# **PSyclone Documentation**

Release 1.6.0

Rupert Ford, Joerg Henrichs, Iva Kavcic and Andrew Porter

## **CONTENTS**

1	Getti	Getting Going 3				
	1.1	Download	3			
	1.2	Dependencies	3			
		1.2.1 System-specific set-up	4			
		1.2.2 fparser	4			
		1.2.3 pyparsing	4			
		1.2.4 graphviz	5			
		1.2.5 termcolor	5			
		1.2.6 py.test	5			
	1.3	Environment	6			
	1.4	Configuration	6			
	1.5	Test	7			
	1.6	Run	7			
2	<b>PSyc</b>	lone Script	11			
	2.1	Running	11			
	2.2	Basic Use	12			
	2.3	Choosing the API	12			
	2.4	File output	12			
	2.5	Algorithm files with no invokes	12			
	2.6	Kernel directory	13			
	2.7	Transformation script	14			
	2.8	Fortran line length	14			
	2.9		14			
	2.10	Automatic Profiling Instrumentation	14			
3	Kern		15			
	3.1		15			
	3.2	Metadata	16			
4			17			
	4.1		17			
			18			
	4.2	Limitations	18			
5	PSy l	93/07	21			
J	5.1		21			
	5.2		21			
	5.3		24			
	5.4		24 25			
	J.4	Schedule	۷)			

	5.5	Schedule visualisation	. 25				
6	Tran	sformations 29					
	6.1	Finding	. 29				
	6.2	Available					
	6.3	Kernels	. 37				
	6.4	Applying					
		6.4.1 Interactive					
		6.4.2 Script					
	6.5	OpenMP					
	0.5	6.5.1 Reductions					
		6.5.2 Restrictions					
		0.3.2 Restrictions	. 71				
7	Distr	ibuted Memory	43				
	7.1	Correctness					
	7.2	Performance					
	7.3	Implementation					
	7.4	Control					
	7.5	Status					
	1.5	Status					
8	Built	-ins	45				
_	8.1	Example					
	8.2	Supported Built-in operations					
	8.3	Adding new additional Built-in operations					
	0.5	rading new additional Built in operations	. 40				
9	Dyna	mo0.3 API	51				
	9.1	Algorithm	. 51				
		9.1.1 Example					
		9.1.2 Field					
		9.1.3 Field Vector					
		9.1.4 Scalar					
		9.1.5 Operator					
		9.1.6 Column-Wise Operator					
		9.1.7 Quadrature					
		9.1.8 Stencils					
	0.0	9.1.9 Inter-grid					
	9.2	PSy-layer					
	0.2	9.2.1 Module name					
	9.3	Kernel					
		9.3.1 Rules for all User-Supplied Kernels					
		9.3.2 Rules specific to General-Purpose Kernels without CMA Operators					
		9.3.3 Rules for Kernels that work with CMA Operators					
		9.3.4 Rules for Inter-Grid Kernels					
		9.3.5 Metadata					
		9.3.6 Subroutine					
	9.4	Built-ins	. 70				
		9.4.1 Naming scheme	. 70				
		9.4.2 Addition	. 71				
		9.4.3 Subtraction	. 73				
		9.4.4 Multiplication	. 74				
		9.4.5 Scaling	. 75				
		9.4.6 Division					
		9.4.7 Setting to value					
		9.4.8 Raising to power					
		9.4.9 Inner product					
			. 0				

			78
	9.5		79
	9.6	Conventions	79
	9.7	Configuration	3(
		9.7.1 Annexed DoFs	3(
	9.8	Transformations	3(
10	GOce	ean1.0 API	35
	10.1	Introduction	33
	10.2	The GOcean Library	35
		10.2.1 Grid	35
		10.2.2 Fields	37
		10.2.3 Example	37
	10.3		39
		$\mathcal{E}$	39
	10.4		)(
			)(
			):
	10.5		)(
	10.6		)(
	10.7		97
	10.7		) }
	10.8	1	) }
			,
11		O API	
		Algorithm	
	11.2	Constructing the PSyIRe	
	11.0	11.2.1 Loops	
		Example	
	11.4	Limitations	):
12		Generation 10	
	12.1	Quick Start	
		Introduction	
	12.3	Use	
		Kernels	
		Example	
	12.6	Errors	(
13	Line l	length 11	11
	13.1	Script	[ ]
	13.2	Interactive	[ ]
	13.3	Limitations	12
14	API	11	13
	14.1	The generator module	
	14.2	The parse module	
	14.3	The transformations module	
	14.4	The psyGen module	
	14.5	The algGen module	
	14.5	The line_length module	
15	Profil		
	15.1	Profiling API	
		15.1.1 ProfileInit()	3]

		15.1.2	ProfileFinalise()	131
		15.1.3	ProfileStart()/ProfileEnd()	132
	15.2		g Command Line Options	
	15.3	Profilin	g in Scripts - ProfileRegionTransform	134
	15.4	Interfac	te to Third Party Profiling Tools	134
16 System-specific set-up				135
10			t-up	135
	10.1	16.1.1	Installing dependencies	
		16.1.2	Installing PSyclone	
			Common installation	
	16.2		per set-up	
	10.2	16.2.1	Installing PSyclone From Git	
		16.2.2	Installing documentation tools	
		16.2.3	Installing testing tools	
		10.2.3	mouning coming tools	150
<b>17</b>		iguratio		141
	17.1		8	141
		17.1.1	DEFAULT Section	
		17.1.2	dynamo0.3 Section	142
		17.1.3	gocean1.0 Section	142
18	Devel	lopers' g	mide	145
10	18.1		g With PSyclone from GitHub	145
	10.1		Installation	
		18.1.2	Test Suite	
		18.1.3	Continuous Integration	147
		18.1.4	Code Review	
	18.2	Generic	Code	148
		18.2.1		148
	18.3	New Al	PIs	
	18.4		g API's	
			Dynamo0.3	
			GOcean1.0	
	18.5		88	
		18.5.1	Module: f2pygen	
		18.5.2	Module: configuration	
		18.5.3	Module: transformations	
Da	liaaua	anhe.		167
DIL	oliogra	apily		167
Python Module Index				169
Ind	Index			

PSyclone, the PSy code generator, is being developed for use in finite element, finite volume and finite difference codes. PSyclone development started with the aim to support the emerging API in the GungHo project for a finite element dynamical core.

The GungHo project was initiated in 2011 to address challenges of weather and climate prediction on the next generation of supercomputers. The project ran for 5 years as a collaboration between the Met Office, NERC (via NERC funded academics) and STFC. It laid a foundation for redesign of the heart of the Met Office's Unified Model, known as the dynamical core, from the choices of numerical methods and model grids to the implementation of parallel algorithms that will scale to millions of cores.

The software infrastructure based on the GungHo project recommendations is now being developed in the LFRic project and is expected to be operational in 2022. Its development is led by the requirements to support multiple meshes and element types, thus allowing for future model development.

GungHo also proposed a novel separation of concerns for the software implementation of the dynamical core. This approach distinguishes between three layers: the Algorithm layer, the Kernel layer and the Parallelisation System (PSy) layer. Together this separation is termed PSyKAl.

The Algorithm layer specifies the algorithm that the scientist would like to run (in terms of calls to kernel routines and built-in operations) and logically operates on full fields.

The Kernel layer provides the implementation of the code kernels as subroutines. These subroutines operate on local fields (a set of elements, a vertical column, or a set of vertical columns, depending on the kernel).

The PSy layer sits in-between the algorithm and kernel layers and its primary role is to provide node-based parallel performance for the target architecture. The PSy layer can be optimised for a particular hardware architecture, such as multi-core, many-core, GPGPUs, or some combination thereof with no change to the algorithm or kernel layer code. This approach therefore offers the potential for portable performance.

Rather than writing the PSy layer manually, the PSyclone code generation system can help a user to optimise the code for a particular architecture (by providing optimisations such as blocking, loop merging, inlining etc), or alternatively, generate the PSy layer automatically.

PSyclone is also being extended to support an API being developed in the GOcean project for two finite difference ocean model benchmarks, one of which is based on the NEMO ocean model.

CONTENTS 1

2 CONTENTS

**CHAPTER** 

ONE

#### **GETTING GOING**

#### 1.1 Download

The following instructions are intended for a PSyclone user who wants to work with a released version of the code. If you are a developer or wish to test a specific branch of PSyclone from the GitHub repository please see *Installation* in the *Developers' guide*.

PSyclone is available on the Python Package Index (pypi.org) and is hosted on GitHub:

```
https://github.com/stfc/PSyclone
```

The latest release is 1.6.0 and the latest stable version is on the master branch.

PSyclone can be installed using pip:

```
> pip install psyclone
```

or downloaded from github - either see 1.6.0 in the releases tab on the PSyclone page or download and extract the latest release of PSyclone directly, e.g.

```
> wget https://github.com/stfc/PSyclone/archive/1.6.0.tar.gz
> gunzip 1.6.0.tar.gz
> tar xf 1.6.0.tar
> rm 1.6.0.tar
> ls
PSyclone-1.6.0
```

Hereon the location where you download or clone PSyclone (including the PSyclone directory itself) will be referred to as <PSYCLONEHOME>.

## 1.2 Dependencies

PSyclone is written in Python so needs Python to be installed on the target machine. PSyclone has been tested under Python 2.6.5, 2.7.3 and 3.6.

**Warning:** As of version 1.6, PSyclone requires version 0.0.7 or greater of fparser.

PSyclone immediately relies on four external Python packages; six, configparser, fparser and pyparsing. In order to run the test suite py.test is required. The easiest way to satisfy the Python dependencies is to use the Python Package Index (pypi.org) and pip. See https://packaging.python.org/installing/ for more information.

If everything is working correctly then using pip to install PSyclone:

```
> pip install psyclone
```

will automatically install the Python dependencies (fparser and pyparsing).

In addition to the mandatory dependencies just described, PSyclone also has optional dependencies on both graphviz and termcolor. PSyclone can use graphviz to produce a visualisation of a schedule's dependency graph. If this is desired then the Python package graphviz (for the Python bindings) as well as the graphviz package itself must be installed. If the graphviz package is not available then the associated PSyclone routines will return silently and no visualisations will be produced. The Python package termcolor is used for pretty-printing a schedule in terminals that support coloured text. If the package is not available then the schedule is simply printed in plain text without colour highlighting.

#### 1.2.1 System-specific set-up

System-specific set-up instructions are available for Ubuntu 14.04.2 and OpenSUSE 42.2.

#### 1.2.2 fparser

The fparser package (https://github.com/stfc/fparser) is a Fortran parser originally developed as a part of the f2py project.

The minimum version of fparser required by PSyclone is currently 0.0.7 but we strongly recommend you install the latest version to reduce the chance of encountering problems when parsing existing algorithm or kernel code. (Note that for older versions of PSyclone up to and including 1.5.1 you must use version 0.0.6 of fparser.)

fparser is available from the Python Package Index and thus may be installed using pip (https://packaging.python.org/installing/#requirements-for-installing-packages):

```
> pip install fparser
```

If you do not have sufficient permissions to perform a system-wide install then you can instruct pip to do a user-local install:

```
> pip install --user fparser
```

Should you wish to remove fparser then simply do:

```
> pip uninstall fparser
```

If you have already installed fparser and want to upgrade to the latest version simply do:

```
> pip install fparser --upgrade
```

#### 1.2.3 pyparsing

PSyclone requires pyparsing, a library designed to allow parsers to be be built in Python. PSyclone uses pyparsing to parse fortran regular expressions as fparser does not fully parse these, (see http://pyparsing.wikispaces.com for more information).

PSyclone has been tested with pyparsing versions 1.5.2, 2.0.1 and 2.2.0.

You can test whether pyparsing is already installed on your machine by typing import pyparsing from the python command line. If pyparsing is installed, this command will complete successfully. If pyparsing is installed you can check its version by typing pyparsing. \_\_version\_\_ after successfully importing it.

If pyparsing is not installed on your system then it may be installed from the Python Package Index using pip:

```
> pip install pyparsing
```

Should you wish to, uninstalling is simply performed by doing:

```
> pip uninstall pyparsing
```

If you do not have sufficient privileges for a system-wide install then you can instruct pip to do a user-local install:

```
> pip install --user pyparsing
```

Alternatively, you could follow the instructions here http://pyparsing.wikispaces.com/Download+and+Installation.

#### 1.2.4 graphviz

The data dependencies of a PSy-layer schedule (see Section *Schedule*) determine the validity of changes to a schedule. PSyclone supports the visualisation of these dependencies as a graph using graphviz. This visualisation is not needed to use PSyclone.

If the Python bindings to graphviz are not installed on your system then it may be installed from the Python Package Index using pip:

```
> sudo pip install graphviz
```

Should you wish to, uninstalling is simply performed by doing:

```
> sudo pip uninstall graphviz
```

If you do not have sufficient privileges for a system-wide install then you can instruct pip to do a user-local install:

```
> pip install --user graphviz
```

If graphviz itself is not installed on your system and your system supports the apt package manager then see below, otherwise please refer to the download and install instructions which are available here http://www.graphviz.org/Download.php.

If your system supports the apt package manager then it can be installed and removed in the following way:

```
> sudo apt install graphviz
> sudo apt remove graphviz
```

#### 1.2.5 termcolor

By default, the <code>view()</code> method of a <code>schedule</code> object (representing the schedule of a PSy-layer routine) prints a plain-text representation to standard-out. However, if the <code>termcolor</code> package is available then PSyclone uses this to add colour highlighting to the output text.

Installation (and uninstallation) of this package can be done via pip in exactly the same way as for graphviz, as described above.

#### 1.2.6 py.test

The PSyclone test suite uses py.test. This is not needed to use PSyclone but is useful to check whether PSyclone is working correctly on your system. You can test whether it is already installed by simply typing py.test at a shell prompt. If it is present you will get output that begins with

1.2. Dependencies 5

```
========= test session starts ========
```

If you do not have it then py.test can again be installed using pip or from here http://pytest.org/latest/ (or specifically here http://pytest.org/latest/getting-started.html).

#### 1.3 Environment

In order to use PSyclone (including running the test suite and building documentation) you will need to install it. The simplest way to do this is to use pip with the supplied setup.py file:

```
> cd <PSYCLONEHOME>
> pip install .
```

By default pip will attempt a system-wide install. If you wish to do a user-local install instead then supply the --user flag:

```
> pip install --user .
```

This installs the PSyclone modules in ~/.local/lib/pythonX.Y/site-packages (where X.Y is the version of Python that you are using) and the 'psyclone' script in ~/.local/bin. Depending on your linux distribution, you may need to add the latter location to your \$PATH.

If for some reason you'd rather not use pip then you can run the setup manually:

```
> python setup.py install
```

or, if you don't have root access:

```
> python setup.py install --user
```

or,

```
> python setup.py install --prefix /my/install/path
```

If using the latter method then it will be necessary to take further action to ensure PSyclone can find the configuration file installed as a part of this process - see below.

## 1.4 Configuration

Various aspects of PSyclone are configured through a configuration file, psyclone.cfg. The default version of this file is installed to <python-base-prefix>/shared/psyclone/ during the installation process. If a system-wide installation is being performed then this will be something like /usr/share/psyclone/. If a user-local installation is performed (--user flag to pip install) then the location will be something like ~/.local/share/psyclone/.

**Warning:** if PSyclone is installed to a non-standard location (e.g. by specifying the --prefix=/some/path option to pip install) then PSyclone will not be able to find the configuration file at execution time. There are two solutions to this: 1. copy the configuration file to a location where PSyclone will find it (see *Configuration*) or 2. set the PSYCLONE\_CONFIG environment variable to the full-path to the configuration file, e.g.:

```
> export PSYCLONE_CONFIG=/some/path/PSyclone/config/psyclone.cfg
```

**Warning:** when installing in 'editable' mode (-e flag to pip), pip does *not* install the configuration file. You will have to take one of the two actions described above.

See *Configuration* for details of the settings contained within the config file.

#### 1.5 Test

Once you have the necessary dependencies installed and your environment configured, you can check that things are working by using the PSyclone test suite. These tests are not required and can be skipped if preferred:

```
> cd <PSYCLONEHOME>/src/psyclone/tests
> py.test
```

If everything is working as expected then you should see output similar to:

Most of the tests use Fortran source files in the <PSYCLONEHOME>/src/psyclone/tests directory and many of them can be compiled during the testing process. To enable compilation testing run:

```
> py.test --compile --f90="<compiler_name>" --f90flags="<compiler_flags_list>"
```

"<compiler\_name>" and "<compiler\_flags\_list>" are optional arguments. The default value for "<compiler\_name>" is "gfortran" and there are no defaults for the "<compiler\_flags\_list>". Please note that the onus is on the user to provide correct values for these options.

#### 1.6 Run

You are now ready to try running PSyclone on the examples. One way of doing this is to use the psyclone driver script. Assuming it is on your PATH:

1.5. Test 7

```
filename psyclone: error: too few arguments
```

As indicated above, the psyclone script takes the name of the Fortran source file containing the algorithm specification (in terms of calls to invoke()). It parses this, finds the necessary kernel source files and produces two Fortran files. The first contains the PSy, middle layer and the second a re-write of the algorithm code to use that layer. These files are named according to the user-supplied arguments (options -oalg and -opsy). If those arguments are not supplied then the script writes the generated/re-written Fortran to the terminal.

Examples are provided in the examples directory. There are 3 subdirectories (dynamo, gocean and gunghoproto) corresponding to different API's that are supported by PSyclone. In this case we are going to use one of the dynamo examples

```
> cd <PSYCLONEHOME>/examples/dynamo/eg1
> psyclone -api dynamo0.1 \
> -oalg dynamo_alg.f90 -opsy dynamo_psy.f90 dynamo.F90
```

You should see two new files created called dynamo\_alg.f90 (containing the re-written algorithm layer) and dynamo\_psy.f90 (containing the generated PSy- or middle-layer). Since this is a dynamo example the Fortran source code has dependencies on the dynamo system and therefore cannot be compiled stand-alone.

You can also use the runme.py example to see the interactive API in action. This script contains:

```
from psyclone.parse import parse
from psyclone.psyGen import PSyFactory
# This example uses version 0.1 of the Dynamo API
api="dynamo0.1"
# Parse the file containing the algorithm specification and
# return the Abstract Syntax Tree and invokeInfo objects
ast,invokeInfo=parse("dynamo.F90",api=api)
# Create the PSy-layer object using the invokeInfo
psy=PSyFactory(api).create(invokeInfo)
# Generate the Fortran code for the PSy layer
print psy.gen
# List the invokes that the PSy layer has
print psy.invokes.names
# Examine the 'schedule' (e.g. loop structure) that each
schedule=psy.invokes.get('invoke_0_v3_kernel_type').schedule
schedule.view()
schedule=psy.invokes.get('invoke_1_v3_solver_kernel_type').schedule
schedule.view()
```

It can be run non-interactively as follows:

```
> cd <PSYCLONEHOME>/example/dynamo/eg1
> python runme.py
```

However, to understand this example in more depth it is instructive to cut-and-paste from the runme.py file into your own, interactive python session:

```
> cd <PSYCLONEHOME>/example/dynamo/eg1
> python
```

In addition to the runme.py script, there is also runme\_openmp.py which illustrates how one applies an OpenMP transform to a loop schedule within the PSy layer. The initial part of this script is the same as that of runme.py (above) and is therefore omitted here:

```
# List the various invokes that the PSy layer contains
print (psy.invokes.names)
# Get the loop schedule associated with one of these
# invokes
schedule = psy.invokes.get('invoke_v3_kernel_type').schedule
schedule.view()
# Get the list of possible loop transformations
from psyclone.psyGen import TransInfo
t = TransInfo()
print(t.list)
# Create an OpenMPLoop-transformation object
ol = t.get_trans_name('OMPLoopTrans')
# Apply it to the loop schedule of the selected invoke
new_schedule, memento = ol.apply(schedule.children[0])
new_schedule.view()
# Replace the original loop schedule of the selected invoke
# with the new, transformed schedule
psy.invokes.get('invoke_v3_kernel_type')._schedule = new_schedule
# Generate the Fortran code for the new PSy layer
print (psy.gen)
```

1.6. Run 9

#### **PSYCLONE SCRIPT**

The simplest way to run PSyclone is to use the psyclone script. If you installed PSyclone using pip then this script should be available on your PATH (see *Environment* for more details). Alternatively it can be found in the <PSYCLONEHOME>/bin directory. The script takes an algorithm file as input and outputs modified algorithm code and generated PSy code. This section walks through its functionality.

