray features are easy-to-use, high efficiency and portability. Users can write simple operator expressions in Fortran to solve partial differential equations (PDEs) in parallel. The performance of the programs implemented with OpenArray is similar to that of original parallel program manually optimized by experienced programmers. In addition to CPU platforms, the current version of OpenArray also supports the Sunway TaihuLight system. The GPU version is under development.

This quick guide summarily describes the OpenArray Version 1.0, a comprehensive user manual including more details is currently under preparation. As the new version

of OpenArray is being developed, these documents will be continuously enhanced and updated. For the latest version of the documents, please visit *https://github.com/hxmhuang/OpenArray/tree/master/doc.*

The main reference for OpenArray v1.0 is:
Xiaomeng Huang, Xing Huang, Dong Wang, Qi Wu, et al., 2019, OpenArray v1.0: A Simple Operator Library for the Decoupling of Ocean Modelling and Parallel Computing. https://www.geosci-model-dev-discuss.net/gmd-2019-28/.

Please send feedback to *hxm@mail.tsinghua.edu.cn*.

2. Installation

The following is a detailed installation guide on Mac OS and Linux operating system, respectively. Before compiling OpenArray, the following software and libraries are required:

1) gcc/g++/gfortran compilers, version 4.9.0 or later.
2) Message Passing Interface (MPI) compilers (MPICH v3.2.1 or later; Openmpi v3.0.0 or later; Intel MPI compiler 2017 or later).
3) Parallel NetCDF (PnetCDF), version 1.7.0 or higher.

Important notices:
1) The version of GNU Compiler Collection must be 4.9.0 or later, which supports the C++ 11 standard OpenArray requires.
2) Installing MPI library, PnetCDF, and OpenArray with different version of compilers may cause errors during the link step.

If you have any question while installing OpenArray, please do not hesitate to submit issues on GitHub or email to *hxm@mail.tsinghua.edu.cn*, we will deal with the problems as soon as possible.

2.1 Installation on Linux

The following instructions take you through a sequence of steps to get the default configuration of OpenArray up and running on Linux. Please use the same compilers to install PnetCDF and OpenArray.

(a) Check gcc/g++/gfortran compilers and MPI compilers have been installed already and specify the MPI compiler.

gcc -v

g++ -v

gfortran –v

Make sure the gcc version is 4.9.0 or later.

For MPICH and Openmpi:

which mpicc

which mpicxx

which mpif90

export MPICC=mpicc

export MPICXX=mpicxx

export MPIF90=mpif90

export MPIF77=mpif77

For Intel compiler and Intel MPI library:

which mpiicc

which mpiicpc

which mpiifort

export MPICC=mpiicc

export MPICXX=mpicpc

export MPIF90=mpiifort

export MPIF77=mpiifort

(b) Install PnetCDF. The default installation directory of PnetCDF is *${HOME}/install*:

wget http://cucis.ece.northwestern.edu/projects/PnetCDF/Release/pnetcdf-1.11.2.tar.gz

tar xf pnetcdf-1.11.2.tar.gz

cd pnetcdf-1.11.2

./configure --prefix=${HOME}/install

make (or try parallel make: make –j8)

make install

(c) Install OpenArray. The default installation directory of OpenArray is *${HOME}/install*.

wget https://github.com/hxmhuang/OpenArray/archive/master.zip

unzip master.zip

cd OpenArray-master

export PNETCDF_DIR=${HOME}/install

./configure --prefix=${HOME}/install

make (or try parallel make: make –j8)

make install

(d) Test. This executable file *manual_main* is a demo written based on OpenArray. If you have completed all of the above steps, you have successfully installed OpenArray.

./manual_main

2.2 Installation on Mac OS

The following instructions take you through a sequence of steps to get the default configuration of OpenArray up and running on MacOS. Please use the same compilers to install PnetCDF and OpenArray.

(a) Check gcc/g++/gfortran compilers and MPI compilers have been installed already and specify the MPI compiler on the target machine. If the compilers are not already installed, the home brew package manager is recommended to install and manage the softwares.
Note: the gcc command on MacOS is symbolically linked to llvm-gcc, please use *brew list gcc* to check the GNU Compiler Collection has been installed.