## 2.1 Running

The psyclone script is executable and designed to be run from the command line, e.g.:

```
> psyclone <args>
```

The optional -h argument gives a description of the options provided by the script:

```
> psyclone -h
usage: psyclone [-h] [-oalg OALG] [-opsy OPSY] [-api API] [-s SCRIPT]
                [-d DIRECTORY] [-1] [-dm] [-nodm]
                [--profile {invokes, kernels}]
                [--force-profile {invokes, kernels}] [-v] filename
Run the PSyclone code generator on a particular file
positional arguments:
 filename
                        algorithm-layer source code
optional arguments:
 -h, --help
                       show this help message and exit
                      filename of transformed algorithm code
 -oalg OALG
 -opsy OPSY
                      filename of generated PSy code
 -api API
                       choose a particular api from ['gunghoproto',
                        'dynamo0.1', 'dynamo0.3', 'gocean0.1', 'gocean1.0'],
                        default 'dynamo0.3'.
 -s SCRIPT, --script SCRIPT
                        filename of a PSyclone optimisation script
 -d DIRECTORY, --directory DIRECTORY
                       path to root of directory structure containing kernel
                        source code
 -1, --limit
                       limit the fortran line length to 132 characters
 -dm, --dist_mem
                       generate distributed memory code
 -nodm, --no_dist_mem do not generate distributed memory code
 --profile {invokes, kernels}, -p {invokes, kernels}
                        Add profiling hooks for either 'kernels' or 'invokes'
```

```
--force-profile {invokes, kernels}

Add profiling hooks for either 'kernels' or 'invokes'
even if a transformation script is used. Use at your
own risk.

-v, --version Display version information (1.6.0)
```

#### 2.2 Basic Use

The simplest way to use psyclone is to provide it with an algorithm file:

```
> psyclone alg.f90
```

If the algorithm file is invalid for some reason, the script should return with an appropriate error. For example, if we use the Python genkernelstub script as an algorithm file we get the following:

```
> psyclone <PSYCLONEHOME>/bin/genkernelstub
...
1:#!/usr/bin/env python <== no parse pattern found for "#" in 'BeginSource' block.
'Parse Error: Fatal error in external fparser tool'
```

If the algorithm file is valid then the modified algorithm code and the generated PSy code will be output to the terminal screen.

## 2.3 Choosing the API

In the previous section we relied on PSyclone using the default API. The default API, along with the supported APIs can be seen by running the psyclone script with the -h option.

If you use a particular API frequently and it is not the default then you can change the default by creating a copy of the default psyclone.cfg file and editing it. See *Configuration* for more details.

If your code uses an API that is different to the default then you can specify this as an argument to the psyclone script.

```
> psyclone -api dynamo0.1 alg.f90
```

## 2.4 File output

By default the modified algorithm code and the generated PSy code are output to the terminal. These can instead be output to files by using the -oalg <file> and -opsy <file> options, respectively. For example, the following will output the generated PSy code to the file 'psy.f90' but the algorithm code will be output to the terminal:

```
> psyclone -opsy psy.f90 alg.f90
```

## 2.5 Algorithm files with no invokes

If the psyclone script is provided with a file that contains no invoke calls then the script outputs a warning to stdout and copies the input file to stdout, or to the specified algorithm file (if the -oalg <file> option is

used). No PSy code will be output. If a file is specified using the -opsy <file> option this file will not be created.

```
> psyclone -opsy psy.f90 -oalg alg_new.f90 empty_alg.f90
Warning: 'Algorithm Error: Algorithm file contains no invoke() calls: refusing to generate empty PSy code'
```

### 2.6 Kernel directory

When an algorithm file is parsed, the parser looks for the associated kernel files. The way in which this is done requires that any user-defined kernel routine (as opposed to *Built-ins*) called within an invoke must have an explicit use statement. For example, the following code gives an error:

```
> cat no_use.f90
program no_use
    call invoke(testkern_type(a,b,c,d,e))
end program no_use
> psyclone -api gocean1.0 no_use.f90
"Parse Error: kernel call 'testkern_type' must either be named in a use statement or_
    →be a recognised built-in (one of '[]' for this API)"
```

(If the chosen API has any *Built-ins* defined then these will be listed within the [] in the above error message.) If the name of the kernel is provided in a use statement then the parser will look for a file with the same name as the module in the use statement. In the example below, the parser will look for a file called "testkern.f90" or "testkern.F90":

```
> cat use.f90
program use
  use testkern, only : testkern_type
  call invoke(testkern_type(a,b,c,d,e))
end program use
```

Therefore, for PSyclone to find kernel files, the module name of a kernel file must be the same as its filename. By default the parser looks for the kernel file in the same directory as the algorithm file. If this file is not found then an error is reported.

```
> psyclone use.f90
Kernel file 'testkern.[fF]90' not found in <location>
```

The -d option can be used to tell psyclone where to look for kernel files by supplying it with a directory. The script will recurse from the specified directory path to look for the required file. There must be only one instance of the specified file within (or below) the specified directory:

```
> cd <PSYCLONEHOME>/src/psyclone
> psyclone -d . use.f90
More than one match for kernel file 'testkern.[fF]90' found!
> psyclone -d tests/test_files/dynamo0p3 -api dynamo0.3 use.f90
[code output]
```

**Note:** The -d option is limited to a single directory. Therefore a current limitation in PSyclone is that all kernel files required by an algorithm file must exist within a directory hierarchy where their file names are unique.

## 2.7 Transformation script

By default the psyclone script will generate 'vanilla' PSy layer code. The -s option allows a Python script to be specified which can transform the PSy layer. This option is discussed in more detail in the *Script* section.

### 2.8 Fortran line length

By default the psyclone script will generate fortran code with no consideration of Fortran line-length limits. As the line-length limit for free-format Fortran is 132 characters, the code that is output may be non-conformant.

Line length is not an issue for many compilers as they allow compiler flags to be set which allow lines longer than the Fortran standard. However this is not the case for all compilers.

When the -1 option is specified to the psyclone script, the output will be line wrapped so that the output lines are always within the 132 character limit.

The -1 option also checks the parsed algorithm and kernel files for conformance and raises an error if they do not conform.

Line wrapping is not performed by default. There are two reasons for this. This first reason is that most compilers are able to cope with long lines. The second reason is that the line wrapping implementation could fail in certain pathological cases. The implementation and limitations of line wrapping are discussed in the *Limitations* section.

## 2.9 Distributed memory

By default the psyclone script will generate distributed memory (DM) code (i.e. parallelised using MPI). As with the choice of API, this default may be configured by editing psyclone.cfg - see *Configuration*. Alternatively, whether or not to generate DM code can be specified as an argument to the psyclone script using the -dm/--dist\_mem or -nodm/--no\_dist\_mem flags, respectively.

For details of PSyclone's support for generating DM code see *Distributed Memory*.

## 2.10 Automatic Profiling Instrumentation

The <code>--profile</code> option allows the user to instruct PSyclone to automatically insert profiling calls within the generated PSy code. Two options are provided, <code>invokes</code> and <code>kernels</code>. The first of these causes PSyclone to insert profiling-start and -stop calls at the beginning and end of every generated invoke routine. The second puts profiling calls around every kernel call (including the associated loops). The generated code must be linked against the PSyclone profiling interface and the profiling tool itself. The application that calls the PSyclone-generated code is responsible for initialising and finalising the profiling library that is being used. For full details on the use of this profiling functionality please see the <code>Profiling</code> section.

**CHAPTER** 

THREE

#### **KERNEL LAYER**

In the PSyKAl separation of concerns, Kernel code (code which is created to run within the Kernel layer), works over a subset of a field (such as a column). The reason for doing this is that it gives the PSy layer the responsibility of calling the Kernel over the spatial domain which is where parallelism is typically exploited in finite element and finite difference codes. The PSy layer is therefore able to call the kernel layer in a flexible way (blocked and/or in parallel for example). Kernel code in the kernel layer is not allowed to include any parallelisation calls or directives and works on raw fortran arrays (to allow the compiler to optimise the code).

Since a Kernel is called over the spatial domain (by the PSy layer) it must take at least one field or operator as an argument.

#### 3.1 API

Kernels in the kernel layer are implemented as subroutines within fortran modules. One or more kernel modules are allowed, each of which can contain one or more kernel subroutines. In the example below there is one module integrate\_one\_module which contains one kernel subroutine integrate\_one\_code. The kernel subroutines contain the code that operates over a subset of the field (such as a column).

Metadata describing the kernel subroutines is required by the PSyclone system to generate appropriate PSy layer code. The metadata is written by the kernel developer and is kept with the kernel code in the same module using a sub-type of the kernel\_type type. In the example below the integrate\_one\_kernel type specifies the appropriate metadata information describing the kernel code for the gunghoproto api.

```
module integrate_one_module
 use kernel_mod
 implicit none
 private
 public integrate_one_kernel
 public integrate_one_code
 type, extends(kernel_type) :: integrate_one_kernel
   type(arg) :: meta_args(2) = (/&
         arg(READ, (CG(1)*CG(1))**3, FE), &
         arg(SUM, R, FE)/)
    integer :: ITERATES_OVER = CELLS
    contains
    procedure, nopass :: code => integrate_one_code
 end type integrate_one_kernel
contains
 subroutine integrate_one_code(layers, pldofm, X, R)
```

```
integer, intent(in) :: layers
integer, intent(in) :: pldofm(6)
real(dp), intent(in) :: X(3,*)
real(dp), intent(inout) :: R
end subroutine integrate_one_code
end module integrate_one_module
```

#### 3.2 Metadata

Kernel metadata is not required if the PSy layer is going to be written manually, its sole purpose is to let PSyclone know how to generate the PSy layer. The content of Kernel metadata differs depending on the particular API and this information can be found in the API-specific sections of this document.

In all API's the kernel metadata is implemented as an extension of the *kernel\_type* type. The reason for using a type to specify metadata is that it allows the metadata to be kept with the code and for it to be compilable. In addition, currently all API's will contain information about the arguments in an array called meta\_args, a specification of what the kernel code iterates over in a variable called iterates\_over and a reference to the kernel code as a type bound procedure.

```
type, extends(kernel_type) :: integrate_one_kernel
...
  type(...) :: meta_args(...) = (/ ... /)
...
  integer :: ITERATES_OVER = ...
   contains
...
  procedure ...
  end type integrate_one_kernel
```

**CHAPTER** 

**FOUR** 

#### ALGORITHM LAYER

In the PSyKAl separation of concerns, the Algorithm layer specifies the algorithm that the scientist would like to run (in terms of calls to kernel routines and Built-in operations) and logically operates on full fields. Algorithm code in the algorithm layer is not allowed to include any parallelisation calls or directives and passes datatypes specified by the particular API.

#### 4.1 API

The Algorithm layer is forbidden from calling the Kernel layer directly. In PSyclone, if the programmer would like to call a Kernel routine or a Built-in operation from the algorithm layer they must use the <code>invoke</code> call (which is common to all API's). The <code>invoke</code> call is not necessary (and indeed will not work) if the PSy layer is written manually.

To make an invoke call, the algorithm layer developer adds one or more call invoke () statements to their code and within the content of the invoke call they add a reference to the required Kernel/Built-in and the data to pass to it. For example,

```
call invoke(integrate_one_kernel(arg1,arg2))
...
```

For more information on the concept of Built-in operations see the *Built-ins* Section. Details of which operations are supported for a specific API are given in the documentation of that API.

The algorithm layer can consist of an arbitrary number of files containing fortran code, any of which may contain as many invoke() calls as is required. PSyclone is applied to an individual algorithm layer file and must therefore be run multiple times if multiple files containing invoke() calls exist in the algorithm layer.

The algorithm developer is also able to reference more than one Kernel/Built-in within an invoke. In fact this feature is encouraged for performance reasons. As a general guideline the developer should aim to use as few invokes as possible with as many Kernel references within them as is possible. The reason for this is that it allows for greater freedom for optimisation in the PSy layer as PSy layer optimisations are limited to the contents of individual invoke calls - PSyclone currently does not attempt to optimise the PSy layer over multiple invoke calls.

As well as generating the PSy layer code, PSyclone modifies the Algorithm layer code, replacing invoke calls with calls to the generated PSy layer so that the algorithm code is compilable and linkable to the PSy layer and adding in the appropriate use statement. For example, the above integrate\_one\_kernel invoke is translated into something like the following:

```
use psy, only : invoke_0_integrate_one_kernel
...
call invoke_0_integrate_one_kernel(arg1, arg2)
...
```

You may have noticed from other examples in this guide that an algorithm specification in an invoke call references the metadata type in an invoke call, not the code directly; this is by design.

For example, in the invoke call below, integrate\_one\_kernel is used.

```
call invoke(integrate_one_kernel(arg1,arg2))
...
```

integrate\_one\_kernel is the name of the metadata type in the module, not the name of the subroutine in the Kernel ...

#### 4.1.1 Named Invokes

PSyclone permits the user to optionally specify a label for an invoke call like so:

```
call invoke(integrate_one_kernel(arg1,arg2), & name="compute something")
...
```

The name argument to the invoke call is optional. If supplied it must be a string literal. The content of this string (with any spaces replaced by '\_' characters) is used in naming the corresponding PSy-layer routine generated by PSyclone. So, for the above example, the generated PSy-layer subroutine will be named "invoke\_compute\_something." Each invoke label must currently be unique within an Algorithm source file. Note that, in keeping with the Fortran language, labels are not case sensitive and, after having any spaces replaced by underscores, must be valid Fortran names (e.g. name="compute(1)" is invalid). In the future it is intended that the labelling of invokes will help to support invoke-specific optimisations to be applied as well as enabling more readable profiling output. It may also be used to instruct PSyclone to just generate a single subroutine to implement all invokes that share the same label.

#### 4.2 Limitations

In order to re-write the Algorithm layer, as just described, PSyclone must obviously be able to parse the invoke calls. Since the Fortran expression parser used by PSyclone is relatively simple, this means there are limitations on what Fortran may be used when specifying kernel arguments in an invoke call. Since these limitations can have a direct impact on the natural science code, the PSyclone developers endeavour to keep them to a minimum.

The current list of known limitations/restrictions on the form of kernel arguments within an invoke is:

• No arithmetic expressions (e.g. kernel\_type (a+b) or kernel\_type (-a))

• No named (optional) arguments (e.g. kernel\_type (fn (my\_arg=a)))

If you encounter any other limitations (or have a burning desire to use one of the above forms) then please contact the PSyclone developers.

4.2. Limitations

**CHAPTER** 

**FIVE** 

#### **PSY LAYER**

In the PSyKAl separation of concerns, the PSy layer is responsible for linking together the Algorithm and Kernel layers and for providing the implementation of any Built-in operations used. Its functional responsibilities are to

- 1. map the arguments supplied by an Algorithm invoke call to the arguments required by a Built-in or Kernel call (as these will not have a one-to-one correspondance).
- 2. call any Kernel routines such that they cover the required iteration space and
- 3. perform any Built-in operations (either by including the necessary code directly in the PSy layer or by e.g. calling a maths library) and
- 4. include any required distributed memory operations such as halo swaps and reductions.

Its other role is to allow the optimisation expert to optimise any required distributed memory operations, include and optimise any shared memory parallelism and optimise for single node (e.g. cache and vectorisation) performance.

#### 5.1 Code Generation

The PSy layer can be written manually but this is error prone and potentially complex to optimise. The PSyclone code generation system generates the PSy layer so there is no need to write the code manually.

To generate correct PSy layer code, PSyclone needs to understand the arguments and datatypes passed by the algorithm layer and the arguments and datatypes expected by the Kernel layer; it needs to know the name of the Kernel subroutine(s); it needs to know the iteration space that the Kernel(s) is/are written to iterate over; it also needs to know the ordering of Kernels and Built-ins as specified in the algorithm layer. Finally, it needs to know where to place any distributed memory operations.

PSyclone determines the above information by being told the API in question (by the user), by reading the appropriate Kernel and Built-in metadata and by reading the order of Kernels and Built-ins in an invoke call (as specified in the algorithm layer).

PSyclone has an API-specific parsing stage which reads the algorithm layer and all associated Kernel metadata. This information is passed to a PSy-generation stage which creates a high level view of the PSy layer. From this high level view the PSy-generation stage can generate the required PSy code.

For example, the following Python code shows a code being parsed, a PSy-generation object being created using the output from the parser and the PSy layer code being generated by the PSy-generation object.

```
from parse import parse
from psyGen import PSyFactory

# This example uses version 0.1 of the Dynamo API
api = "dynamo0.1"
```

```
# Parse the file containing the algorithm specification and
# return the Abstract Syntax Tree and invokeInfo objects
ast, invokeInfo = parse("dynamo.F90", api=api)

# Create the PSy-layer object using the invokeInfo
psy = PSyFactory(api).create(invokeInfo)
# Generate the Fortran code for the PSy layer
print psy.gen
```

#### 5.2 Structure

PSyclone provides a hierarchy of base classes which specific API's can subclass to support their particular API. All API's implemented so far, follow this hierarchy.

At the top level is the **PSy** class. The **PSy** class has an **Invokes** class. The **Invokes** class can contain one or more **Invoke** classes (one for each invoke in the algorithm layer). Each **Invoke** class has a **Schedule** class.

The class diagram for the above base classes is shown below using the dynamo0.3 API as an illustration. This class diagram was generated from the source code with pyreverse and edited with inkscape.



#### 5.3 API

```
class psyclone.psyGen.PSy(invoke_info)
```

Base class to help manage and generate PSy code for a single algorithm file. Takes the invocation information output from the function parse.parse() as its input and stores this in a way suitable for optimisation and code generation.

**Parameters** invoke\_info (FileInfo) – An object containing the required invocation information for code optimisation and generation. Produced by the function parse.parse().

For example:

```
>>> import psyclone
>>> from psyclone.parse import parse
>>> ast, info = parse("argspec.F90")
>>> from psyclone.psyGen import PSyFactory
>>> api = "..."
>>> psy = PSyFactory(api).create(info)
>>> print(psy.gen)
```

#### inline (module)

inline all kernel subroutines into the module that are marked for inlining. Avoid inlining the same kernel more than once.

```
{\tt class} \ {\tt psyclone.psyGen.Invokes} \ ({\it alg\_calls, Invoke})
```

Manage the invoke calls

**class** psyclone.psyGen.**Invoke** (alg\_invocation, idx, schedule\_class, reserved\_names=None)

Manage an individual invoke call

```
first_access (arg_name)
```

Returns the first argument with the specified name passed to a kernel in our schedule

```
unique_declarations (datatype, access=None)
```

Returns a list of all required declarations for the specified datatype. If access is supplied (e.g. "gh\_write") then only declarations with that access are returned.

```
unique declns by intent (datatype)
```

Returns a dictionary listing all required declarations for each type of intent ('inout', 'out' and 'in').

**Parameters** datatype (string) – the type of the kernel argument for the particular API for which the intent is required

**Returns** dictionary containing 'intent' keys holding the kernel argument intent and declarations of all kernel arguments for each type of intent

```
Return type dict
```

**Raises** GenerationError – if the kernel argument is not a valid datatype for the particular API.

```
class psyclone.psyGen.Schedule(KernFactory, BuiltInFactory, alg_calls=[])
```

Stores schedule information for an invocation call. Schedules can be optimised using transformations.

```
>>> from parse import parse
>>> ast, info = parse("algorithm.f90")
>>> from psyGen import PSyFactory
>>> api = "..."
>>> psy = PSyFactory(api).create(info)
>>> invokes = psy.invokes
>>> invokes.names
```

```
>>> invoke = invokes.get("name")
>>> schedule = invoke.schedule
>>> schedule.view()
```

#### coloured\_text

Returns the name of this node with appropriate control codes to generate coloured output in a terminal that supports it.

**Returns** Text containing the name of this node, possibly coloured

**Return type** string

#### dag\_name

Return the name to use in a dag for this node

```
view(indent=0)
```

Print a text representation of this node to stdout and then call the view() method of any children.

**Parameters indent** (integer) – Depth of indent for output text

#### 5.4 Schedule

A PSy **Schedule** object consists of a tree of objects (called Nodes in PSyclone) which can be used to describe the required schedule for a PSy layer subroutine. This subroutine is called by the Algorithm layer and itself calls one or more Kernels and/or implements any required Built-in operations. The Node objects can currently be a **Loop**, a **Kernel**, a **Built-in** (see the *Built-ins* section), a **Directive** (of various types), a **HaloExchange**, or a **GlobalSum** (the latter two are only used if distributed memory is supported and is switched on; see the *Distributed Memory* section). The order of the tree (depth first) indicates the order of the associated Fortran code.

PSyclone will initially create a "vanilla" (functionally correct but not optimised) schedule. This "vanilla" schedule can be modified by changing the objects within it. For example, the order that two Kernel calls appear in the generated code can be changed by changing their order in the tree. The ability to modify this high level view of a schedule allows the PSy layer to be optimised for a particular architecture (by applying optimisations such as blocking, loop merging, inlining, OpenMP parallelisation etc.). The tree could be manipulated directly, however, to simplify optimisation, a set of transformations are supplied. These transformations are discussed in the next section.

#### 5.5 Schedule visualisation

PSyclone supports visualising a schedule in two ways. Firstly the *view()* method outputs textual information about the contents of a schedule. If we were to look at the dynamo eg6 example we would see the following output:

```
>>> schedule.view()
Schedule[invoke='invoke_0' dm=True]
   Directive[OMP parallel do]
      Loop[type='dofs',field_space='any_space_1',it_space='dofs']
      Call setval_X_code(p,z)
      Call X_innerproduct_Y_code(rs_old,res,z)
GlobalSum[scalar='rs_old']
```

The above output tells us that the invoke name for the schedule we are looking at is *invoke\_0* and that the distributed\_memory option has been switched on. Within the schedule is an OpenMP parallel directive containing a loop which itself contains two builtin calls. As the latter of the two builtin calls requires a reduction and distributed memory is switched on, PSyclone has added a GlobalSum call for the appropriate scalar.

5.4. Schedule 25

Secondly, the dag() method (standing for directed acyclic graph), outputs the schedule and its data dependencies. By default a file in dot format is output with the name dag and a file in svg format is output with the name dag.svg. The file name can be changed using the file\_name optional argument and the output file format can be changed using the file\_format optional argument. The file\_format value is simply passed on to graphviz so the graphviz documentation should be consulted for valid formats if svg is not required.

```
>>> schedule.dag(file_name="lovely", file_format="png")
```

Note: The dag method can be called from any node and will output the dag for that node and all of its children.

If we were to look at the dynamo eg6 example we would see the following image:



In the image, all nodes (Psyclone's generic name for objects in the schedule) with children are split into a start vertex and an end vertex (for example the Schedule node has both *schedule\_start* and *schedule\_end* vertices). Blue arrows indicate that there is a parent to child relationship (from a start node) or a child to parent relationship (to an end node). Green arrows indicate that a Node depends on another Node later in the schedule (which we call a forward dependence). Therefore the OMP parallel loop must complete before the globalsum is performed. Red arrows indicate that a Node depends on another Node that is earlier in the schedule (which we call a backward dependence). However the direction of the red arrows are reversed to improve the flow of the dag layout. In this example the forward and backward dependence is the same, however this is not always the case. The two built-ins do not depend on each other, so they have no associated green or red arrows.

The dependence graph output gives an indication of whether nodes can be moved in the schedule. In this case it is valid to run the builtin's in either order. The underlying dependence analysis used to create this graph is used to determine whether a transformation of a schedule is valid from the perspective of data dependencies.

#### **TRANSFORMATIONS**

As discussed in the previous section, transformations can be applied to a schedule to modify it. Typically transformations will be used to optimise the PSy layer for a particular architecture, however transformations could be added for other reasons, such as to aid debugging or for performance monitoring.

## 6.1 Finding

Transformations can be imported directly, but the user needs to know what transformations are available. A helper class **TransInfo** is provided to show the available transformations

```
class psyclone.psyGen.TransInfo(module=None, base_class=None)
```

This class provides information about, and access, to the available transformations in this implementation of PSyclone. New transformations will be picked up automatically as long as they subclass the abstract Transformation class.

For example:

```
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> print(t.list)
There is 1 transformation available:
   1: SwapTrans, A test transformation
>>> # accessing a transformation by index
>>> trans = t.get_trans_num(1)
>>> # accessing a transformation by name
>>> trans = t.get_trans_name("SwapTrans")
```

#### get\_trans\_name (name)

return the transformation with this name (use list() first to see available transformations)

#### get\_trans\_num(number)

return the transformation with this number (use list() first to see available transformations)

#### list

return a string with a human readable list of the available transformations

#### num\_trans

return the number of transformations available

#### 6.2 Available

Most transformations are generic as the schedule structure is independent of the API, however it often makes sense to specialise these for a particular API by adding API-specific errors checks. Some transformations are API-specific (or specific to a set of API's e.g. dynamo). Currently these different types of transformation are indicated by their names.

The generic transformations currently available are listed in alphabetical order below (a number of these have specialisations which can be found in the API-specific sections).

**Note:** PSyclone currently only supports OpenACC transformations for the GOcean 1.0 API. Attempts to apply these transformations to (members of) Schedules from other APIs will be rejected.

#### class psyclone.transformations.ACCDataTrans

Adds an OpenACC "enter data" directive to a Schedule. For example:

```
>>> from psyclone.parse import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> dtrans = t.get_trans_name('ACCDataTrans')
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>>
>>> # Add an enter-data directive
>>> newschedule, _ = dtrans.apply(schedule)
>>> newschedule.view()
```

#### apply (sched)

Adds an OpenACC "enter data" directive to the invoke associated with the supplied Schedule. Any fields accessed by OpenACC kernels within this schedule will be added to this data region in order to ensure they remain on the target device.

Parameters sched (sub-class of psyclone.psyGen.Schedule.) - Schedule to which to add an "enter data" directive.

**Returns** Tuple of the modified schedule and a record of the transformation.

Return type (psyclone.psyGen.Schedule, psyclone.undoredo.Memento)

#### Raises

- NotImplementedError for any API other than GOcean 1.0.
- **TransformationError** if passed something that is not a (subclass of) psyclone. psyGen.Schedule.

#### name

**Returns** the name of this transformation.

Return type str

## class psyclone.transformations.ACCLoopTrans

Adds an OpenACC loop directive to a loop. This directive must be within the scope of some OpenACC Parallel region (at code-generation time).

For example:

```
>>> from psyclone.parse import parse, ParseError
>>> from psyclone.psyGen import PSyFactory, GenerationError
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> ltrans = t.get_trans_name('ACCLoopTrans')
>>> rtrans = t.get_trans_name('ACCParallelTrans')
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> new_schedule = schedule
# Apply the OpenACC Loop transformation to *every* loop
# in the schedule
>>> for child in schedule.children:
>>>
       newschedule, memento = ltrans.apply(child, reprod=True)
>>>
        schedule = newschedule
>>>
# Enclose all of these loops within a single OpenACC
# PARALLEL region
>>> rtrans.omp_schedule("dynamic,1")
>>> newschedule, memento = rtrans.apply(schedule.children)
>>>
```

# apply (node, collapse=None, independent=True)

Apply the ACCLoop transformation to the specified node in a Schedule. This node must be a Loop since this transformation corresponds to inserting a directive immediately before a loop, e.g.:

```
!$ACC LOOP
do ...
...
end do
```

At code-generation time (when psyclone.psyGen.ACCLoopDirective.gen\_code() is called), this node must be within (i.e. a child of) a PARALLEL region.

### **Parameters**

- **node** (psyclone.psyGen.Loop.) the supplied node to which we will apply the Loop transformation.
- collapse (int) number of loops to collapse into single iteration space or None.
- **independent** (bool) whether to add the "independent" clause to the directive (not strictly necessary within PARALLEL regions).

Returns (psyclone.psyGen.Schedule, psyclone.undoredo.Memento)

6.2. Available 31

#### name

Returns the name of this transformation as a string.

#### class psyclone.transformations.ACCParallelTrans

Create an OpenACC parallel region by inserting directives. This parallel region *must* come after an enter-data directive (see *ACCDataTrans*). For example:

```
>>> from psyclone.parse import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> ptrans = t.get_trans_name('ACCParallelTrans')
>>> dtrans = t.get_trans_name('ACCDataTrans')
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> # Enclose everything within a single OpenACC PARALLEL region
>>> newschedule, _ = ptrans.apply(schedule.children)
>>> # Add an enter-data directive
>>> newschedule, _ = dtrans.apply(newschedule)
>>> newschedule.view()
```

## apply (nodes)

Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Loops in the schedule within a single parallel region.

Parameters nodes ((list of) psyclone.psyGen.Node.) - a single Node or a list of Nodes.

**Raises** *TransformationError* – if the nodes argument is not of the correct type.

#### name

**Returns** The name of this transformation as a string.

Return type str

## class psyclone.transformations.ColourTrans

Apply a colouring transformation to a loop (in order to permit a subsequent parallelisation over colours). For example:

```
>>> invoke = ...
>>> schedule = invoke.schedule
>>>
>>> ctrans = ColourTrans()
>>>
>>> # Colour all of the loops
>>> for child in schedule.children:
>>> cschedule, _ = ctrans.apply(child)
>>>
>>> csched.view()
```

### apply (node)

Converts the Loop represented by node into a nested loop where the outer loop is over colours and the inner loop is over cells of that colour. :param node: The loop to transform. :type node: psyclone.psyGen.Loop:returns: Tuple of modified schedule and record of transformation:rtype: (psyclone.psyGen.Schedule, :py:class:`psyclone.undoredo.Memento)

#### name

Returns the name of this transformation as a string.

## class psyclone.transformations.KernelModuleInlineTrans

Switches on, or switches off, the inlining of a Kernel subroutine into the PSy layer module. For example:

```
>>> invoke = ...
>>> schedule = invoke.schedule
>>>
>>> inline_trans = KernelModuleInlineTrans()
>>>
>>> ischedule, _ = inline_trans.apply(schedule.children[0].children[0])
>>> ischedule.view()
```

**Warning:** For this transformation to work correctly, the Kernel subroutine must only use data that is passed in by argument, declared locally or included via use association within the subroutine. Two examples where in-lining will not work correctly are:

- 1. A variable is declared within the module that contains the Kernel subroutine and is then accessed within that Kernel:
- 2. A variable is included via use association at the module level and accessed within the Kernel subroutine

There are currently no checks that these rules are being followed when in-lining so the onus is on the user to ensure correctness.

## apply (node, inline=True)

Checks that the node is of the correct type (a Kernel) then marks the Kernel to be inlined, or not, depending on the value of the inline argument. If the inline argument is not passed the Kernel is marked to be inlined.

## name

Returns the name of this transformation as a string.

## class psyclone.transformations.LoopFuseTrans

Provides a loop-fuse transformation. For example:

6.2. Available 33

```
apply (node1, node2)
```

Fuse the loops represented by node1 and node2.

#### name

Returns the name of this transformation as a string.

#### class psyclone.transformations.MoveTrans

Provides a transformation to move a node in the tree. For example:

Nodes may only be moved to a new location with the same parent and must not break any dependencies otherwise an exception is raised.

```
apply (node, location, position='before')
```

Move the node represented by node before location location (which is also a node) by default and after if the optional *position* argument is set to 'after'. An exception is raised if the move is invalid.

#### name

Returns the name of this transformation as a string.

### class psyclone.transformations.ProfileRegionTrans

Create a profile region around a list of statements. For example:

```
>>> from psyclone.parse import parse, ParseError
>>> from psyclone.psyGen import PSyFactory, GenerationError
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> p_trans = t.get_trans_name('ProfileRegionTrans')
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>>
>>> # Enclose all children within a single profile region
>>> newschedule, _ = p_trans.apply(schedule.children)
>>> newschedule.view()
```

### apply (nodes)

Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Nodes in the schedule within a single profiler region.

**Parameters nodes** (psyclone.psygen.Node or list of psyclone.psygen.Node.) – Can be a single node or a list of nodes.

#### name

Returns the name of this transformation as a string

```
class psyclone.transformations.OMPLoopTrans(omp_schedule='static')
```

Adds an orphaned OpenMP directive to a loop. i.e. the directive must be inside the scope of some other OMP Parallel REGION. This condition is tested at code-generation time. The optional 'reprod' argument in the apply method decides whether standard OpenMP reduction support is to be used (which is not reproducible) or whether a manual reproducible reproduction is to be used.

**Parameters** omp\_schedule (str) - The OpenMP schedule to use.

For example:

```
>>> from psyclone.parse import parse, ParseError
>>> from psyclone.psyGen import PSyFactory, GenerationError
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>> print psy.invokes.names
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> ltrans = t.get_trans_name('OMPLoopTrans')
>>> rtrans = t.get_trans_name('OMPParallelTrans')
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> new_schedule = schedule
>>>
# Apply the OpenMP Loop transformation to *every* loop
# in the schedule
>>> for child in schedule.children:
>>>
       newschedule, memento = ltrans.apply(child, reprod=True)
       schedule = newschedule
>>>
>>>
# Enclose all of these loops within a single OpenMP
# PARALLEL region
>>> rtrans.omp_schedule("dynamic,1")
>>> newschedule, memento = rtrans.apply(schedule.children)
>>>
>>>
```

## apply (node, reprod=None)

Apply the OMPLoopTrans transformation to the specified node in a Schedule. This node must be a Loop since this transformation corresponds to wrapping the generated code with directives like so:

```
! $OMP DO
do ...
...
end do
! $OMP END DO
```

At code-generation time (when OMPLoopDirective.gen\_code() is called), this node must be within (i.e. a child of) an OpenMP PARALLEL region.

6.2. Available 35

The optional reprod argument will cause a reproducible reduction to be generated if it is set to True, otherwise the default value (as read from the psyclone.cfg file) will be used. Note, reproducible in this case means obtaining the same results with the same number of OpenMP threads, not for different numbers of OpenMP threads.

## **Parameters**

- node (psyclone.psyGen.Node) the supplied node to which we will apply the OMPLoopTrans transformation
- reprod (Boolean or None) optional argument to determine whether to generate reproducible OpenMP reductions (True) or not (False). The default value is None which will cause PSyclone to look up a default value

Returns (psyclone.psyGen.Schedule, psyclone.undoredo.Memento)

### name

Returns the name of this transformation as a string.

## omp\_schedule

Returns the OpenMP schedule that will be specified by this transformation. The default schedule is 'static'.

class psyclone.transformations.OMPParallelLoopTrans(omp\_schedule='static')
 Adds an OpenMP PARALLEL DO directive to a loop.

For example:

```
>>> from psyclone.parse import parse
>>> from psyclone.psyGen import PSyFactory
>>> ast, invokeInfo = parse("dynamo.F90")
>>> psy = PSyFactory("dynamo0.1").create(invokeInfo)
>>> schedule = psy.invokes.get('invoke_v3_kernel_type').schedule
>>> schedule.view()
>>>
>>> from psyclone.transformations import OMPParallelLoopTrans
>>> trans = OMPParallelLoopTrans()
>>> new_schedule, memento = trans.apply(schedule.children[0])
>>> new_schedule.view()
```

## apply (node)

Apply an OMPParallelLoop Transformation to the supplied node (which must be a Loop). In the generated code this corresponds to wrapping the Loop with directives:

```
!$OMP PARALLEL DO ...
do ...
end do
!$OMP END PARALLEL DO
```

**Parameters node** (psyclone.f2pygen.DoGen) – the node (loop) to which to apply the transformation.

**Returns** Two-tuple of transformed schedule and a record of the transformation.

```
Return type (psyclone.psyGen.Schedule, :py:class:`psyclone.undoredo.Memento)
```

### name

Returns the name of this transformation as a string.

class psyclone.transformations.OMPParallelTrans

Create an OpenMP PARALLEL region by inserting directives. For example:

```
>>> from psyclone.parse import parse, ParseError
>>> from psyclone.psyGen import PSyFactory, GenerationError
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> ltrans = t.get_trans_name('GOceanOMPLoopTrans')
>>> rtrans = t.get_trans_name('OMPParallelTrans')
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> new_schedule = schedule
>>>
>>> # Apply the OpenMP Loop transformation to *every* loop
>>> # in the schedule
>>> for child in schedule.children:
       newschedule, memento = ltrans.apply(child)
>>>
       schedule = newschedule
>>>
>>> # Enclose all of these loops within a single OpenMP
>>> # PARALLEL region
>>> newschedule, _ = rtrans.apply(schedule.children)
>>> newschedule.view()
```

name

**Returns** The name of this transformation as a string.

Return type str

**Note:** PSyclone does not support (distributed-memory) halo swaps or global sums within OpenMP parallel regions. Attempting to create a parallel region for a set of nodes that includes halo swaps or global sums will produce an error. In such cases it may be possible to re-order the nodes in the Schedule such that the halo swaps or global sums are performed outside the parallel region. The *MoveTrans* transformation may be used for this.

# 6.3 Kernels

PSyclone supports the transformation of Kernels as well as PSy-layer code. Currently just one transformation is provided for this purpose:

class psyclone.transformations.ACCRoutineTrans

Transform a kernel subroutine by adding a "!\$acc routine" directive (causing it to be compiled for the OpenACC accelerator device). For example:

```
>>> from psyclone.parse import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean1.0"
```

6.3. Kernels 37

```
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api)
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.transformations import ACCRoutineTrans
>>> rtrans = ACCRoutineTrans()
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> kern = schedule.children[0].children[0].children[0]
>>> # Transform the kernel
>>> newkern, _ = rtrans.apply(kern)
```

## apply (kern)

Modifies the AST of the supplied kernel so that it contains an '!\$acc routine' OpenACC directive.

Parameters kern (psyclone.psyGen.Call) - The kernel object to transform.

**Returns** (transformed kernel, memento of transformation)

Return type 2-tuple of (psyclone.psyGen.Kern, psyclone.undoredo.Memento).

**Raises** *TransformationError* – if we fail to find the subroutine corresponding to the kernel object.

#### name

**Returns** the name of this transformation class.

Return type str

# 6.4 Applying

Transformations can be applied either interactively or through a script.

# 6.4.1 Interactive

To apply a transformation interactively we first parse and analyse the code. This allows us to generate a "vanilla" PSy layer. For example . . .

```
from parse import parse
from psyGen import PSyFactory

# This example uses version 0.1 of the Dynamo API
api = "dynamo0.1"

# Parse the file containing the algorithm specification and
# return the Abstract Syntax Tree and invokeInfo objects
ast, invokeInfo = parse("dynamo.F90", api=api)

# Create the PSy-layer object using the invokeInfo
psy = PSyFactory(api).create(invokeInfo)

# Optionally generate the vanilla PSy layer fortran
print psy.gen
```

We then extract the particular schedule we are interested in. For example ...

```
# List the various invokes that the PSy layer contains
print psy.invokes.names

# Get the required invoke
invoke = psy.invokes.get('invoke_0_v3_kernel_type')

# Get the schedule associated with the required invoke
schedule = invoke.schedule
schedule.view()
```

Now we have the schedule we can create and apply a transformation to it to create a new schedule and then replace the original schedule with the new one. For example ...

```
# Get the list of possible loop transformations
from psyGen import TransInfo
t = TransInfo()
print t.list

# Create an OpenMPLoop-transformation
ol = t.get_trans_name('OMPParallelLoopTrans')

# Apply it to the loop schedule of the selected invoke
new_schedule, memento = ol.apply(schedule.children[0])
new_schedule.view()

# Replace the original loop schedule of the selected invoke
# with the new, transformed schedule
invoke.schedule=new_schedule

# Generate the Fortran code for the new PSy layer
print psy.gen
```

More examples of use of the interactive application of transformations can be found in the runme\*.py files within the examples/dynamo/eg1 and examples/dynamo/eg2 directories. Some simple examples of the use of transformations are also given in the previous section.

# 6.4.2 Script

PSyclone provides a Python script (**psyclone**) that can be used from the command line to generate PSy layer code and to modify algorithm layer code appropriately. By default this script will generate "vanilla" (unoptimised) PSy layer code. For example:

```
> psyclone algspec.f90
> psyclone -oalg alg.f90 -opsy psy.f90 -api dynamo0.3 algspec.f90
```

The **psyclone** script has an optional **-s** flag which allows the user to specify a script file to modify the PSy layer as required. Script files may be specified without a path. For example:

```
> psyclone -s opt.py algspec.f90
```

In this case the Python search path **PYTHONPATH** will be used to try to find the script file.

Alternatively, script files may be specified with a path. In this case the file is expected to be found in the specified location. For example . . .

6.4. Applying 39

```
> psyclone -s ./opt.py algspec.f90
> psyclone -s ../scripts/opt.py algspec.f90
> psyclone -s /home/me/PSyclone/scripts/opt.py algspec.f90
```

PSyclone also provides the same functionality via a function (which is what the **psyclone** script calls internally).