/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

brew update

brew list gcc

For MPICH and Openmpi:

which mpicc

which mpicxx

which mpif90

export MPICC=mpicc

export MPICXX=mpicxx

export MPIF90=mpif90

export MPIF77=mpif77

For Intel compiler and Intel MPI library:

which mpiicc

which mpiicpc

which mpiifort

export MPICC=mpiicc

export MPICXX=mpicpc

export MPIF90=mpiifort

export MPIF77=mpiifort

(b) Install PnetCDF. The installation directory of PnetCDF is *${HOME}/install*.

wget http://cucis.ece.northwestern.edu/projects/PnetCDF/Release/pnetcdf-1.11.2.tar.gz

tar xf pnetcdf-1.11.2.tar.gz

cd pnetcdf-1.11.2

./configure --prefix=${HOME}/install

make (or try parallel make: make –j8)

make install

(c) Install OpenArray. The default installation directory of OpenArray is

${HOME}/install.

```
wget https://github.com/hxmhuang/OpenArray/archive/master.zip
unzip master.zip
cd OpenArray-master
export PNETCDF_DIR=${HOME}/install
./configure --prefix=${HOME}/install
make (or try parallel make: make –j8 )
make install
```

(d) Test. This executable file *manual_main* is a demo written based on OpenArray. If you have completed all of the above steps, you have successfully installed OpenArray.

```
./manual_main
```

3. Operators and Functions

For simplicity, functions in OpenArray are divided into 6 classes for introduction (Table 1). The functions are written in a style similar with Matlab, to smooth the learning curve.

Table 1: Major functions provided in OpenArray

Classes	Function	Description
Array Creation	zeros(m,n,k)	Create array of all zeros
	ones(m,n,k)	Create array of all ones
	rands(m,n,k)	Create an array with random number
	seqs(m,n,k)	Create a sequence integer array staring from 0
Arithmetic Operation	+,-,*,/	Basic arithmetic operators
	>,<,>=,<=,/=,==	Comparison operators
	.or., .and.	Logical operators
	sin, cos, tan, asin, acos, atan, abs, rcp, exp, log, log10, tanh, sinh, cosh, **	Basic math functions
Array Operation	sub	Get a sub slice of Array
	shift	Shift Array in a given direction
	sum	Sum Array in a given direction
	csum	Cumulative sum
	max, max_at	Get maximum value or position
	min, min_at	Get minimum value or position
	rep	Repeat array in a given direction
	set	Array assignment
Stencil Operation	AXB, AXF, AYB, AYF, AZB, AZF	Averaging stencil operators
	DXB, DXF, DYB, DYF, DZB, DZF	Differential stencil operators
I/O Operation	save(A,…)	Save array into file
	A=load(…)	Load array from file
Other functions	display(A)	Display array
	tic, toc, show_timer	Calculate and print the execution

		time
	grid_init('C', dx, dy, dz)	Initialize the Arakawa C-grid with a set of grid increments, dx, dy, dz
	grid_bind(A, pos)	Bind *Array* A to the *pos* point

The following examples are provided to illustrate the usage of the functions. You can get the complete source code of all the examples in oa_main.F90.

Outline of the main program for the following examples:

```
program main
   use mpi
   use oa_test      ! import example module
   implicit none
   integer :: step
   integer :: i, nt, nx, ny, nz
   ! initialize OpenArray, no split in z-direction
   call oa_init(MPI_COMM_WORLD, [-1, -1, 1])

   call array_creation()
   call arithmetic_operation()
   ...
   call show_timer()
   call oa_finalize()              ! finalize the OpenArray
end program main
```

How to run:

```
// compile and link
make –f makefile.intel oalib_obj
make –f makefile.intel manual_main
./manual_main
```

3.1 Array Creation

This class creates *Array* type variables. *Array* is a new derived data type that comprises a series of information, including a 3-dimensional array to store data, a pointer to the computational grid, a Message Passing Interface (MPI) communicator, the size of the halo region and other information about the data distribution.

Syntax:

X = zeros(m, n, k) returns a m-by-n-by-k *Array* of zeros, where m, n, k indicate the size of each dimension.

X = ones(m, n, k) returns a m-by-n-by-k *Array* of ones, where m, n, k indicate the size of each dimension.

X = seqs(m, n, k) returns a m-by-n-by-k *Array* of sequence numbers starting from [0, 0, 0], where m, n, k indicate the size of each dimension.

X = rands(m, n, k) returns a m-by-n-by-k *Array* of random numbers, where m, n, k indicate the size of each dimension.