Takes a GungHo algorithm specification as input and outputs the associated generated algorithm and psy codes suitable for compiling with the specified kernel(s) and GungHo infrastructure. Uses the parse.parse() function to parse the algorithm specification, the psyGen.PSy class to generate the PSy code and the algGen.Alg class to generate the modified algorithm code.

#### **Parameters**

- **filename** (str) The file containing the algorithm specification.
- **kernel\_path** (*str*) The directory from which to recursively search for the files containing the kernel source (if different from the location of the algorithm specification).
- **script\_name** (*str*) A script file that can apply optimisations to the PSy layer (can be a path to a file or a filename that relies on the PYTHONPATH to find the module).
- line\_length (bool) A logical flag specifying whether we care about line lengths being longer than 132 characters. If so, the input (algorithm and kernel) code is checked to make sure that it conforms. The default is False.
- **distributed\_memory** (bool) A logical flag specifying whether to generate distributed memory code. The default is set in the config.py file.

**Returns** 2-tuple containing fparser1 ASTs for the algorithm code and the psy code.

Raises IOError – if the filename or search path do not exist

For example:

```
>>> from psyclone.generator import generate
>>> alg, psy = generate("algspec.f90")
>>> alg, psy = generate("algspec.f90", kernel_path="src/kernels")
>>> alg, psy = generate("algspec.f90", script_name="optimise.py")
>>> alg, psy = generate("algspec.f90", line_length=True)
>>> alg, psy = generate("algspec.f90", distributed_memory=False)
```

A valid script file must contain a **trans** function which accepts a **PSy** object as an argument and returns a **PSy** object, i.e.:

```
def trans(psy)
    ...
    return psy
```

It is up to the script what it does with the PSy object. The example below does the same thing as the example in the *Interactive* section.

```
def trans(psy):
    from transformations import OMPParallelLoopTrans
    invoke = psy.invokes.get('invoke_0_v3_kernel_type')
    schedule = invoke.schedule
    ol = OMPParallelLoopTrans()
```

```
new_schedule, _ = ol.apply(schedule.children[0])
invoke.schedule = new_schedule
return psy
```

Of course the script may apply as many transformations as is required for a particular schedule and may apply transformations to all the schedules (i.e. invokes) contained within the PSy layer.

Examples of the use of transformation scripts can be found in the examples/dynamo/eg3 and examples/dynamo/scripts directories. Please read the examples/dynamo/README file first as it explains how to run the examples (and see also the examples/check\_examples script).

# 6.5 OpenMP

OpenMP is added to a code by using transformations. The three transformations currently supported allow the addition of an **OpenMP Parallel** directive, an **OpenMP Do** directive and an **OpenMP Parallel Do** directive, respectively, to a code.

The generic versions of these three transformations (i.e. ones that theoretically work for all API's) were given in the *Available* section. The API-specific versions of these transformations are described in the API-specific sections of this document.

## 6.5.1 Reductions

PSyclone supports parallel scalar reductions. If a scalar reduction is specified in the Kernel metadata (see the API-specific sections for details) then PSyclone ensures the appropriate reduction is performed.

In the case of distributed memory, PSyclone will add **GlobalSum's** at the appropriate locations. As can be inferred by the name, only "summation" reductions are currently supported for distributed memory.

In the case of an OpenMP parallel loop the standard reduction support will be used by default. For example

```
!$omp parallel do, reduction(+:x)
!loop
!$omp end parallel do
```

OpenMP reductions do not guarantee to give bit reproducible results for different runs of the same problem even if the same problem is run using the same resources. The reason for this is that the order in which data is reduced is not mandated.

Therefore, an additional **reprod** option has been added to the **OpenMP Do** transformation. If the reprod option is set to "True" then the OpenMP reduction support is replaced with local per-thread reductions which are reduced serially after the loop has finished. This implementation guarantees to give bit-wise reproducible results for different runs of the same problem using the same resources, but will not bit-wise compare if the code is rerun with different numbers of OpenMP threads.

## 6.5.2 Restrictions

If two reductions are used within an OpenMP region and the same variable is used for both reductions then PSyclone will raise an exception. In this case the solution is to use a different variable for each reduction.

PSyclone does not support (distributed-memory) halo swaps or global sums within OpenMP parallel regions. Attempting to create a parallel region for a set of nodes that includes halo swaps or global sums will produce an error. In such cases it may be possible to re-order the nodes in the Schedule using the *MoveTrans* transformation.

6.5. OpenMP 41

**CHAPTER** 

SEVEN

# **DISTRIBUTED MEMORY**

PSyclone supports the generation of code for distributed memory machines. When this option is switched on, PSyclone takes on responsibility for both performance and correctness, as described below.

# 7.1 Correctness

PSyclone is responsible for adding appropriate distributed memory communication calls to the PSy layer to ensure that the distributed memory code runs correctly. For example, a stencil operation will require halo exchanges between the different processes.

The burden of correctly placing distributed memory communication calls has traditionally been born by the user. However, PSyclone is able to determine the placing of these within the PSy-layer, thereby freeing the user from this responsibility. Thus, the Algorithm and Kernel code remain the same, irrespective of whether the target architecture does or does not require a distributed memory solution.

# 7.2 Performance

PSyclone adds **HaloExchange** and **GlobalSum** objects to the generated PSy **Schedule** (see the *Schedule* Section) at the required locations. The halo-exchange and global-sum objects are exposed here for the purposes of optimisation. For example the halo-exchange and/or global-sum objects may be moved in the schedule (via appropriate transformations) to enable overlap of computation with communication.

**Note:** When these optimisations are implemented, add a reference to the *Transformations* Section.

A halo exchange is required with distributed memory when a processor requires data from its halo and the halo information is out of date. One example is where a field is written to and then read using a stencil access. Halo exchanges have performance implications so should only be used where necessary.

A global sum is required with distributed memory when a scalar is written to. Global sums can have performance implications so should only be used where necessary. Global sums currently only occur in certain Built-in kernels. The description of Built-ins indicates when this is the case.

# 7.3 Implementation

Within the contents of an invoke () call, PSyclone is able to statically determine which communication calls are required and where they should be placed. However, PSyclone has no information on what happens outside invoke () calls and thus is unable to statically determine whether communication is required between these calls. The solution

we use is to add run-time flags in the PSy layer to keep track of whether data has been written to and read from. These flags are then used to determine whether communication calls are required upon entry to an invoke().

# 7.4 Control

Support for distributed memory can be switched on or off with the default being on. The default can be changed permanently by modifying the DISTRIBUTED\_MEMORY variable in the psyclone.cfg configuration file to false (see *Configuration*).

Distributed memory can be switched on or off from the psyclone script using the -dm/--dist\_mem or -nodm/--no\_dist\_mem flags, respectively.

For interactive access, the distributed memory option can be changed interactively from the PSyFactory class by setting the optional distributed\_memory flag; for example:

```
psy = PSyFactory(api=api, distributed_memory=False)
```

Similarly the distributed memory option can be changed interactively from the generate function by setting the optional distributed\_memory flag, e.g.:

```
psy, alg = generate("file.f90", distributed_memory=False).
```

# 7.5 Status

Distributed memory support is currently limited to the dynamo0.3 API. The remaining API's ignore the distributed memory flag and continue to produce code without any distributed memory functionality, irrespective of its value.

**CHAPTER** 

# **EIGHT**

# **BUILT-INS**

Built-ins (named by analogy with the native functionality provided by Python) are operations which can be specified within an invoke call in the algorithm layer but do not require an associated kernel to be implemented as they are provided directly by the infrastructure.

One use of Built-ins is for commonly used operations. In this case Built-ins simplify the use of the system as users do not need to write kernel routines. Built-ins also offer a potential performance advantage as they provide a specification of what is required without an implementation. Therefore the PSy layer is free to implement these operations in whatever way it chooses.

**Note:** In general, PSyclone will need to know the types of the arguments being passed to any Built-ins. The parser obtains this information from an API-specific file that contains the metadata for all Built-in operations supported for that API.

# 8.1 Example

In the following example, the invoke call includes a call to a Built-in (setval\_c) and a user-supplied kernel (matrix\_vector\_kernel\_mm\_type). The Built-in sets all values in the field Ax to 0.0. Notice that, unlike the kernel call, no use association is required for the Built-in since it is provided as part of the environment (c.f. Fortran intrinsics such as sin()).

```
subroutine jacobi_solver_algorithm(lhs, rhs, mm, mesh, n_iter)
 use matrix_vector_mm_mod, only: matrix_vector_kernel_mm_type
 integer,
                      intent(in)
                                    :: n_iter
 type(field_type), intent(inout) :: lhs, rhs
 type(operator_type), intent(inout) :: mm
                   intent(in)
 type (mesh_type) ,
                                  :: mesh
                                    :: Ax, lumped_weight, res
 type(field_type)
 real(kind=r_def), parameter :: MU = 0.9_r_def
 do iter = 1,n_iter
   call invoke( setval_c(Ax, 0.0) )
    call invoke( matrix_vector_kernel_mm_type(Ax,lhs,mm) )
 end do
end subroutine jacobi_solver_algorithm
```

Below is an example of a kernel that is consistent with the matrix\_vector\_kernel\_mm\_type kernel specified in the example above.

```
module matrix_vector_mm_mod
 type, public, extends(kernel_type) :: matrix_vector_kernel_mm_type
   private
   type(arg_type) :: meta_args(3) = (/
        arg_type(GH_FIELD, GH_INC, ANY_SPACE_1),
                             GH_READ, ANY_SPACE_1),
        arg_type(GH_FIELD,
        arg_type(GH_OPERATOR, GH_READ, ANY_SPACE_1, ANY_SPACE_1)
   integer :: iterates_over = CELLS
 contains
   procedure, nopass ::matrix_vector_mm_code
  end type
contains
 subroutine matrix_vector_mm_code(cell,
                                   nlayers,
                                   lhs, x,
                                   ncell_3d,
                                   mass_matrix, &
                                   ndf, undf, map)
 end subroutine matrix_vector_mm_code
end module matrix_vector_mm_mod
```

We now translate the algorithm layer code and generate the psy layer code. The algorithm code is assumed to be in a file call *solver\_mod.x90*. In this case we use the top level python interface. See the *API* section for different ways to translate/generate code.

```
>>> from psyclone.generator import generate
>>> alg, psy = generate("solver_mod.x90")
>>> print alg
>>> print psy
```

The resultant generated algorithm code is given below.

Ignoring the difference in case (which is due to the output format of the code parser) the differences between the original algorithm code and the translated algorithm code are:

- the generic calls to invoke have been replaced by specific CALL invoke\_xx. The calls within the invoke are removed, as are duplicate arguments and any literals leaving the three fields being passed in.
- a use statement is added for the each of the new CALL invoke\_xx which will call the generated PSy layer code.

The existance of a call to a Built-in has made no difference at this point:

```
SUBROUTINE jacobi_solver_algorithm(lhs, rhs, mm, mesh, n_iter)
USE solver_mod_psy, ONLY: invoke_5_matrix_vector_kernel_mm_type
USE solver_mod_psy, ONLY: invoke_4
INTEGER, intent(in) :: n_iter
TYPE(field_type), intent(inout) :: lhs, rhs
TYPE(operator_type), intent(inout) :: mm
TYPE(mesh_type), intent(in) :: mesh
TYPE(field_type) ax, lumped_weight, res

REAL(KIND=r_def), parameter :: mu = 0.9_r_def
INTEGER iter
```

```
INTEGER rhs_fs
TYPE(function_space_type) fs
...
DO iter = 1,n_iter
    CALL invoke_4(ax)
    CALL invoke_5_matrix_vector_kernel_mm_type(ax, lhs, mm)
    ...
END DO
END SUBROUTINE jacobi_solver_algorithm
```

A vanilla (not optimised) version of the generated PSy layer is given below. As expected the kernel code is called from the PSy layer. However, in the case of the  $setval\_c$  Built-in, the code for this has been written directly into the PSy layer (the loop setting  $ax\_proxy\%data(df) = 0.0$ ). This example illustrates that Built-ins may be implemented in whatever way the generator sees fit with no change to the algorithm and kernel layers.

```
MODULE solver_mod_psy
 SUBROUTINE invoke_4 (ax)
   USE mesh_mod, ONLY: mesh_type
   TYPE(field_type), intent(inout) :: ax
   INTEGER df
   INTEGER undf_any_space_1
   TYPE (field_proxy_type) ax_proxy
   ! Initialise field proxies
   ax_proxy = ax%get_proxy()
    ! Initialise sizes and allocate any basis arrays for any_space_1
   undf_any_space_1 = ax_proxy%vspace%get_undf()
    ! Call our kernels
   DO df=1, undf_any_space_1
     ax_proxy%data(df) = 0.0
   END DO
    !
 END SUBROUTINE invoke_4
 SUBROUTINE invoke_5_matrix_vector_kernel_mm_type(ax, lhs, mm)
   USE matrix_vector_mm_mod, ONLY: matrix_vector_mm_code
   TYPE (field_type), intent (inout) :: ax, lhs
   TYPE(operator_type), intent(inout) :: mm
   ! Initialise field proxies
   ax_proxy = ax%get_proxy()
   lhs_proxy = lhs%get_proxy()
   mm_proxy = mm%get_proxy()
    ! Initialise number of layers
   nlayers = ax_proxy%vspace%get_nlayers()
```

8.1. Example 47

This example is distributed with PSyclone and can be found in <PSYCLONEHOME>/examples/dynamo/eg4.

# 8.2 Supported Built-in operations

The list of supported Built-ins is API-specific and therefore is described under the documentation of each API.

# 8.3 Adding new additional Built-in operations

- 1. Identify the PSyclone source file for the API to be extended. *e.g.* for dynamo0.3 it is src/dynamo0p3\_builtins.py.
- 2. Edit this source file to create the class for this new call. It must inherit from the API-specific parent class for Built-in operations (DynBuiltInKern for dynamo0.3).
- 3. Implement \_\_str\_\_ and gen\_code () methods for this new class.
- 4. Add the name of the new Built-in operation and its corresponding class to the BUILTIN\_MAP dictionary in that source file.
- 5. Add metadata describing this call to the appropriate file specified in the BUILTIN\_DEFINITIONS\_FILE in that source file. For dynamo0.3 this is dynamo0p3\_builtins\_mod.f90.
- 6. Add relevant tests to the PSyclone test file for the API to be extended. *e.g.* for dynamo0.3 it is src/tests/dynamo0p3\_builtins\_test.py. The tests rely on single\_invoke Fortran examples in the relevant src/tests/test\_files/ subfolder.
- 7. Add an appropriate Fortran single\_invoke example for the new Built-in in the relevant src/tests/test\_files/ subfolder. e.g. for dynamo0.3 it is src/tests/test\_files/dynamo0p3/. Names of examples follow the template <category.number>.<subcategory.number>.<subcategory.number>.is 15.

8. Document the new Built-in in the documentation of the relevant API (e.g. doc/dynamo0p3.rst).

If the API being extended does not currently support any Built-ins then the BUILTIN\_MAP and BUILTIN\_DEFINITIONS\_FILE module variables must be added to the source file for the API. A Fortran module file must be created in the PSyclone src directory (with the name specified in BUILTIN\_DEFINITIONS\_FILE) containing metadata describing the Built-in operations. Finally, parse.get\_builtin\_defs() must be extended to import BUILTIN MAP and BUILTIN DEFINITIONS FILE for this API.

50 Chapter 8. Built-ins

**CHAPTER** 

NINE

# DYNAMO0.3 API

This section describes the Dynamo0.3 application programming interface (API). This API explains what a user needs to write in order to make use of the Dynamo0.3 API in PSyclone.

As with the majority of PSyclone API's, the Dynamo0.3 API specifies how a user needs to write the algorithm layer and the kernel layer to allow PSyclone to generate the PSy layer. These algorithm and kernel API's are discussed separately in the following sections.

The Dynamo0.3 API supports the Met Office's finite element (hereafter FEM) based GungHo dynamical core (see *Introduction*). The Met Office Collaboration Wiki (login required) provides more detailed information about the dynamical core's formulation and data model.

# 9.1 Algorithm

The general requirements for the structure of an Algorithm are explained in the *Algorithm layer* section. This section explains the Dynamo0.3-specific specialisations and extensions.

# 9.1.1 Example

An example Dynamo0.3 API invoke call is given below with various different types of objects supported by the API. These different objects and their use are discussed in the following sections.

```
real(kind=r_def)
                               :: scalar1
integer(kind=i_def)
                               :: stencil_extent
type(field_type)
                               :: field1, field2, field3
                               :: field5(3), field6(3)
type(field_type)
type (quadrature_type)
type(operator_type)
                               :: operator1
type(columnwise_operator_type) :: cma_op1
call invoke( kernel1(field1, field2, operator1, qr),
             builtin1(scalar1, field2, field3),
             kernel2(field1, stencil_extent, field3, scalar1), &
             assembly_kernel(cma_op1, operator1),
             name="some_calculation"
                                                                 δε
call invoke( prolong_kernel_type(field1, field4),
                                                                 S.
             restrict_kernel_type(field5, field6)
           )
```

Please see the Algorithm layer section for a description of the name argument.

Objects in the Dynamo0.3 API can be categorised by their functionality as data types and information that specifies supported operations on a particular data type. The above example introduces four of five data types supported by the Dynamo0.3 API: field, scalar, operator and column-wise operator (field vector is the fifth). qr represents a quadrature object which provides information required by a kernel to operate on fields (see section *Quadrature* for more details).

## 9.1.2 Field

Dynamo0.3 API fields, identified with GH\_FIELD metadata, represent FEM discretisations of various dynamical core prognostic and diagnostic variables. In FEM, variables are discretised by placing them into a function space (see *Supported Function Spaces*) from which they inherit a polynomial expansion via the basis functions of that space. Field values at points within a cell are evaluated as the sum of a set of basis functions multiplied by coefficients which are the data points. Points of evaluation are determined by a quadrature object (*Quadrature*) and are independent of the function space the field is on. Placement of field data points, also called degrees of freedom ("dof"), is determined by the function space the field is on.

# 9.1.3 Field Vector

Depending on the function space a field lives on, the field data at a point can be a scalar or a vector (see *Supported Function Spaces* for the list of scalar and vector function spaces). There is an additional option which specifies whether the data itself is vector valued and it usually refers to bundles of scalar valued fields. Field vectors are represented as GH\_FIELD\*N where N is the size of the vector. The 3D coordinate field, for example, has (x, y, z) scalar values at the nodes and therefore has a vector size of 3.

## 9.1.4 Scalar

In Dynamo 0.3 API a scalar is a single value variable that can be either real or integer. Real scalars are identified with GH REAL and integer scalars are identified with GH INTEGER metadata.

# 9.1.5 Operator

Represents a matrix constructed on a per-cell basis using Local Matrix Assembly (LMA) and is identified with GH OPERATOR metadata.

# 9.1.6 Column-Wise Operator

The Dynamo 0.3 API has support for the construction and use of column-wise/Column Matrix Assembly (CMA) operators whose metadata identifier is GH\_COLUMNWISE\_OPERATOR. As the name suggests, these are operators constructed for a whole column of the mesh. These are themselves constructed from the Local Matrix Assembly (LMA) operators of each cell in the column. The rules governing Kernels that have CMA operators as arguments are given in the *Kernel* section below.

There are three recognised Kernel types involving CMA operations; construction, application (including inverse application) and matrix-matrix. The following example sketches-out what the use of such kernels might look like in the Algorithm layer:

```
use field_mod, only: field_type
use operator_mod, only : operator_type, columnwise_operator_type
type(field_type) :: field1, field2, field3
type(operator_type) :: lma_op1, lma_op2
type(columnwise_operator_type) :: cma_op1, cma_op2, cma_op3
real(kind=r_def) :: alpha
```

```
call invoke(

assembly_kernel(cma_op1, lma_op1, lma_op2),

assembly_kernel2(cma_op2, lma_op1, lma_op2, field3),

apply_kernel(field1, field2, cma_op1),

matrix_matrix_kernel(cma_op3, cma_op1, alpha, cma_op2), &

apply_kernel(field3, field1, cma_op3),

name="cma_example")
```

The above invoke uses two LMA operators to construct the CMA operator cma\_op1. A second CMA operator, cma\_op2, is assembled from the same two LMA operators but also uses a field. The first of these CMA operators is then applied to field2 and the result stored in field1 (assuming that the metadata for apply\_kernel specifies that it is the first field argument that is written to). The two CMA operators are then combined to produce a third, cma\_op3. This is then applied to field1 and the result stored in field3.