Example

```
   subroutine array_creation()
      implicit none
      type(array) :: A, B, C, D

      A = zeros(2, 2, 2)
```

```
    B = ones(2, 2, 2)
    C = seqs(2, 2, 2)
    D = rands(2, 2, 2)

    call display(A, "zeros = ")
    call display(B, "ones = ")
    call display(C, "seqs = ")
    call display(D, "rands = ")
  end subroutine
```

// The output displays array's information and data, we only display zeros information, in other examples we will omit array information.

```
zeros =
      data type = float
      pos = -1
      is_pseudo = 0
      bitset = 111
      global_shape = [2, 2, 2]
      procs_shape = [1, 1, 1]
      bound_type = [0, 0, 0]
      stencil_type = 1
      stencil_width = 1
      lx = [2]
      ly = [2]
      lz = [2]
      clx = [0, 2]
      cly = [0, 2]
      clz = [0, 2]
[k = 0]
          j = 0                  j = 1
i = 0     0.000000000000000      0.000000000000000
i = 1     0.000000000000000      0.000000000000000
[k = 1]
          j = 0                  j = 1
i = 0     0.000000000000000      0.000000000000000
i = 1     0.000000000000000      0.000000000000000

ones =
[k = 0]
          j = 0                  j = 1
i = 0     1.000000000000000      1.000000000000000
i = 1     1.000000000000000      1.000000000000000
[k = 1]
          j = 0                  j = 1
i = 0     1.000000000000000      1.000000000000000
i = 1     1.000000000000000      1.000000000000000

seqs =
[k = 0]
          j = 0                  j = 1
i = 0     0.000000000000000      2.000000000000000
i = 1     1.000000000000000      3.000000000000000
[k = 1]
          j = 0                  j = 1
i = 0     4.000000000000000      6.000000000000000
i = 1     5.000000000000000      7.000000000000000

rands =
[k = 0]
          j = 0                  j = 1
i = 0     0.108893647789955      0.701414048671722
i = 1     0.520004570484161      0.738651990890503
[k = 1]
          j = 0                  j = 1
i = 0     0.133548513054848      0.721330642700195
i = 1     0.041211254894733      0.591001868247986
```

3.2 Arithmetic Operation

This class provides arithmetic operators, including basic arithmetic operators,

comparison operators, logical operators, and mathematical functions.

Syntax:

- **Basic arithmetic operators**

C = A + B adds *Array* A and B and returns the result in C.

- **Comparison operators**

A == B returns a logical *Array* with elements set to logical 1 (true) where *Array* A and B are equal; otherwise, the element is logical 0 (false).

- **Logical operators**

A .and. B performs a logical conjunction of *Array* A and B, and returns an *Array* with elements set to logical 1 or 0.

- **Mathematical functions**

Y = sin(X) returns the sine of the elements of *Array* X.
Y = exp(X) returns the exponential for each element in *Array* X.

Example

```
subroutine arithmetic_operation()
    implicit none
    type(array) :: A, B, C, D, E, ans

    ! basic arithmetic operators
    A = zeros(2, 2, 1)
    B = ones(2, 2, 1)
    C = seqs(2, 2, 1)
    D = rands(2, 2, 1)

    E = A + B * C
    call display(E, "E = ")
    E = E / 2.0
    call display(E, "E = ")

    ! comparison operators
    ans = B < C
    call display(ans, "ones < seqs = ")
    ans = A == C
    call display(ans, "ones == seqs = ")

    ! logical operators
    ans = A .or. C
    call display(ans, "zeros .or. seqs = ")
    ans = A .and. C
    call display(ans, "zeros .and. seqs = ")

    ! basic math functions
    E = consts_double(2, 2, 1, 2.D0*atan(1.D0))
    ans = sin(E)
    call display(ans, "sin(PI/2) = ")
    ans = exp(C)
    call display(ans, "exp(seqs) = ")
    ans = log(ans)
    call display(ans, "log(exp(seqs)) = ")
end subroutine
```

3.3 Array Operation

This class provides basic operations for *Array*.

- **sub**

Get a sub slice of *Array*.

Syntax:

B = sub(A, [stx, edx], [sty, edy], [stz, edz]) gets a slice of *Array* A and the result is a (edx-stx+1)-by-(edy-sty+1)-by-(edz-stz+1) *Array*

B = sub(A, [stx, edx], [sty, edy], [stz, edz]) gets a slice of array A, starting from [stx, sty, stz] and ending at [edx, edy, edz]. The result is a (edx-stx+1)-by-(edy-sty+1)-by-(edz-stz+1) array.

B = sub(A, 1, 2) gets a slice of *Array* A and the result is a 1-by-1-by-k array, where k is the size of z-dimension of *Array* A

B = sub(A, ':', ':', [2, 2]) gets a slice of array A and the result is a m-by-n-by-1 array

Example

```
! get a sub slice of array
A = seqs(5, 5, 2)
call display(A, "A = ")
! sub function index start from 1
ans = sub(A, [2, 4], [2, 4], [1, 1])
call display(ans, "sub(A, [2, 4], [2, 4], [1, 1]) = ")
ans = sub(A, ':', ':', [2, 2])
call display(ans, "sub(A, ':', ':', [2, 2]) = ")
ans = sub(A, 1, 2)
call display(ans, "sub(A, 1, 2) = ")
```

Output

```
A =
[k = 0]
          j = 0                 j = 1                 j = 2                 j = 3
i = 0     0.000000000000000     4.000000000000000     8.000000000000000     12.000000000000000
i = 1     1.000000000000000     5.000000000000000     9.000000000000000     13.000000000000000
i = 2     2.000000000000000     6.000000000000000     10.000000000000000    14.000000000000000
i = 3     3.000000000000000     7.000000000000000     11.000000000000000    15.000000000000000
[k = 1]
          j = 0                 j = 1                 j = 2                 j = 3
i = 0     16.000000000000000    20.000000000000000    24.000000000000000    28.000000000000000
i = 1     17.000000000000000    21.000000000000000    25.000000000000000    29.000000000000000
i = 2     18.000000000000000    22.000000000000000    26.000000000000000    30.000000000000000
i = 3     19.000000000000000    23.000000000000000    27.000000000000000    31.000000000000000

sub(A, [2, 4], [2, 4], [1, 1]) =
[k = 0]
          j = 0                 j = 1                 j = 2
i = 0     5.000000000000000     9.000000000000000     13.000000000000000
i = 1     6.000000000000000     10.000000000000000    14.000000000000000
i = 2     7.000000000000000     11.000000000000000    15.000000000000000

sub(A, ':', ':', [2, 2]) =
[k = 0]
          j = 0                 j = 1                 j = 2                 j = 3
i = 0     16.000000000000000    20.000000000000000    24.000000000000000    28.000000000000000
i = 1     17.000000000000000    21.000000000000000    25.000000000000000    29.000000000000000
i = 2     18.000000000000000    22.000000000000000    26.000000000000000    30.000000000000000
i = 3     19.000000000000000    23.000000000000000    27.000000000000000    31.000000000000000

sub(A, 1, 2) =
[k = 0]
          j = 0
i = 0     4.000000000000000
[k = 1]
          j = 0
i = 0     20.000000000000000
```

- **shift, circshift**

Shift an *Array* in a given direction.

Syntax:

B= shift(A, i, j, k) shifts the values in *Array* A by i, j, k positions along corresponding dimension. The added elements are set to zeros.

B= circshift(A, i, j, k) circularly shifts the elements in *Array* A by i, j, k positions along corresponding dimensions.

Example

```
! shift array in a given direction
A = seqs(2, 2, 2)
call display(A, "A = ")
ans = shift(A, 1, 0, 0)
call display(ans, "shift(A, 1, 0, 0) = ")
ans = shift(A, 0, 1)
call display(ans, "shift(A, 0, 1) = ")
ans = shift(A, 0, -1, 0)
call display(ans, "shift(A, 0, -1, 0) = ")
ans = circshift(A, 0, 1)
call display(ans, "circshift(A, 0, 1) = ")
```

Output

```
A =
[k = 0]
          j = 0                 j = 1
i = 0     0.000000000000000     2.000000000000000
i = 1     1.000000000000000     3.000000000000000
[k = 1]
          j = 0                 j = 1
i = 0     4.000000000000000     6.000000000000000
i = 1     5.000000000000000     7.000000000000000

shift(A, 1, 0, 0) =
[k = 0]
          j = 0                 j = 1
i = 0     0.000000000000000     0.000000000000000
i = 1     0.000000000000000     2.000000000000000
[k = 1]
          j = 0                 j = 1
i = 0     0.000000000000000     0.000000000000000
i = 1     4.000000000000000     6.000000000000000

shift(A, 0, 1) =
[k = 0]
          j = 0                 j = 1
i = 0     0.000000000000000     0.000000000000000
i = 1     0.000000000000000     1.000000000000000
[k = 1]
          j = 0                 j = 1
i = 0     0.000000000000000     4.000000000000000
i = 1     0.000000000000000     5.000000000000000

shift(A, 0, -1, 0) =
[k = 0]
          j = 0                 j = 1
i = 0     2.000000000000000     0.000000000000000
i = 1     3.000000000000000     0.000000000000000
[k = 1]
          j = 0                 j = 1
i = 0     6.000000000000000     0.000000000000000
i = 1     7.000000000000000     0.000000000000000

circshift(A, 0, 1) =
[k = 0]
          j = 0                 j = 1
i = 0     2.000000000000000     0.000000000000000
i = 1     3.000000000000000     1.000000000000000
[k = 1]
          j = 0                 j = 1
i = 0     6.000000000000000     4.000000000000000
```

```
i = 1     7.000000000000000     5.000000000000000
```