Note that PSyclone identifies the type of kernels performing Column-Wise operations based on their arguments as described in metadata (see *Rules for Kernels that work with CMA Operators* below). The names of the kernels in the above example are purely illustrative and are not used by PSyclone when determining kernel type.

A full example of CMA operator construction is available in examples/dynamo/eg7.

# 9.1.7 Quadrature

Kernels conforming to the Dynamo 0.3 API may require quadrature information (specified using e.g.  $gh\_shape = gh\_quadrature\_XYoZ$  in the kernel metadata - see Section  $gh\_shape$  and  $gh\_evaluator\_targets$ ). This information must be passed to the kernel from the Algorithm layer in the form of a  $quadrature\_type$  object. This must be the last argument passed to the kernel, e.g.:

This quadrature object specifies the set of points at which the basis/differential-basis functions required by the kernel are to be evaluated.

# 9.1.8 Stencils

Kernel metadata may specify that a Kernel performs a stencil operation on a field. Any such metadata must provide a stencil type. See the *meta\_args* section for more details. The supported stencil types are X1D, Y1D, XORY1D or CROSS.

If a stencil operation is specified by the Kernel metadata the algorithm layer must provide the extent of the stencil (the maximum distance from the central cell that the stencil extends). The Dynamo0.3 API expects this information to be added as an additional integer argument immediately after the relevant field when specifying the Kernel via an invoke.

For example:

```
integer :: extent = 2
call invoke(kernel(field1, field2, extent))
```

where field2 has kernel metadata specifying that it has a stencil access.

9.1. Algorithm 53

extent may also be passed as a literal. For example:

```
call invoke(kernel(field1, field2, 2))
```

where, again, field2 has kernel metadata specifying that it has a stencil access.

**Note:** The stencil extent specified in the Algorithm layer is not the same as the stencil size passed in to the Kernel. The latter contains the number of cells in the stencil which is dependent on both the stencil type and extent.

If the Kernel metadata specifies that the stencil is of type XORY1D (which means X1D or Y1D) then the algorithm layer must specify whether the stencil is X1D or Y1D for that particular kernel call. The Dynamo0.3 API expects this information to be added as an additional argument immediately after the relevant stencil extent argument. The argument should be an integer with valid values being x\_direction or y\_direction, both being supplied by the LFRic infrastructure via the flux\_direction\_mod fortran module

For example:

```
use flux_direction_mod, only : x_direction
integer :: direction = x_direction
integer :: extent = 2
! ...
call invoke(kernel(field1, field2, extent, direction))
```

direction may also be passed as a literal. For example:

```
use flux_direction_mod, only : x_direction
integer :: extent = 2
! ...
call invoke(kernel(field1, field2, extent, x_direction))
```

If certain fields use the same value of extent and/or direction then the same variable, or literal value can be provided.

For example:

```
call invoke(kernel1(field1, field2, extent, field3, extent, direction), & kernel2(field1, field2, extent2, field4, extent, direction))
```

In the above example field2 and field3 in kernel1 and field4 in kernel2 will have the same extent value but field2 in kernel2 may have a different value. Similarly, field3 in kernel1 and field4 in kernel2 will have the same direction value.

An example of the use of stencils is available in examples/dynamo/eg5.

There is currently no attempt to perform type checking in PSyclone so any errors in the type and/or position of arguments will not be picked up until compile time. However, PSyclone does check for the correct number of algorithm arguments. If the wrong number of arguments is provided then an exception is raised.

For example, running test 19.2 from the Dynamo0.3 API test suite gives:

```
cd <PSYCLONEHOME>/src/psyclone/tests
psyclone test_files/dynamo0p3/19.2_single_stencil_broken.f90
"Generation Error: error: expected '5' arguments in the algorithm layer but found '4'.
Expected '4' standard arguments, '1' stencil arguments and '0' qr_arguments'"
```

# 9.1.9 Inter-grid

From the Algorithm layer, an Invoke for inter-grid kernels (those that map fields between grids of different resolution) looks much like an Invoke containing general-purpose kernels. The only restrictions to be aware of are that inter-grid kernels accept only field or field-vectors as arguments and that an Invoke may not mix inter-grid kernels with any other kernel type. (Hence the second, separate Invoke in the example Algorithm code given at the beginning of this Section.)

# 9.2 PSy-layer

The general details of the PSy-layer are explained in the *PSy layer* section. This section describes any dynamo0p3 specific issues.

## 9.2.1 Module name

The PSy-layer code is contained within a Fortran module. The name of the module is determined from the algorithm-layer name with "\_psy" appended. The algorithm-layer name is the algorithm's module name if it is a module, its subroutine name if it is a subroutine that is not within a module, or the program name if it is a program.

So, for example, if the algorithm code is contained within a module called "fred" then the PSy-layer module name will be "fred\_psy".

# 9.3 Kernel

The general requirements for the structure of a Kernel are explained in the *Kernel layer* section. In the Dynamo API there are four different Kernel types; general purpose, CMA, inter-grid and *Built-ins*. In the case of built-ins, PSyclone generates the source of the kernels. This section explains the rules for the other three, user-supplied kernel types and then goes on to describe their metadata and subroutine arguments.

# 9.3.1 Rules for all User-Supplied Kernels

In the following, 'operator' refers to both LMA and CMA operator types.

- 1. A Kernel must have at least one argument that is a field, field vector, or operator. This rule reflects the fact that a Kernel iterates over a space and therefore must have some representation over that space.
- 2. The continuity of the iteration space of the Kernel is determined from the function space of the modified argument (see *Supported Function Spaces*). If more than one argument is modified then the iteration space is taken to be the largest required by any of those arguments. e.g. if a Kernel writes to two fields, the first on W3 (discontinuous) and the second on W1 (continuous), then the iteration space of that Kernel will be determined by the field on the continuous space.
- 3. If the function space of the modified argument(s) cannot be determined then they are assumed to be continuous. This is the case if any of the modified arguments are declared as ANY\_SPACE and their actual space cannot be determined statically. This assumption is always safe but leads to additional computation if the quantities being updated are actually on discontinuous function spaces.
- 4. Operators do not have halo operations operating on them as they are either cell- (LMA) or column-based (CMA) and therefore act like discontinuous fields.
- 5. Any Kernel that writes to an operator will have its iteration space expanded such that valid values for the operator are computed in the level-1 halo.

9.2. PSy-layer 55

6. Any Kernel that reads from an operator must not access halos beyond level 1. In this case PSyclone will check that the Kernel does not require values beyond the level-1 halo. If it does then PSyclone will abort.

# 9.3.2 Rules specific to General-Purpose Kernels without CMA Operators

- 1. General-purpose kernels accept arguments of any of the following types: field, field vector, LMA operator, scalar integer, scalar real.
- 2. A Kernel is permitted to write to more than one quantity (field or operator) and these quantities may be on the same or different function spaces.
- 3. A Kernel may not write to a scalar argument. (Only *Built-ins* are permitted to do this.) Any scalar aguments must therefore be declared in the metadata as GH\_READ see below.

# 9.3.3 Rules for Kernels that work with CMA Operators

The Dynamo 0.3 API has support for kernels that assemble, apply (or inverse-apply) column-wise/Column Matrix Assembly (CMA) operators. Such operators may also be used by matrix-matrix kernels. There are thus three types of CMA-related kernels. Since, by definition, CMA operators only act on data within a column, they have no horizontal dependencies. Therefore, kernels that write to them may be parallelised without colouring.

All three CMA-related kernel types must obey the following rules:

- 1. Since a CMA operator only acts within a single column of data, stencil operations are not permitted.
- 2. No vector quantities (e.g. "GH\_FIELD\*3" see below) are permitted as arguments.

There are then additional rules specific to each of the three CMA kernel types. These are described below.

## **Assembly**

CMA operators are themselves constructed from Local-Matrix-Assembly (LMA) operators. Therefore, any kernel which assembles a CMA operator must obey the following rules:

- 1. Have one or more LMA operators as read-only arguments.
- 2. Have exactly one CMA operator argument which must have write access.
- 3. Other types of argument (e.g. scalars or fields) are permitted but must be read-only.

# **Application and Inverse Application**

Column-wise operators can only be applied to fields. CMA-Application kernels must therefore:

- 1. Have a single CMA operator as a read-only argument.
- 2. Have exactly two field arguments, one read-only and one that is written to.
- 3. The function spaces of the read and written fields must match the from and to spaces, respectively, of the supplied CMA operator.

## **Matrix-Matrix**

A kernel that has just column-wise operators as arguments and zero or more read-only scalars is identified as performing a matrix-matrix operation. In this case:

1. Arguments must be CMA operators and, optionally, one or more scalars.

2. Exactly one of the CMA arguments must be written to while all other arguments must be read-only.

## 9.3.4 Rules for Inter-Grid Kernels

- 1. An inter-grid kernel is identified by the presence of a field or field-vector argument with the optional *mesh\_arg* metadata element (see *Inter-Grid Metadata*).
- 2. An invoke that contains one or more inter-grid kernels must not contain any other kernel types. (This restriction is an implementation decision and could be lifted in future if there is a need.)
- 3. An inter-grid kernel is only permitted to have field or field-vector arguments.
- 4. All inter-grid kernel arguments must have the *mesh\_arg* metadata entry.
- 5. An inter-grid kernel (and metadata) must have at least one field on each of the fine and coarse meshes. Specifying all fields as coarse or fine is forbidden.
- 6. Fields on different meshes must always live on different function spaces.
- 7. All fields on a given mesh must be on the same function space.

A consequence of Rules 5-7 is that an inter-grid kernel will only involve two function spaces.

# 9.3.5 Metadata

The code below outlines the elements of the Dynamo0.3 API kernel metadata, 1) 'meta\_args', 2) 'meta\_funcs', 3) 'gh\_shape', 4) 'iterates\_over' and 5) 'procedure'.

```
type, public, extends(kernel_type) :: my_kernel_type
  type(arg_type) :: meta_args(...) = (/ ... /)
  type(func_type) :: meta_funcs(...) = (/ ... /)
  integer :: gh_shape = gh_quadrature_XYoZ
  integer :: iterates_over = cells
contains
  procedure, nopass :: my_kernel_code
end type
```

These five metadata elements are discussed in order in the following sections.

# meta\_args

The meta\_args array specifies information about data that the kernel code expects to be passed to it via its argument list. There is one entry in the meta\_args array for each scalar, field, or operator passed into the Kernel and the order that these occur in the meta\_args array must be the same as they are expected in the kernel code argument list. The entry must be of arg\_type which itself contains metadata about the associated argument. The size of the meta\_args array must correspond to the number of scalars, fields and operators passed into the Kernel.

**Note:** it makes no sense for a Kernel to have only **scalar** arguments (because the PSy layer will call a Kernel for each point in the spatial domain) and PSyclone will reject such Kernels.

For example, if there are a total of 2 scalar / field / operator entities being passed to the Kernel then the meta\_args array will be of size 2 and there will be two arg\_type entries:

9.3. Kernel 57

Argument-metadata (metadata contained within the brackets of an arg\_type entry), describes either a **scalar**, a **field** or an **operator** (either LMA or CMA).

The first argument-metadata entry describes whether the data that is being passed is for a real scalar (GH\_REAL), an integer scalar (GH\_INTEGER), a field (GH\_FIELD) or an operator (either GH\_OPERATOR for LMA or GH\_COLUMNWISE\_OPERATOR for CMA). This information is mandatory.

Additionally, argument-metadata can be used to describe a vector of fields (see the *Field Vector* section for more details).

As an example, the following meta\_args metadata describes 4 entries, the first is a real scalar, the next two are fields and the fourth is an operator. The third entry is a field vector of size 3.

```
type(arg_type) :: meta_args(4) = (/
    arg_type(GH_REAL, ...),
    arg_type(GH_FIELD, ...),
    arg_type(GH_FIELD*3, ...),
    arg_type(GH_OPERATOR, ...)
    /)
```

The second entry to argument-metadata (information contained within the brackets of an arg\_type) describes how the Kernel makes use of the data being passed into it (the way it is accessed within a Kernel). This information is mandatory. There are currently 5 possible values of this metadata GH\_WRITE, GH\_READ, GH\_INC, GH\_READWRITE and GH\_SUM. However, not all combinations of metadata entries are valid and PSyclone will raise an exception if an invalid combination is specified. Valid combinations are specified later in this section (see *Valid Access Modes*).

- GH\_WRITE indicates the data is modified in the Kernel before (optionally) being read.
- GH\_READ indicates that the data is read and is unmodified.
- GH\_INC indicates that different iterations of a Kernel make contributions to shared values. For example, values at cell faces may receive contributions from cells on either side of the face. This means that such a Kernel needs appropriate synchronisation (or colouring) to run in parallel.
- GH\_READWRITE indicates that different iterations of a Kernel update quantitites which do not share dofs, such
  as operators and fields over discontinuous function spaces. If a Kernel modifies only discontinuous fields and/or
  operators there is no need for synchronisation or colouring when running such Kernels in parallel. However,
  modifying another field with a GH\_INC access in a Kernel means that synchronisation or colouring is required
  for parallel runs.
- GH\_SUM is an example of a reduction and is the only reduction currently supported in PSyclone. This metadata indicates that values are summed over calls to Kernel code.

## For example:

```
type(arg_type) :: meta_args(4) = (/
    arg_type(GH_REAL, GH_SUM),
    arg_type(GH_FIELD, GH_INC, ...),
    arg_type(GH_FIELD*3, GH_WRITE, ...),
    arg_type(GH_OPERATOR, GH_READ, ...)
    /)
```

**Note:** In the Dynamo 0.3 API only *Built-ins* are permitted to write to scalar arguments (and hence perform reductions). Furthermore, this permission is currently restricted to real scalars (GH\_REAL) as the LFRic infrastructure does not yet

support integer reductions.

For a scalar the argument metadata contains only these two entries. However, fields and operators require further entries specifying function-space information. The meaning of these further entries differs depending on whether a field or an operator is being described.

In the case of an operator, the 3rd and 4th arguments describe the to and from function spaces respectively. In the case of a field the 3rd argument specifies the function space that the field lives on. More details about the supported function spaces are in subsection *Supported Function Spaces*.

For example, the metadata for a kernel that applies a Column-wise operator to a field might look like:

```
type(arg_type) :: meta_args(3) = (/ &
    arg_type(GH_FIELD, GH_INC, W1), &
    arg_type(GH_FIELD, GH_READ, W2H), &
    arg_type(GH_COLUMNWISE_OPERATOR, GH_READ, W1, W2H) &
    /)
```

In some cases a Kernel may be written so that it works for fields and/or operators from any type of w2 space i.e. one of w2, w2h or w2v. In this case the metadata should be specified as being any\_w2.

**Warning:** in the current implementation it is assumed that all fields and/or operators specifying any\_w2 within a kernel will use the **same** function space. It is up to the user to ensure this is the case as otherwise invalid code would be generated.

It may be that a Kernel is written such that a field and/or operators may be on/map-between any function space(s). In this case the metadata should be specified as being one of any\_space\_1, any\_space\_2, ..., any\_space\_9. The reason for having different names is that a Kernel might be written to allow 2 or more arguments to be able to support any function space but for a particular call the function spaces may have to be the same as each other.

In the example below, the first field entry supports any function space but it must be the same as the operator's to function space. Similarly, the second field entry supports any function space but it must be the same as the operator's from function space. Note, the metadata does not forbid ANY\_SPACE\_1 and ANY\_SPACE\_2 from being the same.

Note also that the scope of this naming of any-space function spaces is restricted to the argument list of individual kernels. i.e. if an Invoke contains say, two kernel calls that each support arguments on any function space, e.g. ANY\_SPACE\_1, there is no requirement that these two function spaces be the same. Put another way, if an Invoke contained two calls of a kernel with arguments described by the above metadata then the first field argument passed to each kernel call need not be on the same space.

**Note:** A GH\_FIELD argument that specifies GH\_WRITE or GH\_READWRITE as its access pattern must be a discontinuous function in the horizontal (see *Valid Access Modes* below). That means it must belong to w3, wtheta or w2v function spaces (see *Supported Function Spaces*). A GH\_FIELD that specifies GH\_INC as its access pattern may be continuous in the vertical (and discontinuous in the horizontal), continuous in the horizontal (and discontinuous in the vertical), or continuous in both. In each case the code is the same. However, if a field is discontinuous in the horizontal then it will not need colouring and, if is described as being on any space, there is currently no way to determine this from the metadata (unless we can statically determine the space of the field being passed in). At the moment this type

9.3. Kernel 59

of Kernel is always treated as if it is continuous in the horizontal, even if it is not.

### Valid Access Modes

As mentioned earlier, not all combinations of metadata are valid. Valid combinations are summarised here. All types of data (GH\_INTEGER, GH\_REAL, GH\_FIELD, GH\_OPERATOR and GH\_COLUMNWISE\_OPERATOR) may be read within a Kernel and this is specified in metadata using GH\_READ. At least one kernel argument must be listed as being modified. When data is *modified* in a Kernel then the permitted access modes depend on the type of data it is and the function space it is on. Valid values are given in the table below.

Argument Type	Function space	Access type
GH_INTEGER	n/a	GH_SUM (Built-ins only)
GH_REAL	n/a	GH_SUM (Built-ins only)
GH_FIELD	Discontinuous	GH_WRITE, GH_READWRITE
GH_FIELD	Continuous	GH_INC
GH_OPERATOR	Any for both 'to' and 'from'	GH_WRITE, GH_READWRITE
GH_COLUMNWISE_OPERATOR	Any for both 'to' and 'from'	GH_WRITE, GH_READWRITE

**Note:** As mentioned above, note that only Built-ins may modify scalar arguments. Since the LFRic infrastructure does not currently support integer reductions, integer scalar arguments are restricted to having read-only access.

There is no restriction on the number and function-spaces of other quantities that a general-purpose kernel can modify other than that it must modify at least one. The rules for kernels involving CMA operators, however, are stricter and only one argument may be modified (the CMA operator itself for assembly, a field for CMA-application and a CMA operator for matrix-matrix kernels). If a kernel writes to quantities on different function spaces then PSyclone generates loop bounds appropriate to the largest iteration space. This means that if a single kernel updates one quantity on a continuous function space and one on a discontinuous space then the resulting loop will include cells in the level 1 halo since they are required for a quantity on a continuous space. As a consequence, any quantities on a discontinuous space will then be computed redundantly in the level 1 halo. Currently PSyclone makes no attempt to take advantage of this (by e.g. setting the appropriate level-1 halo to 'clean').

PSyclone ensures that both CMA and LMA operators are computed (redundantly) out to the level-1 halo cells. This permits their use in kernels which modify quantities on continuous function spaces and also in subsequent redundant computation of other quantities on discontinuous function spaces. In conjunction with this, PSyclone also checks (when generating the PSy layer) that any kernels which read operator values do not do so beyond the level-1 halo. If any such accesses are found then PSyclone aborts.

## **Supported Function Spaces**

As mentioned in the *Field* and *Field Vector* sections, the function space of an argument specifies how it maps onto the underlying topology and, additionally, whether the data at a point is a vector.

Function spaces can share dofs between cells in the horizontal, vertical or both directions. Depending on the function space and FEM order, the shared dofs can lie on one or more cell entities (faces, edges and vertices) in each direction. This property is referred to as the **continuity** of a function space (horizontal, vertical or full). Alternatively, if there are no shared dofs a function space is described as **discontinuous** (fully or in a particular direction).

The mixed FEM formulation is built on a foundation set of four function spaces described below.

• w0 is the space of scalar functions with full continuity. The shared dofs lie on cell vertices in the lowest order FEM and on all three entities in higher order FEM.

- w1 is the space of vector functions with full continuity in the tangential direction only. In the lowest order FEM the shared dofs lie on cell edges for each component, whereas in higher order they also lie on cell faces.
- w2 is the space of vector functions with full continuity in the normal direction only. The shared dofs lie on cell faces for each component.
- w3 is the space of scalar functions with full discontinuity. All dofs lie within the cell volume and are not shared across the cell boundaries.

Additional function spaces required for representation of scalar or component-wise vector variables are:

- wtheta is the space of scalar functions based on the vertical part of w2, discontinuous in the horizontal and continuous in the vertical;
- w2v is the space of vector functions based on the vertical part of w2, discontinuous in the horizontal and continuous in the vertical;
- w2h is the space of vector functions based on the horizontal part of w2, continuous in the horizontal and discontinuous in the vertical.