- **sum, csum**

Syntax:

S = sum(A, dim) returns the sum along dim-dimension of *Array* A

S = csum(A, dim) returns the cumulative sum along dim-dimension of *Array* A

Example

```
! sum array in a given direction
A = seqs(2, 2, 2)
ans = sum(A, 1)
call display(ans, "sum(A, 1) = ")

! cumulative sum
ans = csum(A, 2)
call display(ans, "csum(A, 2) = ")
```

Output

```
sum(A, 1) =
[k = 0]
          j = 0                 j = 1
i = 0     1.000000000000000     5.000000000000000
[k = 1]
          j = 0                 j = 1
i = 0     9.000000000000000    13.000000000000000

csum(A, 2) =
[k = 0]
          j = 0                 j = 1
i = 0     0.000000000000000     2.000000000000000
i = 1     1.000000000000000     4.000000000000000
[k = 1]
          j = 0                 j = 1
i = 0     4.000000000000000    10.000000000000000
i = 1     5.000000000000000    12.000000000000000
```

- **max, max_at, min, min_at**

Syntax:

M = max(A) returns the maximum element of an *Array A*

M = max_at(A) returns the position of maximum element of an *Array A*

M = max(A, B) returns the maximum value between *Array* A and B

Example

```
! get maximum value or its position
A = rands(2, 2, 1)
B = rands(2, 2, 1)
call display(A, "A = ")
call display(B, "B = ")
ans = max(A, B)
call display(ans, "max(A, B) = ")
mx = max(A)
print *, "max(A) = ", mx
pos = max_at(A)
print *, "max_at(A) = ", pos

! get minimum value or its position
ans = min(A, B)
call display(ans, "min(A, B) = ")
mx = min(A)
print *, "min(A) = ", mx

pos = min_at(A)
print *, "min_at(A) = ", pos
```

Output

```
A =
[k = 0]
          j = 0                    j = 1
i = 0     0.232108786702156        0.777967989444733
i = 1     0.062826067209244        0.049972448498011

B =
[k = 0]
          j = 0                    j = 1
i = 0     0.448162168264389        0.238313674926758
i = 1     0.337830632925034        0.033533636480570

max(A, B) =
[k = 0]
          j = 0                    j = 1
i = 0     0.448162168264389        0.777967989444733
i = 1     0.337830632925034        0.049972448498011

max(A) =      0.7779680
max_at(A) =                  1                  2                  1

min(A, B) =
[k = 0]
          j = 0                    j = 1
i = 0     0.232108786702156        0.238313674926758
i = 1     0.062826067209244        0.033533636480570

 min(A)      =     4.9972448E-02
 min_at(A) =                  2                  2                  1
```

- **rep**

Syntax:

B = rep(A, i, j, k) returns an *Array* containing i*j*k copies of A, the size of B is size(A).*[i,j,k]

Example

```
! repeat array in a given direction
A = seqs(2, 2, 1)
ans = rep(A, 1, 2, 1)
call display(ans, "rep(A, 1, 2, 1) = ")
```

Output

```
rep(A, 1, 2, 1) =
[k = 0]
          j = 0                    j = 1                    j = 2                    j = 3
i = 0     0.000000000000000        2.000000000000000        0.000000000000000        2.000000000000000
i = 1     1.000000000000000        3.000000000000000        1.000000000000000        3.000000000000000
```

- **set**

Syntax:

set(sub(A, [stx,edx],[sty,edy],[stz,edz]), const) sets slice of array A to value const

set(sub(A, [stx,edx],[sty,edy],[stz,edz]), farray) sets slice of array A to fortran array

Example

```
! array assignment
call set(sub(A, [1,2], [1,1], [1,1]), -0.5)
call display(A, "set(A) = ")

if(allocated(farray)) deallocate(farray)
allocate(farray(2, 1, 1))
farray(:,:,:) = 10
call set(sub(A, [1,2], [1,1], [1,1]), farray)
call display(A, "set with fortran array = ")
```

Output

```
set(A) =
[k = 0]
          j = 0                    j = 1
i = 0    -0.500000000000000     2.000000000000000
i = 1    -0.500000000000000     3.000000000000000

set with fortran array =
[k = 0]
          j = 0                    j = 1
i = 0    10.000000000000000     2.000000000000000
i = 1    10.000000000000000     3.000000000000000
```

3.4 Stencil Operation

OpenArray currently provides twelve basic stencil operations abstracted from PDEs. The naming of the operators (AXF, AXB, DXF, DXB, etc.) abbreviates the corresponding operations into three letters with the form [A|D][X|Y|Z][F|B]. The first letter contains two options, A or D, indicating an average or a differential operator. The second letter contains three options, X, Y or Z, representing the direction of operation. The last letter contains two options, F or B, representing forward or backward operation.

Syntax:

AXB(A) represents the Averaging of *Array* A in X direction with Forward step.

DZB(A) represents the Differential of *Array* A in Z direction with Backward step.

Example

```
A = seqs(2, 2, 2)

! averaging stencil operators
ans = AXB(A)
call display(ans, "AXB(A) = ")
ans = AZF(A)
call display(ans, "AZF(A) = ")

! differential stencil operators
ans = DXB(A)
call display(ans, "DXB(A) = ")
ans = DYF(A)
```

Output

```
AXB(A) =
[k = 0]
          j = 0                    j = 1
i = 0     0.000000000000000     0.000000000000000
i = 1     0.500000000000000     2.500000000000000
[k = 1]
          j = 0                    j = 1
i = 0     0.000000000000000     0.000000000000000
i = 1     4.500000000000000     6.500000000000000

AZF(A) =
[k = 0]
          j = 0                    j = 1
i = 0     2.000000000000000     4.000000000000000
i = 1     3.000000000000000     5.000000000000000
[k = 1]
          j = 0                    j = 1
i = 0     0.000000000000000     0.000000000000000
i = 1     0.000000000000000     0.000000000000000

DXB(A) =
[k = 0]
```

```
          j = 0                 j = 1
i = 0     0.000000000000000     0.000000000000000
i = 1     1.000000000000000     1.000000000000000
[k = 1]
          j = 0                 j = 1
i = 0     0.000000000000000     0.000000000000000
i = 1     1.000000000000000     1.000000000000000

DYF(A) =
[k = 0]
          j = 0                 j = 1
i = 0     2.000000000000000     0.000000000000000
i = 1     2.000000000000000     0.000000000000000
[k = 1]
          j = 0                 j = 1
i = 0     2.000000000000000     0.000000000000000
i = 1     2.000000000000000     0.000000000000000
```

3.5 I/O Operation

The I/O functions are encapsulated based on the PnetCDF library.

save, load

Syntax:

save(variables, filename, data-name) saves variables into a NetCDF file.

A = load(filename, data-name) loads variable from a NetCDF file.

Example

```
A = ones(2, 2, 2)

! save array into file
call save(A, "test_io.nc", "a")

! load array from file
ans = load("test_io.nc", "a")
call display(ans, "load from file")
```

Output

```
load from file
[k = 0]
          j = 0                 j = 1
i = 0     1.000000000000000     1.000000000000000
i = 1     1.000000000000000     1.000000000000000
[k = 1]
          j = 0                 j = 1
i = 0     1.000000000000000     1.000000000000000
i = 1     1.000000000000000     1.000000000000000
```

3.6 Other functions

Syntax:

call display(A, “A = “) prints information and data of *array* A;

call tic(“label”) start stopwatch timer with a user-defined ‘label’;

call toc(“label”) read elapsed time from stopwatch started by the tic function with ‘label’;

call grid_init(‘C’, dx, dy, dz) initializes the Arakawa C-grid with a set of grid increments, dx, dy, dz

call grid_bind(A, pos) binds array A to point pos;

Example

```
subroutine util_operation()
    implicit none
    type(array) :: A, ans, dx, dy, dz

    A = zeros(3, 3, 3, dt=OA_DOUBLE)
    dx = sub(A, ':', ':', 1)    + 0.1D0
    dy = sub(A, ':', ':', 1)    + 0.2D0
    dz = sub(A,    1,      1, ':') + 0.15D0

    call display(dx, 'dx = ')
    call display(dy, 'dy = ')
    call display(dz, 'dz = ')

    call tic("grid_init")    ! start the timer
    call grid_init('C', dx, dy, dz) ! init grid C with dx, dy, dz
    call toc("grid_init")    ! end the timer

    A = seqs(3, 3, 3, dt=OA_DOUBLE)
    ans = 1.0 * DXF(A)
    call display(ans, "DXF(A) = ")

    call grid_bind(A, 3)    ! bind A to point 3
    ans = DXF(A)
    call display(ans, "after binding C gird at point 3, DXF(A) = ")
end subroutine
```