Since the Dynamo0.3 API operates on columns of data, function spaces are categorised as continuous or discontinuous with regard to their horizontal continuity.

- Continuous function spaces are w0, w1, w2 and w2h;
- **Discontinuous** function spaces are w3, wtheta and w2v.

Two additional function space metadata descriptors as mentioned in sections above are:

- ANY\_W2 for any type of w2 function spaces;
- ANY\_SPACE for when the function space of the modified argument(s) cannot be determined and/or for when a Kernel has been written so that it works with fields on any of the available spaces.

As mentioned previously, both ANY\_W2 and ANY\_SPACE function space types are treated as continuous.

Horizontally discontinuous function spaces and fields over them will not need colouring so PSyclone does not perform it. If such attempt is made, PSyclone will raise a Generation Error in the **Dynamo0p3ColourTrans** transformation (see *Transformations* for more details on transformations). An example of fields iterating over a discontinuous function space wtheta is given in examples/dynamo/eg9, with the GH\_READWRITE access descriptor denoting an update to the relevant fields. This example also demonstrates how to only colour loops over continuous function spaces when transformations are applied.

## **Optional Field Metadata**

A field entry in the meta\_args array may have an optional fourth element. This element describes either a stencil access or, for inter-grid kernels, which mesh the field is on. Since an inter-grid kernel is not permitted to have stencil accesses, these two options are mutually exclusive. The metadata for each case is described in the following sections.

## **Stencil Metadata**

Stencil metadata specifies that the corresponding field argument is accessed as a stencil operation within the Kernel. Stencil metadata only makes sense if the associated field is read within a Kernel i.e. it only makes sense to specify stencil metadata if the first entry is GH\_FIELD and the second entry is GH\_READ.

Stencil metadata is written in the following format:

STENCIL (type)

9.3. Kernel 61

where type may be one of X1D, Y1D, XORY1D or CROSS. As the stencil extent (the maximum distance from the central cell that the stencil extends) is not provided in the metadata, it is expected to be provided by the algorithm writer as part of the invoke call (see Section *Stencils*). As there is currently no way to specify a fixed extent value for stencils in the Kernel metadata, Kernels must therefore be written to support different values of extent (i.e. stencils with a variable number of cells).

The XORY1D stencil type indicates that the Kernel can accept either X1D or Y1D stencils. In this case it is up to the algorithm developer to specify which of these it is from the algorithm layer as part of the invoke call (see Section *Stencils*).

For example, the following stencil (with extent=2):

```
| 4 | 2 | 1 | 3 | 5 |
```

#### would be declared as

```
STENCIL(X1D)
```

and the following stencil (with extent=2)

#### would be declared as

```
STENCIL (CROSS)
```

Below is an example of stencil information within the full kernel metadata.

```
type(arg_type) :: meta_args(3) = (/
    arg_type(GH_FIELD, GH_INC, W1),
    arg_type(GH_FIELD, GH_READ, W2H, STENCIL(CROSS)),
    arg_type(GH_OPERATOR, GH_READ, W1, W2H)
    /)
```

There is a full example of this distributed with PSyclone. It may be found in examples/dynamo/eg5.

# **Inter-Grid Metadata**

The alternative form of the optional fourth metadata argument for a field specifies which mesh the associated field is on. This is required for inter-grid kernels which perform prolongation or restriction operations on fields (or field vectors) existing on grids of different resolutions.

Mesh metadata is written in the following format:

```
mesh_arg=type
```

where type may be one of GH\_COARSE or GH\_FINE. Any kernel having a field argument with this metadata is assumed to be an inter-grid kernel and, as such, all of its other arguments (which must also be fields) must have it specified too. An example of the metadata for such a kernel is given below:

```
type(arg_type) :: meta_args(2) = (/
    arg_type(GH_FIELD, GH_INC, ANY_SPACE_1, mesh_arg=GH_COARSE), &
```

```
arg_type(GH_FIELD, GH_READ, ANY_SPACE_2, mesh_arg=GH_FINE ) &
/)
```

Note that an inter-grid kernel must have at least one field (or field-vector) argument on each mesh type. Fields that are on different meshes cannot be on the same function space while those on the same mesh must also be on the same function space.

## Column-wise Operators (CMA)

In this section we provide example metadata for each of the three recognised kernel types involving CMA operators.

Column-wise operators are constructed from cell-wise (local) operators. Therefore, in order to **assemble** a CMA operator, a kernel must have at least one read-only LMA operator, e.g.:

CMA operators (and their inverse) are **applied** to fields. Therefore any kernel of this type must have one read-only CMA operator, one read-only field and a field that is updated, e.g.:

**Matrix-matrix** kernels compute the product/linear combination of CMA operators. They must therefore have one such operator that is updated while the rest are read-only. They may also have read-only scalar arguments, e.g.:

```
type(arg_type) :: meta_args(3) = (/
    arg_type(GH_COLUMNWISE_OPERATOR, GH_WRITE, ANY_SPACE_1, ANY_SPACE_2), &
    arg_type(GH_COLUMNWISE_OPERATOR, GH_READ, ANY_SPACE_1, ANY_SPACE_2), &
    arg_type(GH_COLUMNWISE_OPERATOR, GH_READ, ANY_SPACE_1, ANY_SPACE_2), &
    arg_type(GH_REAL, GH_READ) /)
```

**Note:** The order with which arguments are specified in metadata for CMA kernels does not affect the process of identifying the type of kernel (whether it is assembly, matrix-matrix etc.)

# meta\_funcs

The (optional) second component of kernel metadata specifies whether any quadrature or evaluator data is required for a given function space. (If no quadrature or evaluator data is required then this metadata should be omitted.) Consider the following kernel metadata:

9.3. Kernel 63

The arg\_type component of this metadata describes a kernel that takes three arguments (an operator, a field and an integer scalar). Following the meta\_args array we now have a meta\_funcs array. This allows the user to specify that the kernel requires basis functions (gh\_basis) and/or the differential of the basis functions (gh\_diff\_basis) on one or more of the function spaces associated with the arguments listed in meta\_args. In this case we require both for the W0 function space but only basis functions for W1.

## gh shape and gh evaluator targets

If a kernel requires basis or differential-basis functions then the metadata must also specify the set of points on which these functions are required. This information is provided by the gh\_shape component of the metadata. Currently PSyclone supports two shapes; gh\_quadrature\_XYoZ for Gaussian quadrature points and gh\_evaluator for evaluation (of the basis/differential-basis functions) at nodal points. For the latter, there are two options: if an evaluator is required for multiple function spaces then these can be specified using the additional gh\_evaluator\_targets metadata entry. This entry is a one-dimensional, integer array containing the desired function spaces. For example, to request basis/differential-basis functions evaluated on both W0 and W1 the metadata would be:

```
integer :: gh_shape = gh_evaluator
integer :: gh_evaluator_targets(2) = (/W0, W1/)
```

The kernel must have an argument (field or operator) on each of the function spaces listed in gh\_evaluator\_targets. The default behaviour if gh\_evaluator\_targets is not specified is to provide evaluators for each function space associated with the quantities that the kernel is updating. All necessary data is extracted in the PSy layer and passed to the kernel(s) as required - nothing is required from the Algorithm layer. If a kernel requires quadrature on the other hand, the Algorithm writer must supply a quadrature\_type object as the last argument to the kernel (see Section *Quadrature*).

Note that it is an error for kernel metadata to specify a value for gh\_shape if no basis or differential-basis functions are required. It is also an error to specify gh\_evaluator\_targets if the kernel does not require an evaluator (i.e. gh\_shape != gh\_evaluator).

## iterates over

The fourth type of metadata provided is ITERATES\_OVER. This specifies that the Kernel has been written with the assumption that it is iterating over the specified entity. For user-supplied kernels this currently only has one valid value which is CELLS.

### **Procedure**

The fifth and final type of metadata is procedure metadata. This specifies the name of the Kernel subroutine that this metadata describes.

For example:

```
procedure, nopass :: my_kernel_subroutine
```

# 9.3.6 Subroutine

## **Rules for General-Purpose Kernels**

The arguments to general-purpose kernels (those that do not involve either CMA operators or prolongation/restriction operations) follow a set of rules which have been specified for the Dynamo0.3 API. These rules are encoded in the generate() method within the Argordering abstract class in the dynamo0p3.py file. The rules, along with PSyclone's naming conventions, are:

- 1. If an LMA operator is passed then include the cells argument, cells is an integer and has intent in.
- 2. Include nlayers, the number of layers in a column. nlayers is an integer and has intent in.
- 3. For each scalar/field/vector\_field/operator in the order specified by the meta\_args metadata:
  - (a) if the current entry is a scalar quantity then include the Fortran variable in the argument list. The intent is determined from the metadata (see *meta\_args* for an explanation).
  - (b) if the current entry is a field then include the field array. The field array name is currently specified as being "field\_"<argument\_position>"\_"<field\_function\_space>. A field array is a real array of type r\_def and dimensioned as the unique degrees of freedom for the space that the field is on. This value is passed in separately. Again, the intent is determined from the metadata (see *meta\_args*).
    - i. If the field entry has a stencil access then add an integer stencil-size argument with intent in. This will supply the number of cells in the stencil.
    - ii. If the field entry stencil access is of type XORY1D then add an integer direction argument with intent in.
  - (c) if the current entry field vector each dimension is then for of the vector, include a field array. The field array name is specified as being using "field\_"<argument\_position>"\_"<field\_function\_space>"\_v"<vector\_position>. A field array in a field vector is declared in the same way as a field array (described in the previous step).
  - (d) if the current entry is an operator then first include a dimension size. This is an integer. The name of this size is operator\_name>"\_ncell\_3d". Next include the operator. This is a real array of type r\_def and is 3 dimensional. The first two dimensions are the local degrees of freedom for the to and from function spaces respectively. The third dimension is the dimension size mentioned before. The name of the operator is "op\_"<argument\_position>. Again the intent is determined from the metadata (see meta\_args).
- 4. For each function space in the order they appear in the metadata arguments (the to function space of an operator is considered to be before the from function space of the same operator as it appears first in lexicographic order)
  - (a) Include the number of local degrees of freedom (i.e. number per-cell) for the function space. This is an integer and has intent in. The name of this argument is "ndf\_"<field\_function\_space>.
  - (b) If there is a field on this space
    - i. Include the unique number of degrees of freedom for the function space. This is an integer and has intent in. The name of this argument is "undf "<field function space>.
    - ii. Include the dofmap for this function space. This is an integer array with intent in. It has one dimension sized by the local degrees of freedom for the function space.
  - (c) For each operation on the function space (basis, diff\_basis, orientation) in the order specified in the metadata

9.3. Kernel 65

- i. If it is a basis or differential basis function, include the associated argument. This is a real array of kind r\_def with intent in. The rank and extents of this array depend upon the gh\_shape:
  - A. If gh\_shape is gh\_evaluator then basis and diff basis are real arrays of rank 3 with extent (dimension, number\_of\_dofs, np\_xyz)
  - B. If gh\_shape is gh\_quadrature\_xyoz then basis and diff basis are real arrays of rank 4 with extent (dimension, number\_of\_dofs, np\_xy, np\_z)

where dimension is 1 or 3 and depends upon the function space and whether or not it is a basis or a differential basis function. For the former it is (w0=1, w1=3, w2=3, w3=1, wtheta=1, w2h=3, w2v=3, any\_w2=3). For the latter it is (w0=3, w1=3, w2=1, w3=3, wtheta=3, w2h=1, w2v=1, any\_w2=3). number\_of\_dofs is the number of degrees of freedom (dofs) associated with the function space and np\_\* are the number of points to be evaluated: i) \*\_xyz in all directions (3D); ii) \*\_xy in the horizontal plane (2D); iii) \*\_x, \*\_y in the horizontal (1D); and iv) \*\_z in the vertical (1D). The name of the argument is "basis\_"<field\_function\_space> or "diff\_basis\_"<field\_function\_space>, as appropriate.

- ii. If it is an orientation array, include the associated argument. The argument is an integer array with intent in. There is one dimension of size the local degrees of freedom for the function space. The name of the array is "orientation\_"<field\_function\_space>.
- 5. If Quadrature or an Evaluator is required (this is the case if any of the function spaces require basis or differential basis functions)
  - (a) include integer, scalar arguments with intent in that specify the extent of the basis/diff-basis arrays:
    - i. If gh\_shape is gh\_evaluator then pass np\_xyz for each function space on which an evaluator is required, in the order in which they are encountered in the metadata; i.e. if gh\_evaluator\_targets has been specified then the ordering of function spaces in that list defines this ordering. Otherwise, the order in which the updated (written-to) kernel arguments appear in the argument list is used.
    - ii. if gh\_shape is gh\_quadrature\_XYoZ then pass np\_xy and np\_z.
  - (b) if Quadrature is required (gh\_shape is of type gh\_quadrature\_\*) then include weights which are real arrays of kind r\_def:
    - i. If gh\_quadrature\_XYoZ pass in w\_XZ (np\_xy) and w\_Z (np\_z)

## **Examples**

For instance, if a kernel has only one written argument and requires an evaluator then its metadata might be:

then we only pass the basis functions evaluated on W0 (the space of the written kernel argument). The subroutine arguments will therefore be:

where local\_stencil is the operator, xdata, ydata etc. are the three components of the field vector and map\_w0 is the dof map for the W0 function space.

If instead, gh\_evaluator\_targets is specified in the metadata:

then we will need to pass two sets of basis functions (evaluated at W0 and at W1):

#### **Rules for CMA Kernels**

Kernels involving CMA operators are restricted to just three types; assembly, application/inverse-application and matrix-matrix. We give the rules for each of these in the sections below.

#### **Assembly**

An assembly kernel requires the column-banded dofmap for both the to- and from-function spaces of the CMA operator being assembled as well as the number of dofs for each of the dofmaps. The full set of rules is:

- 1. Include the cell argument. cell is an integer and has intent in.
- 2. Include nlayers, the number of layers in a column. nlayers is an integer and has intent in.
- 3. Include the number of cells in the 2D mesh, ncell\_2d, which is an integer with intent in.
- 4. Include the total number of cells, ncell 3d, which is an integer with intent in.
- 5. For each argument in the meta\_args metadata array:
  - (a) If it is a LMA operator, include a real, 3-dimensional array of type r\_def. The first two dimensions are the local degrees of freedom for the to and from spaces, respectively. The third dimension is ncell\_3d.
  - (b) If it is a CMA operator, include a real, 3-dimensional array of type r\_def. The first dimension is is "bandwidth\_"<operator\_name>, the second is "nrow\_"<operator\_name>, and the third is ncell\_2d.
    - i. Include the number of rows in the banded matrix. This is an integer with intent in and is named as "nrow\_"<operator\_name>.

9.3. Kernel 67

- ii. If the from-space of the operator is *not* the same as the to-space then include the number of columns in the banded matrix. This is an integer with intent in and is named as "ncol\_"<operator\_name>.
- iii. Include the bandwidth of the banded matrix. This is an integer with intent in and is named as "bandwidth\_"<operator\_name>.
- iv. Include banded-matrix parameter alpha. This is an integer with intent in and is named as "alpha\_"<operator\_name>.
- v. Include banded-matrix parameter beta. This is an integer with intent in and is named as "beta\_"<operator\_name>.
- vi. Include banded-matrix parameter gamma\_m. This is an integer with intent in and is named as "gamma\_m\_"<operator\_name>.
- vii. Include banded-matrix parameter gamma\_p. This is an integer with intent in and is named as "gamma\_p\_"<operator\_name>.
- (c) If it is a field or scalar argument then include arguments following the same rules as for general-purpose kernels.
- 6. For each unique function space in the order they appear in the metadata arguments (the to function space of an operator is considered to be before the from function space of the same operator as it appears first in lexicographic order):
  - (a) Include the number of degrees of freedom per cell for the space. This is an integer with intent in. The name of this argument is "ndf\_"<arg\_function\_space>.
  - (b) If there is a field on this space then:
    - i. Include the unique number of degrees of freedom for the function space. This is an integer and has intent in. The name of this argument is "undf\_"<field\_function\_space>.
    - ii. Include the dofmap for this space. This is an integer array with intent in. It has one dimension sized by the local degrees of freedom for the function space.
  - (a) If the CMA operator has this space as its to/from space then include the column-banded dofmap, the list of offsets for the to/from-space. This is an integer array of rank 2. The first dimension is "ndf\_"<arg\_function\_space>`` and the second is nlayers.

### Application/Inverse-Application

A kernel applying a CMA operator requires the column-indirection dofmap for both the to- and from-function spaces of the CMA operator. Since it does not have any LMA operator arguments it does not require the ncell\_3d and nlayers scalar arguments. (Since a column-wise operator is, by definition, assembled for a whole column, there is no loop over levels when applying it.) The full set of rules is then:

- 1. Include the cell argument. cell is an integer and has intent in.
- 2. Include the number of cells in the 2D mesh, ncell\_2d, which is an integer with intent in.
- 3. For each argument in the meta\_args metadata array:
  - (a) If it is a field, include the field array. This is a real array of kind r def of rank 1. The field is currently and is array name specified "field\_"<argument\_position>"\_"<field\_function\_space>. The extent of the array is the number of unique degrees of freedom for the function space that the field is on. This value is passed in separately. The intent of the argument is determined from the metadata (see meta\_args).
  - (b) If it is a CMA operator, include it and its associated parameters (see Rule 5 of CMA Assembly kernels).

- 4. For each of the unique function spaces encountered in the metadata arguments (the to function space of an operator is considered to be before the from function space of the same operator as it appears first in lexicographic order):
  - (a) Include the number of degrees of freedom per cell for the associated function space. This is an integer with intent in. The name of this argument is "ndf\_"<field\_function\_space>.
  - (b) Include the number of unique degrees of freedom for the associated function space. This is an integer with intent in. The name of this argument is "undf "<field function space>.
  - (c) Include the dofmap for this function space. This is a rank-1 integer array with extent equal to the number of degrees of freedom of the space ("ndf\_"<field\_function\_space>).
- 5. Include the indirection map for the to-space of the CMA operator. This is a rank-1 integer array with extent
- 6. If the from-space of the operator is *not* the same as the to-space then include the indirection map for the from-space of the CMA operator. This is a rank-1 integer array with extent ncol.

#### **Matrix-Matrix**

Does not require any dofmaps and also does not require the nlayers and ncell\_3d scalar arguments. The full set of rules are then:

- 1. Include the cell argument. cell is an integer and has intent in.
- 2. Include the number of cells in the 2D mesh, ncell\_2d, which is an integer with intent in.
- 3. For each CMA operator or scalar argument specifed in metadata:
  - (a) If it is a CMA operator, include it and its associated parameters (see Rule 5 of CMA Assembly kernels).
  - (b) If it is a scalar argument include the corresponding Fortran variable in the argument list with intent in.

## **Rules for Inter-Grid Kernels**

As already specified, inter-grid kernels are only permitted to take fields and/or field-vectors as arguments. Fields (and field-vectors) that are on different meshes must be on different function spaces. Fields on the same mesh must also be on the same function space.

Argument ordering follows the general pattern used for 'normal' kernels with field data being followed by dofmap data. The rules for arguments to inter-grid kernels are as follows:

- 1. Include nlayers, the number of layers in a column. nlayers is an integer and has intent in.
- 2. Include the cell\_map for the current cell (column). This is an integer array of rank one and intent in which provides the mapping from the coarse to the fine mesh. It has extent ncell\_f\_per\_c.
- 3. Include ncell\_f\_per\_c, the number of fine cells per coarse cell. This is an integer and has intent in.
- 4. Include ncell\_f, the number of cells (columns) in the fine mesh. This is an integer and has intent in.
- 5. For each argument in the meta\_args metadata array (which must be a field or field-vector):
  - (a) Pass in field data as done for a regular kernel.
- 6. For each unique function space (of which there will currently be two) in the order in which they are encountered in the meta\_args metadata array, include dofmap information:

If the dofmap is associated with an argument on the fine mesh:

(a) Include ndf\_fine, the number of DoFs per cell for the FS of the field on the fine mesh.

9.3. Kernel 69

- (b) Include undf\_fine, the number of unique DoFs per cell for the FS of the field on the fine mesh.
- (c) Include dofmap\_fine, the *whole* dofmap for the fine mesh. This is an integer array of rank two with intent in. The extent of the first dimension is ndf fine and that of the second is ncell f.

else, the dofmap is associated with an argument on the coarse mesh:

- (a) Include undf\_coarse, the number of unique DoFs for the coarse field. This is an integer with intent in.
- (b) Include dofmap\_coarse, the dofmap for the current cell (column) in the coarse mesh. This is an integer array of rank one and has intent in.