Output

```
DXF(A) =
[k = 0]
          j = 0                 j = 1                 j = 2
i = 0    1.000000000000000    1.000000000000000    1.000000000000000
i = 1    1.000000000000000    1.000000000000000    1.000000000000000
i = 2    0.000000000000000    0.000000000000000    0.000000000000000
[k = 1]
          j = 0                 j = 1                 j = 2
i = 0    1.000000000000000    1.000000000000000    1.000000000000000
i = 1    1.000000000000000    1.000000000000000    1.000000000000000
i = 2    0.000000000000000    0.000000000000000    0.000000000000000
[k = 2]
          j = 0                 j = 1                 j = 2
i = 0    1.000000000000000    1.000000000000000    1.000000000000000
i = 1    1.000000000000000    1.000000000000000    1.000000000000000
i = 2    0.000000000000000    0.000000000000000    0.000000000000000

after binding C gird at point 3, DXF(A) =
[k = 0]
          j = 0                 j = 1                 j = 2
i = 0   20.000000000000000    0.000000000000000    0.000000000000000
i = 1   10.000000000000000   10.000000000000000   10.000000000000000
i = 2    0.000000000000000    0.000000000000000    0.000000000000000
[k = 1]
          j = 0                 j = 1                 j = 2
i = 0   20.000000000000000    0.000000000000000    0.000000000000000
i = 1   10.000000000000000   10.000000000000000   10.000000000000000
i = 2    0.000000000000000    0.000000000000000    0.000000000000000
[k = 2]
          j = 0                 j = 1                 j = 2
i = 0   20.000000000000000    0.000000000000000    0.000000000000000
i = 1   10.000000000000000   10.000000000000000   10.000000000000000
i = 2    0.000000000000000    0.000000000000000    0.000000000000000
```

4. Examples and Applications

Here we present the examples and applications of OpenArray, including four benchmark kernel cases and a three-dimensional ocean model, all the source code included in manual/oa_main.F90 is available in the OpenArray's repository on the

GitHub.

4.1 Four benchmark kernel cases

Four typical PDEs are chosen as benchmark suites to test the performance of OpenArray, including the 2D continuity equation, the 2D heat diffusion equation, the 2D Hotspot, and the 3D Hotspot.

Run the cases:
Complete source code of the four benchmark cases are also included in the oa_main.F90. Make sure the corresponding subroutines are called in the main program, then compile and run with the following command:

```
// compile and link
make –f makefile.intel oalib_obj
make –f makefile.intel manual_main
// run with a single process
mpirun –n 1 ./manual_main
// run with 2 processes
mpirun –n 2 ./manual_main
```

4.1.1 Continuity Equation

The operator expression form:

$$\eta_{t+1} = \eta_{t-1} - 2 * dt * \left(\delta_x^f(\bar{D}_x^b * U) + \delta_y^f(\bar{D}_y^b * V)\right)$$

Source code using OpenArray:

```
subroutine continuity(nt, nx, ny, nz)
  implicit none
  type(array) :: D, U, V ,E
  real*8:: dt
  integer,intent(in) :: nx, ny, nz, nt
  integer :: k

  ! initialize data with random numbers
  D = rands(nx, ny, nz, dt=OA_DOUBLE)
  U = D
  V = D
  E = D
  dt = 0.1

  call tic("continuity")
  do k=1,nt
    E = E - 2*dt*(DXF(AXB(D)*U)+DYF(AYB(D)*V))
  enddo
  call toc("continuity")
end subroutine
```

4.1.2 Heat Diffusion Equation

The operator expression form:

$$T_{t+1} = T + 2 * dt * alpha * \left((\delta_x^f(T) - \delta_x^b(T))/dx^2 + (\delta_y^f(T) - \delta_y^b(T))/dy^2\right)$$

Source code**:**

```
subroutine heat_diffusion(nt, nx, ny, nz)
  implicit none
```

```
    type(array):: T
    real*8 :: dt, dx, dy, alpha
    integer, intent(in) :: nx, ny, nz, nt
    integer :: k

    dx = 0.1
    dy = 0.1
    dt = 0.1
    alpha = 0.1
    T = rands(nx, ny, nz, dt=OA_DOUBLE)

    call tic("heat_diffusion")
    do k=1,nt
        T=T+dt*alpha*2*((DXF(T)-DXB(T))/(dx*dx)+(DYF(T)-DYB(T))/(dy*dy))
    enddo
    call toc("heat_diffusion")
  end subroutine
```

4.1.3 Hotspot2D

The operator expression form:

$$T_{t+1} = T + cap * \left(P + (\delta_x^f(T) - \delta_x^b(T)) * Ry + (\delta_y^f(T) - \delta_y^b(T)) * Rx + (amb\_temp - T) * Rz\right)$$

Source code:

```
  subroutine hotspot2D(nt, nx, ny, nz)
    implicit none
    type(array) :: T, P

...
! The section of the coefficient initialization coefficient is omitted here. See source code in GitHub for details.
...

    T = consts_double(nx,ny,nz,233.3D0,1)
    P = consts_double(nx,ny,nz,233.3D0,1)

    call tic("hotspot2D")
    do i=1,nt
       T=T+cap*(P+(DXF(T)-DXB(T))*Ry+(DYF(T)-DYB(T))*Rx+(amb_temp-T)*Rz)
    enddo
    call toc("hotspot2D")
  end subroutine
```

4.1.4 Hotspot3D

The operator expression form:

$$T_{t+1} = cx * (\bar{T}_x^f + \bar{T}_x^b) + cy * (\bar{T}_y^f + \bar{T}_y^b) + cz * (\bar{T}_z^f + \bar{T}_z^b) + (dt/Cap) * P$$

Source code:

```
subroutine hotspot3D(nt, nx, ny, nz)
    implicit none
    type(array) :: T, P

...
! The section of the coefficient initialization coefficient is omitted here. See source code in github for details.
...

    T = consts_double(nx,ny,nz,233.3D0,1)
    P = consts_double(nx,ny,nz,233.3D0,1)

    call tic("hotspot3D")
    do i=1,nt
      T = cx*(AXF(T)+AXB(T)) + cy*(AYF(T)+AYB(T)) + cz*(AZF(T)+AZB(T))+(dt/Cap)*P &
            + cz*amb_temp
    enddo
```

```
    call toc("hotspot3D")
end subroutine
```

4.2 A three-dimensional ocean model

4.2.1 Introduction

To test the capability and efficiency of OpenArray, we further developed a three-dimensional (3D) ocean model based on the Princeton Ocean Model (POM, Blumberg and Mellor, 1987), The new model is called the Generalized Operator Model of the Ocean (GOMO). The source code is available in the GitHub repository named GOMO (*https://github.com/hxmhuang/GOMO*). GOMO features bottom-following, free-surface, mode-splitting, and staggered grid. The details of the continuous governing equations, the corresponding operator expression form and the descriptions of all the variables used in GOMO are located at *docs* folder in the same repository.

4.2.2 Installation

GOMO is composed of 42 Fortran files (.F90), a header file (.h), and a single namelist file (.txt). Before compiling GOMO, the following software and libraries are required:

1) PnetCDF (version 1.7.0 or later).
2) OpenArray (version 1.0).

If the installation of OpenArray is done, it is fairly easy to build GOMO. The following instructions take you through a sequence of steps to get the default configuration of GOMO up and running. Please use the same compilers to build PnetCDF, OpenArray, and GOMO.

(a) Download GOMO.

```
wget https://github.com/hxmhuang/GOMO/archive/master.zip
unzip master.zip
cd GOMO-master
```

(b) Set the path to PnetCDF and OpenArray, if you install PnetCDF and OpenArray in the default directory, *${HOME}/install*, then:

```
export OPENARRAY_DIR=${HOME}/install
export PNETCDF_DIR=${HOME}/install
```

(c) Compile GOMO. The compilers to build PnetCDF, OpenArray, and GOMO should be set to the same. If PnetCDF and OpenArray are built by gcc with MPICH or gcc with Openmpi, then:

```
./configure MPICC=mpicc MPICXX=mpicxx MPIF77=mpif77 MPIF90=mpif90
make
```

If PnetCDF and OpenArray are built by Intel compilers, then:

./configure MPICC=mpiicc MPICXX=mpiicpc MPIF77=mpiifort MPIF90=mpiifort

make

After compiling, the executable file ***./bin/GOMO*** will be generated. Within the directory ***./bin*** where GOMO and config.txt files exist, type ***./GOMO*** or ***mpirun -np N ./GOMO*** to run an application.

Pre-processing package written in Matlab is used to produce the input file for the ideal test--seamount. The default input file ***seamount65_49_21.nc*** is located at the directory ***./bin/data***. If you want to set a new input file, just follow the commands:

1) Modify the parameters in the init_constants.m for your case.
2) Run the main function, run_preprocess.m.

After running, the output file seamountim_jm_kb.nc will be generated, then move the output file to the folder ./bin/data/ to start your own case.