## 9.4 Built-ins

The basic concept of a PSyclone Built-in is described in the *Built-ins* section. In the Dynamo 0.3 API, calls to Built-ins generally follow a convention that the field/scalar written to comes first in the argument list. Dynamo 0.3 Built-ins must conform to the following four rules:

- 1. Built-in kernels must have one and only one modified (i.e. written to) argument.
- 2. There must be at least one field in the argument list. This is so that we know the number of dofs to iterate over.
- 3. Kernel arguments must be either fields or scalars.
- 4. All field arguments to a given Built-in must be on the same function space. This is because all current Built-ins iterate over dofs and therefore all fields should have the same number. It also means that we can determine the number of dofs uniquely when a scalar is written to.

The Built-ins supported for the Dynamo 0.3 API are listed in the related subsections, grouped by the mathematical operation they perform. For clarity, the calculation performed by each Built-in is described using Fortran array syntax; this does not necessarily reflect the actual implementation of the Built-in (*e.g.* it could be implemented by PSyclone generating a call to an optimised maths library).

## 9.4.1 Naming scheme

The supported Built-ins in the Dynamo 0.3 API are named according to the scheme presented below. Any new Built-in needs to comply with these rules.

- 1. Ordering of arguments in Built-ins calls follows *LHS* (*result*) <- *RHS* (*operation on arguments*) direction, except where a Built-in returns the *LHS* result to one of the *RHS* arguments. In that case ordering of arguments remains as in the *RHS* expression, with the returning *RHS* argument written as close to the *LHS* as it can be without affecting the mathematical expression.
- 2. Field names begin with upper case in short form (e.g. **X**, **Y**, **Z**) and any case in long form (e.g. **Field1**, **field**).
- 3. Scalar names begin with lower case: e.g. **a**, **b**, are **scalar1**, **scalar2**. Special names for scalars are: **constant** (or **c**), **innprod** (inner/scalar product of two fields) and **sumfld** (sum of a field).
- 4. Arguments in Built-ins variable declarations and constructs (PSyclone Fortran and Python definitions):
  - (a) Are always written in long form and lower case (e.g. field1, field2, scalar1, scalar2);
  - (b) *LHS* result arguments are always listed first;
  - (c) *RHS* arguments are listed in order of appearance in the mathematical expression, except when one of them is the *LHS* result.
- 5. Built-ins names in Fortran consist of:

- (a) *RHS* arguments in short form (e.g. **X**, **Y**, **a**, **b**) only;
- (b) Descriptive name of mathematical operation on RHSarguments in the <operationname>\_<RHSarg> for one RHS argument or <RHSargs>\_<operationname>\_<RHSargs> for more;
- (c) Prefix "inc\_" where the result is returned to one of the RHS arguments (i.e. "inc\_"<RHSargs>\_<operationname>\_<RHSargs>).
- 6. Built-ins names in Python definitions are similar to their Fortran counterparts, with a few differences:
  - (a) Operators and RHS arguments are all in upper case (e.g. X, Y, A, B, Plus, Minus);
  - (b) There are no underscores;
  - (c) Common prefix is "Dyn", common suffix is "Kern".

## 9.4.2 Addition

Built-ins which add (scaled) fields are denoted with the keyword plus.

## X\_plus\_Y

**X\_plus\_Y** (field3, field1, field2)

Sums two fields (Z = X + Y):

```
field3(:) = field1(:) + field2(:)
```

#### where:

- type(field\_type), intent(out) :: field3
- type(field\_type), intent(in) :: field1
- type(field\_type), intent(in) :: field2

## inc\_X\_plus\_Y

inc\_X\_plus\_Y (field1, field2)

Adds the second field to the first and returns it (X = X + Y):

```
field1(:) = field1(:) + field2(:)
```

### where:

- type(field\_type), intent(inout) :: field1
- type(field\_type), intent(in) :: field2

## aX\_plus\_Y

 $aX_plus_Y$  (field3, scalar, field1, field2)

Performs Z = aX + Y:

```
field3(:) = scalar*field1(:) + field2(:)
```

9.4. Built-ins 71

#### where:

- real(r\_def), intent(in) :: scalar
- type(field\_type), intent(out) :: field3
- type(field\_type), intent(in) :: field1, field2

## inc\_aX\_plus\_Y

inc\_aX\_plus\_Y (scalar, field1, field2)

Performs X = aX + Y (increments the first field):

```
field1(:) = scalar*field1(:) + field2(:)
```

#### where:

- real(r\_def), intent(in) :: scalar
- type(field\_type), intent(inout) :: field1
- type(field\_type), intent(in) :: field2

## inc X plus bY

inc\_X\_plus\_bY (field1, scalar, field2)

Performs X = X + bY (increments the first field):

```
field1(:) = field1(:) + scalar*field2(:)
```

#### where:

- real(r\_def), intent(in) :: scalar
- type(field\_type), intent(inout) :: field1
- type(field\_type), intent(in) :: field2

### aX\_plus\_bY

aX\_plus\_bY (field3, scalar1, field1, scalar2, field2)

Performs Z = aX + bY:

```
field3(:) = scalar1*field1(:) + scalar2*field2(:)
```

### where:

- real(r\_def), intent(in) :: scalar1, scalar2
- type(field\_type), intent(out) :: field3
- type(field\_type), intent(in) :: field1, field2

## inc\_aX\_plus\_bY

inc\_aX\_plus\_bY (scalar1, field1, scalar2, field2)

Performs X = aX + bY (increments the first field):

```
field1(:) = scalar1*field1(:) + scalar2*field2(:)
```

#### where:

- real(r\_def), intent(in) :: scalar1, scalar2
- type(field\_type), intent(inout) :: field1
- type(field\_type), intent(in) :: field2

## 9.4.3 Subtraction

Built-ins which subtract (scaled) fields are denoted with the keyword minus.

## X\_minus\_Y

**X\_minus\_Y** (field3, field1, field2)

Subtracts the second field from the first and stores the result in the third (Z = X - Y):

```
field3(:) = field1(:) - field2(:)
```

### where:

- type(field\_type), intent(out) :: field3
- type(field\_type), intent(in) :: field1
- type(field\_type), intent(in) :: field2

## inc\_X\_minus\_Y

inc\_X\_minus\_Y (field1, field2)

Subtracts the second field from the first and returns it (X = X - Y):

```
field1(:) = field1(:) - field2(:)
```

## where:

- type(field\_type), intent(inout) :: field1
- type(field\_type), intent(in) :: field2

### aX minus Y

**aX\_minus\_Y** (field3, scalar, field1, field2)

Performs Z = aX - Y:

```
field3(:) = scalar*field1(:) - field2(:)
```

9.4. Built-ins 73

#### where:

- real(r\_def), intent(in) :: scalar
- type(field\_type), intent(out) :: field3
- type(field\_type), intent(in) :: field1, field2

## X\_minus\_bY

**X\_minus\_bY** (field3, field1, scalar, field2)

Performs Z = X - bY:

```
field3(:) = field1(:) - scalar*field2(:)
```

#### where:

- real(r\_def), intent(in) :: scalar
- type(field\_type), intent(out) :: field3
- type(field\_type), intent(in) :: field1, field2

## inc X minus bY

inc\_X\_minus\_bY (field1, scalar, field2)

Performs X = X - bY (increments the first field):

```
field1(:) = field1(:) - scalar*field2(:)
```

#### where:

- real(r\_def), intent(in) :: scalar
- type(field\_type), intent(inout) :: field1
- type(field\_type), intent(in) :: field2

## 9.4.4 Multiplication

Built-ins which multiply (scaled) fields are denoted with the keyword times.

### X times Y

**X\_times\_Y** (field3, field1, field2)

Multiplies two fields together and returns the result in a third field (Z = X\*Y):

```
field3(:) = field1(:)*field2(:)
```

### where:

- type(field\_type), intent(out) :: field3
- type(field\_type), intent(in) :: field1, field2

## inc X times Y

```
inc_X_times_Y (field1, field2)
```

Multiplies the first field by the second and returns it (X = X\*Y):

```
field1(:) = field1(:)*field2(:)
```

#### where:

- type(field\_type), intent(inout) :: field1
- type(field\_type), intent(in) :: field2

## inc\_aX\_times\_Y

```
inc_aX_times_Y (scalar, field1, field2)
```

Performs X = a\*X\*Y (increments the first field):

```
field1(:) = scalar*field1(:)*field2(:)
```

#### where:

- real(r\_def), intent(in) :: scalar
- type(field\_type), intent(inout) :: field1
- type(field\_type), intent(in) :: field2

## 9.4.5 Scaling

Built-ins which scale fields are technically cases of multiplying a field by a scalar and are hence also denoted with the keyword **times**.

## a\_times\_X

```
a_times_X (field2, scalar, field1)
```

Multiplies a field by a scalar and stores the result in a second field (Y = a\*X):

```
field2(:) = scalar*field1(:)
```

#### where:

- real(r\_def), intent(in) :: scalar
- type(field\_type), intent(out) :: field2
- type(field\_type), intent(in) :: field1

## inc a times X

## inc\_a\_times\_X (scalar, field)

Multiplies a field by a scalar value and returns the field (X = a\*X):

9.4. Built-ins 75

```
field(:) = scalar*field(:)
```

#### where:

- real(r\_def), intent(in) :: scalar
- type(field\_type), intent(inout) :: field

## 9.4.6 Division

Built-ins which divide (scaled) fields are denoted with the keyword divideby.

## X\_divideby\_Y

**X\_divideby\_Y** (field3, field1, field2)

Divides the first field by the second and returns the result in the third (Z = X/Y):

```
field3(:) = field1(:)/field2(:)
```

#### where:

- type(field\_type), intent(out) :: field3
- type(field\_type), intent(in) :: field1, field2

## inc\_X\_divideby\_Y

inc\_X\_divideby\_Y (field1, field2)

Divides the first field by the second and returns it (X = X/Y):

```
field1(:) = field1(:)/field2(:)
```

### where:

- type(field\_type), intent(inout) :: field1
- type(field\_type), intent(in) :: field2

## 9.4.7 Setting to value

Built-ins which set field elements to some value and hence are denoted with the keyword setval.

## setval\_c

setval\_c (field, constant)

Sets all elements of the field *field* to the value *constant* (X = c):

```
field(:) = constant
```

### where:

• type(field\_type), intent(out) :: field

• real(r\_def), intent(in) :: constant

**Note:** The field may be on any function space.

### setval X

```
setval_X (field2, field1)
```

Sets a field *field2* equal to field *field1* (Y = X):

```
field2(:) = field1(:)
```

#### where:

- type(field\_type), intent(out) :: field2
- type(field\_type), intent(in) :: field1

## 9.4.8 Raising to power

Built-ins which raise field elements to an exponent are denoted with the keyword **powreal** for real exponent or **powint** for integer exponent.

## inc\_X\_powreal\_a

inc\_X\_powreal\_a (field, rscalar)

Raises a field to a real scalar value and returns the field  $(X = X^{**a})$ :

```
field(:) = field(:)**rscalar
```

#### where:

- type(field\_type), intent(inout) :: field
- real(r\_def), intent(in) :: rscalar

## inc\_X\_powint\_n

inc\_X\_powint\_n (field, iscalar)

Raises a field to an integer scalar value and returns the field  $(X = X^{**}n)$ :

```
field(:) = field(:)**iscalar
```

### where:

- type(field\_type), intent(inout) :: field
- integer(i\_def), intent(in) :: iscalar

9.4. Built-ins 77

## 9.4.9 Inner product

Built-ins which calculate the inner product of two fields or of a field with itself are denoted with the keyword **inner-product**.

## X\_innerproduct\_Y

**X\_innerproduct\_Y** (innprod, field1, field2)

Computes the inner product of the fields field1 and field2, i.e.:

```
innprod = SUM(field1(:)*field2(:))
```

#### where:

- real(r\_def), intent(out) :: innprod
- type(field\_type), intent(in) :: field1, field2

**Note:** When used with distributed memory this Built-in will trigger the addition of a global sum which may affect the performance and/or scalability of the code.

## X\_innerproduct\_X

X\_innerproduct\_X (innprod, field)

Computes the inner product of the field *field1* by itself, *i.e.*:

```
innprod = SUM(field(:)*field(:))
```

#### where:

- real(r\_def), intent(out) :: innprod
- type(field\_type), intent(in) :: field

**Note:** When used with distributed memory this Built-in will trigger the addition of a global sum which may affect the performance and/or scalability of the code.

## 9.4.10 Sum of elements

Built-in which sums the elements of a field is denoted with the keyword sum.

## sum\_X

sum\_X (sumfld, field)

Sums all of the elements of the field field and returns the result in the scalar variable sumfld:

```
sumfld = SUM(field(:))
```

where:

```
• real(r_def), intent(out) :: sumfld
```

• type(field\_type), intent(in) :: field

**Note:** When used with distributed memory this Built-in will trigger the addition of a global sum which may affect the performance and/or scalability of the code.

# 9.5 Boundary Conditions

In the Dynamo0.3 API, boundary conditions for a field or LMA operator can be enforced by the algorithm developer by calling the Kernels enforce\_bc\_type or enforce\_operator\_bc\_type, respectively. These kernels take a field or operator as input and apply boundary conditions. For example:

The particular boundary conditions that are applied are not known by PSyclone, PSyclone simply recognises these kernels by their names and passes pre-specified dofmap and boundary\_value arrays into the kernel implementations, the contents of which are set by the LFRic infrastructure.

Up to and including version 1.4.0 of PSyclone, boundary conditions were applied automatically after a call to matrix\_vector\_type if the field arguments were on a vector function space (one of w1, w2, w2h or w2v). With the subsequent introduction of the ability to apply boundary conditions to operators this functionality is no longer required and has been removed.

Example eg4 in the examples/dynamo directory includes a call to enforce\_bc\_kernel\_type so can be used to see the boundary condition code that is added by PSyclone. See the README in the examples/dynamo directory for instructions on how to run this example.

An example of applying boundary conditions the kernel to operator is an enforce operator bc kernel mod.F90 in the <PSYCLONEHOME>/src/psyclone/tests/ test\_files/dynamo0p3 directory. Since operators are discontinuous quantities, updating their values can be safely performed in parallel (see Section Kernel). The GH\_READWRITE access is used for updating discontinuous operators (see subsection Valid Access Modes for more details).

## 9.6 Conventions

There is a convention in the Dynamo0.3 API kernel code that if the name of the operation being performed is <name> then a kernel file is <name>\_mod. [fF90], the name of the module inside the kernel file is <name>\_mod, the name of the kernel metadata in the module is <name>\_type and the name of the kernel subroutine in the module is <name>\_code. PSyclone does not need this convention to be followed apart from the stub generator (see the *Stub Generation* Section) where the name of the metadata to be parsed is determined from the module name.

The contents of the metadata is also usually declared private but this does not affect PSyclone.

Finally, the procedure metadata (located within the kernel metadata) usually has nopass specified but again this is ignored by PSyclone.

# 9.7 Configuration

### 9.7.1 Annexed DoFs

When a kernel iterates over dofs (rather than cells) for a continuous field using distributed memory (see the *Distributed Memory* Section), then PSyclone need only ensure that dofs owned by a processor are computed. However, for continuous fields, shared dofs at the boundary between processors must be replicated (as different cells share the same dof). Only one processor can own a dof, therefore processors will have continuous fields which contain dofs that the processor does not own. These unowned dofs are called *annexed* in the dynamo0.3 api and are a separate, but related, concept to field halos.

When a kernel that iterates over cells needs to read a continuous field then the annexed dofs must be up-to-date on all processors. If they are not then a halo exchange must be added. Currently PSyclone defaults, for kernels which iterate over dofs, to iterating over only owned dofs. This behaviour can be changed by setting *COMPUTE\_ANNEXED\_DOFS* to true in the *dynamo0.3* section of the configuration file (see the *Configuration* section). PSyclone will then generate code to iterate over both owned and annexed dofs, thereby reducing the number of halo exchanges required (at the expense of redundantly computing annexed dofs). For more details please refer to the *Dynamo0.3* developers section.

## 9.8 Transformations

This section describes the dynamo-api-specific transformations. In all cases, excepting **Dynamo0p3RedundantComputationTrans**, these transformations are specialisations of generic transformations described in the *Transformations* section. The difference between these transformations and the generic ones is that these perform dynamo-api-specific checks to make sure the transformations are valid. In practice these transformations perform the required checks then call the generic ones internally.

The use of the dynamo-api-specific transformations is exactly the same as the equivalent generic ones in all cases excepting **DynamoLoopFuseTrans**. In this case an additional optional argument **same\_space** has been added to the **apply** method. The reason for this is to allow loop fusion when one or more of the iteration-spaces is determined by a function space that is unknown by PSyclone at compile time. This is the case when the **ANY\_SPACE** function space is specified in the Kernel metadata. By default PSyclone will not allow loop fusion if it does not know the spaces are the same. The **same\_space** option allows the user to specify that the spaces are the same. This option should therefore be used with caution. Note, if PSyclone knows the spaces are different this option has no effect and the transformation will always raise an exception.

The **Dynamo0p3RedundantComputationTrans** transformation is only valid for the "Dynamo0p3" API. This is because this API is currently the only one that supports distributed memory. An example of redundant computation can be found in examples/dynamo/eg8.

The Dynamo-specific transformations currently available are given below. If the name of a transformation includes "Dynamo0p3" it means that the transformation is only valid for this particular API. If the name of the transformation includes "Dynamo" then it should work with all versions of the Dynamo API.

**Note:** Only the loop-colouring transformation is currently supported for loops that contain inter-grid kernels. Attempting to apply other transformation types will result in PSyclone raising an error.

## class psyclone.transformations.DynamoLoopFuseTrans

Performs error checking before calling the <code>apply()</code> method of the <code>base class</code> in order to fuse two Dynamo loops.

apply (node1, node2, same\_space=False)

Fuse the two Dynamo loops represented by node1 and node2. The optional same\_space flag asserts that

an unknown iteration space (i.e. any\_space) matches the other iteration space. This is set at the users own risk.

#### **Parameters**

- node1 (psyclone.dynamo0p3.DynLoop) First Loop to fuse.
- node2 (psyclone.dynamo0p3.DynLoop) Second Loop to fuse.

**Returns** two-tuple of modified schedule and Memento

Return type psyclone.psyGen.Schedule, psyclone.undoredo.Memento

**Raises** *TransformationError* – if either of the supplied loops contains an inter-grid kernel.

#### name

Returns the name of this transformation as a string.

**class** psyclone.transformations.**DynamoOMPParallelLoopTrans** (*omp\_schedule='static'*)

Dynamo-specific OpenMP loop transformation. Adds Dynamo specific validity checks. Actual transformation is done by the *base class*.

#### apply (node)

Perform Dynamo specific loop validity checks then call the apply () method of the base class.

#### name

Returns the name of this transformation as a string.

```
class psyclone.transformations.Dynamo0p3OMPLoopTrans(omp_schedule='static')
```

Dynamo 0.3 specific orphan OpenMP loop transformation. Adds Dynamo-specific validity checks. Actual transformation is done by base class.

```
apply (node, reprod=None)
```

Perform Dynamo 0.3 specific loop validity checks then call OMPLoopTrans.apply().

#### **Parameters**

- node (psyclone.psyGen.Node) the Node in the Schedule to check
- reprod (bool or None (default, which indicates to use the default from the config file))—if reproducible reductions should be used.

Raises TransformationError – if an OMP loop transform would create incorrect code.

#### name

Returns the name of this transformation as a string.

### class psyclone.transformations.Dynamo0p3ColourTrans

Split a Dynamo 0.3 loop over cells into colours so that it can be parallelised. For example:

9.8. Transformations 81

```
>>> schedule = invoke.schedule
>>>
>>> ctrans = Dynamo0p3ColourTrans()
>>> otrans = DynamoOMPParallelLoopTrans()
>>>
>>> # Colour all of the loops
>>> for child in schedule.children:
        cschedule, _ = ctrans.apply(child)
>>>
>>> # Then apply OpenMP to each of the colour loops
>>> schedule = cschedule
>>> for child in schedule.children:
>>>
        newsched, _ = otrans.apply(child.children[0])
>>>
>>> newsched.view()
```

Colouring in the Dynamo 0.3 API is subject to the following rules:

- Only kernels with an iteration space of CELLS and which modify a continuous field require colouring. Any other type of loop will cause this transformation to raise an exception.
- A kernel may have at most one field with 'INC' access
- A separate colour map will be required for each field that is coloured (if an invoke contains >1 kernel call)

## apply (node)

Performs Dynamo0.3-specific error checking and then uses the parent class to convert the Loop represented by node into a nested loop where the outer loop is over colours and the inner loop is over cells of that colour.

```
Parameters node (psyclone.dynamo0p3.DynLoop) – the loop to transform.
```

Returns 2-tuple of new schedule and memento of transform

```
Return type (psyclone.dynamo0p3.DynSchedule, psyclone.undoredo. Memento)
```

#### name

Returns the name of this transformation as a string.

## $\textbf{class} \hspace{0.1cm} \texttt{psyclone.trans} formations. \textbf{Dynamo0p3RedundantComputationTrans}$

This transformation allows the user to modify a loop's bounds so that redundant computation will be performed. Redundant computation can result in halo exchanges being modified, new halo exchanges being added or existing halo exchanges being removed.

- This transformation should be performed before any parallelisation transformations (e.g. for OpenMP) to the loop in question and will raise an exception if this is not the case.
- This transformation can not be applied to a loop containing a reduction and will again raise an exception
  if this is the case.
- This transformation can only be used to add redundant computation to a loop, not to remove it.
- This transformation allows a loop that is already performing redundant computation to be modified, but only if the depth is increased.

#### apply (loop, depth=None)

Apply the redundant computation tranformation to the loop loop. This transformation can be applied to loops iterating over 'cells or 'dofs'. if depth is set to a value then the value will be the depth of the field's halo over which redundant computation will be performed. If depth is not set to a value then redundant computation will be performed to the full depth of the field's halo.

## **Parameters**

- loop (psyclone.psyGen.DynLoop) the loop that we are transforming
- **depth** (*int* or *None*) the depth of the stencil. Defaults to None if a depth is not provided.

#### name

Returns the name of this transformation as a string.

9.8. Transformations 83

**CHAPTER** 

TEN

## **GOCEAN1.0 API**

## 10.1 Introduction

The GOcean 1.0 application programming interface (API) was originally designed to support ocean models that use the finite-difference scheme for two-dimensional domains. However, the approach is not specific to ocean models and can potentially be applied to any finite-difference code.

As with all PSyclone API's, the GOcean 1.0 API specifies how a user must write the Algorithm Layer and the Kernel Layer to allow PSyclone to generate the PSy Layer. These Algorithm and Kernel API's are discussed separately in the sections below. Before these we describe the functionality provided by the GOcean Library.

# 10.2 The GOcean Library

The use of PSyclone and the GOcean 1.0 API implies the use of a standard set of data types and associated infrastructure. This is provided by version 1.0 of the GOcean Library (GOLib v.1.0). Currently this library is distributed separately from PSyclone and is available from http://puma.nerc.ac.uk/trac/GOcean.

## 10.2.1 Grid

The GOLib contains a grid\_mod module which defines a grid\_type and associated constructor:

**Note:** The grid object itself must be declared with the target attribute. This is because each field object will contain a pointer to it.

The  $\mbox{grid\_type}$  constructor takes three arguments:

- 1. The type of grid (only ARAKAWA\_C is currently supported)
- 2. The boundary conditions on the domain for the *x*, *y* and *z* dimensions (see below). The value for the *z* dimension is currently ignored.

3. The 'index offset' - the convention used for indexing into offset fields.

Three types of boundary condition are currently supported:

Name	Description	
BC_NONE	No boundary conditions are applied.	
BC_EXTERNAL	Some external forcing is applied. This must be implemented by a kernel. The domain must	
	be defined with a T-point mask (see <i>The grid_init Routine</i> ).	
BC_PERIODIC	Periodic boundary conditions are applied.	

The infrastructure requires this information in order to determine the extent of the model grid.

The index offset is required because a model (kernel) developer has choice in how they actually implement the staggering of variables on a grid. This comes down to a choice of which grid points in the vicinity of a given T point have the same array (i, j) indices. In the diagram below, the image on the left corresponds to choosing those points to the South and West of a T point to have the same (i, j) index. That on the right corresponds to choosing those points to the North and East of the T point (this is the offset scheme used in the NEMO ocean model):



The GOcean 1.0 API supports these two different offset schemes, which we term OFFSET SW and OFFSET NE.

Note that the constructor does not specify the extent of the model grid. This is because this information is normally obtained by reading a file (a namelist file, a netcdf file etc.) which is specific to an application. Once this information has been obtained, a second routine, grid\_init, is provided with which to 'load' a grid object with state. This is discussed below.

### The grid init Routine

Once an application has determined the details of the model configuration, it must use this information to populate the grid object. This is done via a call to the grid\_init subroutine:

```
subroutine grid_init(grid, m, n, dxarg, dyarg, tmask)
!> The grid object to configure
  type(grid_type), intent(inout) :: grid
!> Dimensions of the model grid
  integer, intent(in) :: m, n
!> The (constant) grid spacing in x and y (m)
  real(wp), intent(in) :: dxarg, dyarg
!> Optional T-point mask specifying whether each grid point is
!! wet (1), dry (0) or external (-1).
  integer, dimension(m,n), intent(in), optional :: tmask
```

If no T-mask is supplied then this routine configures the grid appropriately for an all-wet domain with periodic boundary conditions in both the x- and y-dimensions. It should also be noted that currently only grids with constant resolution in x and y are supported by this routine.

## 10.2.2 Fields

Once a model has a grid defined it will require one or more fields. The GOLib contains a field\_mod module which defines an r2d\_field type (real, 2-dimensional field) and associated constructor:

```
use field_mod
...
!> Current ('now') sea-surface height at different grid points
type(r2d_field) :: sshn_u_fld, sshn_v_fld, sshn_t_fld
...
! Sea-surface height now (current time step)
sshn_u = r2d_field(model_grid, U_POINTS)
sshn_v = r2d_field(model_grid, V_POINTS)
sshn_t = r2d_field(model_grid, T_POINTS)
```

The constructor takes two arguments:

- 1. The grid on which the field exists
- 2. The type of grid point at which the field is defined (U\_POINTS, V\_POINTS, T\_POINTS or F\_POINTS)

Note that the grid object need not have been fully configured (by a call to grid\_init for instance) before it is passed into this constructor.

## 10.2.3 Example

PSyclone is distributed with a full example of the use of the GOcean Library. See <PSYCLONEHOME>/examples/gocean/shallow\_alg.f90. In what follows we will walk through a slightly cut-down example for a different program.

The following code illustrates the use of the GOLib in constructing an application:

```
program gocean2d
  use grid_mod ! From GOLib
  use field_mod ! From GOLib
  use model_mod
  use boundary_conditions_mod
  !> The grid on which our fields are defined. Must have the 'target'
  !! attribute because each field object contains a pointer to it.
  type(grid_type), target :: model_grid
  !> Current ('now') velocity component fields
  type(r2d_field) :: un_fld, vn_fld
  !> 'After' velocity component fields
  type(r2d_field) :: ua_fld, va_fld
  ! time stepping index
  integer :: istp
  ! Create the model grid. We use a NE offset (i.e. the {\it U}, {\it V} and {\it F}
  ! points immediately to the North and East of a T point all have the
  ! same i, j index). This is the same offset scheme as used by NEMO.
  model_grid = grid_type(ARAKAWA_C,
                         (/BC_EXTERNAL, BC_EXTERNAL, BC_NONE/), &
                         OFFSET_NE)
```

```
!! read in model parameters and configure the model grid
  CALL model_init (model_grid)
  ! Create fields on this grid
  ! Velocity components now (current time step)
  un_fld = r2d_field(model_grid, U_POINTS)
  vn_fld = r2d_field(model_grid, V_POINTS)
  ! Velocity components 'after' (next time step)
  ua_fld = r2d_field(model_grid, U_POINTS)
  va_fld = r2d_field(model_grid, V_POINTS)
  . . .
  !! time stepping
  do istp = nit000, nitend, 1
   call step(istp,
              ua_fld, va_fld, un_fld, vn_fld,
              . . . )
  end do
end program gocean2d
```

The model\_init routine is application specific since it must determine details of the model configuration being run, *e.g.* by reading a namelist file. An example might look something like:

```
subroutine model_init(grid)
 type(grid_type), intent(inout) :: grid
 !> Problem size, read from namelist
 integer :: jpiglo, jpjglo
 real(wp) :: dx, dy
 integer, dimension(:,:), allocatable :: tmask
  ! Read model configuration from namelist
 call read_namelist(jpiglo, jpjglo, dx, dy, &
                     nit000, nitend, irecord, &
                     jphgr_msh, dep_const, rdt, cbfr, visc)
  ! Set-up the T mask. This defines the model domain.
 allocate(tmask(jpiglo, jpjglo))
 call setup_tpoints_mask(jpiglo, jpjglo, tmask)
  ! Having specified the T points mask, we can set up mesh parameters
 call grid_init(grid, jpiglo, jpjglo, dx, dy, tmask)
  ! Clean-up. T-mask has been copied into the grid object.
 deallocate (tmask)
end subroutine model_init
```

Here, only grid\_type and the grid\_init routine come from the GOLib. The remaining code is all application specific.

Once the grid object is fully configured and all fields have been constructed, a simulation will proceed by performing calculations with those fields. In the example program given above, this calculation is performed in the time-stepping loop within the step subroutine. The way in which this routine uses Invoke calls is described in the *Invokes* Section.

# 10.3 Algorithm

The Algorithm is the top-level specification of the natural science implemented in the software. Essentially it consists of mesh setup, field declarations, initialisation of fields and (a series of) Kernel calls. Infrastructure to support these tasks is provided in version 1.0 of the GOcean library (see *The GOcean Library*).

## **10.3.1 Invokes**

The Kernels to call are specified through the use of Invokes, e.g.:

The location and number of these call invoke (...) statements within the source code is entirely up to the user. The only requirement is that PSyclone must be run on every source file that contains one or more Invokes. The body of each Invoke specifies the kernels to be called, the order in which they are to be applied and the fields (and scalars) that they work with.

Note that the kernel names specified in an Invoke are the names of the corresponding kernel *types* defined in the kernel metadata (see the *Kernel* Section). These are not the same as the names of the Fortran subroutines which contain the actual kernel code. The kernel arguments are typically field objects, as described in the *Fields* Section, but they may also be scalar quantities (real or integer).

In the example gocean2d program shown earlier, there is only one Invoke call and it is contained within the step subroutine:

```
subroutine step(istp,
                                        &
                ua, va, un, vn,
                sshn_t, sshn_u, sshn_v, &
                ssha_t, ssha_u, ssha_v, &
               hu, hv, ht)
 use kind_params_mod ! From GOLib
                 ! From GOLib
! From GOLib
 use grid_mod
 use field mod
 use model_mod, only: rdt ! The model time-step
 use continuity_mod, only: continuity
                      only: momentum_u, momentum_v
 use momentum_mod,
 use boundary_conditions_mod, only: bc_ssh, bc_solid_u
 !> The current time step
                 intent(inout) :: istp
 type(r2d_field), intent(inout) :: un, vn, sshn_t, sshn_u, sshn_v
 type(r2d_field), intent(inout) :: ua, va, ssha_t, ssha_u, ssha_v
 type (r2d_field), intent(inout) :: hu, hv, ht
 call invoke(
              continuity(ssha_t, sshn_t, sshn_u, sshn_v,
                        hu, hv, un, vn, rdt),
              momentum_u(ua, un, vn, hu, hv, ht,
                        ssha_u, sshn_t, sshn_u, sshn_v),
              momentum_v(va, un, vn, hu, hv, ht,
```

10.3. Algorithm 89

```
ssha_v, sshn_t, sshn_u, sshn_v), &
bc_ssh(istp, ssha_t), &
bc_solid_u(ua), &

...
)
end subroutine step
```

Note that in this example the grid was constructed for a model with 'external' boundary conditions. These boundary conditions are applied through several user-supplied kernels, two of which (bc\_ssh and bc\_solid\_u) are include in the above code fragment.

## 10.4 Kernel

The general requirements for the structure of a Kernel are explained in the *Kernel layer* section. This section explains the metadata and subroutine arguments that are specific to the GOcean 1.0 API.

### 10.4.1 Metadata

The metadata for a GOcean 1.0 API kernel has four components:

- 1. 'meta args',
- 2. 'iterates\_over',
- 3. 'index\_offset' and
- 4. 'procedure':

These are illustrated in the code below:

```
type, extends(kernel_type) :: my_kernel_type
   type(arg), dimension(...) :: meta_args = (/ ... /)
   integer :: iterates_over = ...
   integer :: index_offset = ...
contains
   procedure, nopass :: code => my_kernel_code
end type my_kernel_type
```

These four metadata elements are discussed in order in the following sections.

## Argument Metadata: meta\_args

The meta\_args array specifies information about data that the kernel code expects to be passed to it via its argument list. There is one entry in the meta\_args array for each scalar, field, or grid-property passed into the Kernel. Their ordering in the meta\_args array must be the same as that in the kernel code argument list. The entry must be of type arg which itself contains metadata about the associated argument. The size of the meta\_args array must correspond to the total number of scalars, fields and grid properties passed into the Kernel.

For example, if there are a total of two **field** entities being passed to the Kernel then the meta\_args array will be of size 2 and there will be two entries of type arg:

Argument-metadata (metadata contained within the brackets of an arg entry), describes either a scalar, a field or a grid property.

The first argument-metadata entry describes how the kernel will access the corresponding argument. As an example, the following meta\_args metadata describes four entries, the first one is written to by the kernel while the remaining three are only read.

```
type(arg) :: meta_args(4) = (/
    arg(WRITE, ...),
    arg(READ, ...),
    arg(READ, ...),
    arg(READ, ...)
    %
```

The second entry to argument-metadata (information contained within the brackets of an arg type) describes the type of data represented by the argument. This type falls into three categories; field data, scalar data and grid properties. For field data the metadata entry consists of the type of grid-point that field values are defined on. Since the GOcean API supports fields on an Arakawa C grid, the possible grid-point types are CU, CV, CF and CT. GOcean Kernels can also take scalar quantities as arguments. Since these do not live on grid-points they are specified as either R\_SCALAR or I\_SCALAR depending on whether the corresponding Fortran variable is a real or integer quantity. Finally, grid-property entries are used to specify any properties of the grid required by the kernel (e.g. the area of cells at U points or whether T points are wet or dry).

For example:

```
type(arg) :: meta_args(4) = (/
    arg(WRITE, CT, ...),
    arg(READ, CU, ...),
    arg(READ, R_SCALAR, ...),
    arg(READ, GRID_AREA_U)
    /)
```

Here, the first argument is a field on T points, the second is a field on U points, the fourth is a real scalar and the fifth is a property of the grid (cell area at U points).

The full list of supported grid properties in the GOcean 1.0 API is:

Name	Description	Туре
grid_area_t	Cell area at T point	Real array, rank=2
grid_area_u	Cell area at U point	Real array, rank=2
grid_area_v	Cell area at V point	Real array, rank=2
grid_mask_t	T-point mask (1=wet, 0=dry)	Integer array, rank=2
grid_dx_t	Grid spacing in x at T points	Real array, rank=2
grid_dx_u	Grid spacing in x at U points	Real array, rank=2
grid_dx_v	Grid spacing in x at V points	Real array, rank=2
grid_dy_t	Grid spacing in y at T points	Real array, rank=2
grid_dy_u	Grid spacing in y at U points	Real array, rank=2
grid_dy_v	Grid spacing in y at V points	Real array, rank=2
grid_lat_u	Latitude of U points (gphiu)	Real array, rank=2
grid_lat_v	Latitude of V points (gphiv)	Real array, rank=2
grid_dx_const	Grid spacing in x if constant	Real, scalar
grid_dy_const	Grid spacing in y if constant	Real, scalar
grid_x_min_index	Minimum X index	Integer, scalar
grid_x_max_index	Maximum X index	Integer, scalar
grid_y_min_index	Minimum Y index	Integer, scalar
grid_y_max_index	Maximum Y index	Integer, scalar

10.4. Kernel 91

These are stored in a dictionary named  $GRID\_PROPERTY\_DICT$  at the top of the gocean1p0.py file. All of the rank-two arrays have the first rank as longitude (x) and the second as latitude (y).

Scalars and fields contain a third argument-metadata entry which describes whether the kernel accesses the corresponding argument with a stencil. The value POINTWISE indicates that there is no stencil access. Metadata for a scalar field is limited to this value. Grid-property arguments have no third metadata argument. If there are no stencil accesses then the full argument metadata for our previous example will be:

If a kernel accesses a field using a stencil then the third argument metadata entry should take the form stencil (...). Note, a stencil access is only allowed for a field that is READ by a kernel.

In the GOcean API, fields are implemented as two-dimensional arrays. In Fortran, a standard 5-point stencil would look something like the following:

```
a(i,j) + a(i+1,j) + a(i-1,j) + a(i,j+1) + a(i,j-1)
```

If we view the above accesses as co-ordinates relative to the a(i,j) access we get (0,0), (1,0), (-1,0), (0,1), (0,-1). If we then view these accesses in graphical form with i being in the horizontal direction and j in the vertical and with a 1 indicating a (depth-1) access and a 0 indicating there is no access we get the following:

```
010
111
010
```

In the GOcean API a stencil access is captured as a triplet of integers (one row at a time from top to bottom) using the above view i.e.

```
stencil(010,111,010)
```

So far we have only considered depth-1 stencils. In our notation the depth of access is captured by the integer value (0 for no access, 1 for depth 1, 2 for depth 2 etc). For example:

```
a(i,j) + a(i,j+1) + a(i,j+2)
```

would be captured as:

```
stencil(020,010,000)
```

All forms of stencil can be **summarised** using this triplet notation up to a depth of 9 apart from the central a (i, j) value which can either be 0 (not accessed) or 1 (accessed). Note, the central value is not currently used by PSyclone. The notation is a **summary** in two ways

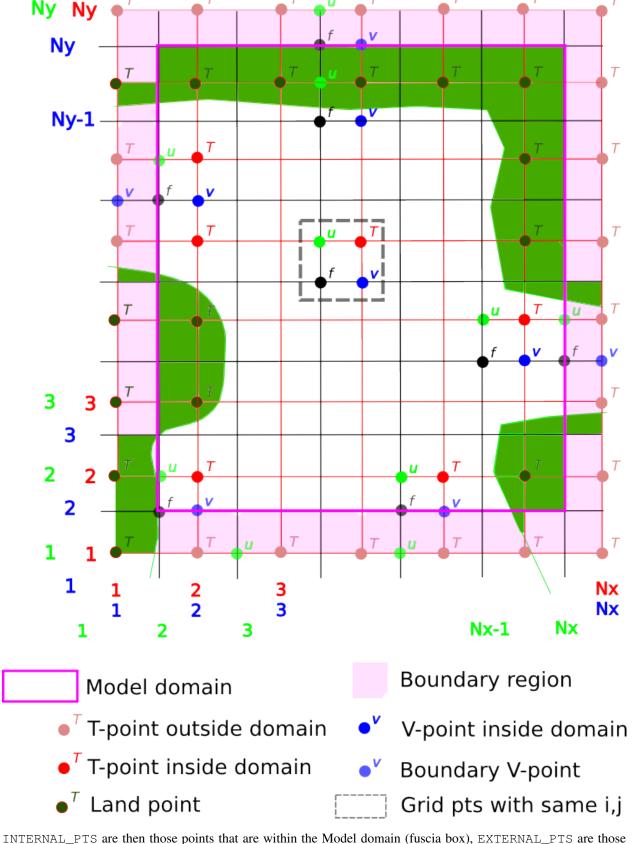
- 1. it only captures the depth of the stencil in a particular direction, not the actual accesses. Therefore, there is no way to distinguish between the stencil a (i+2, j) and the stencil a (i+1, j) + a(i+2, j).
- 2. when there are offsets for both i and j e.g. a(i+1, j+1) it only captures whether there is an access in that direction at a particular depth, not the details of the access. For example, there is no way to distinguish between a(i+2, j+2) and a(i+2, j+2) + a(i+1, j+2) + a(i+2, j+1).

Whilst the description is a summary, it is accurate enough for PSyclone as this information is primarily used to determine which grid partitions must communicate with which for the purposes of placing halo exchange calls. In this case, it is the depth and direction information that is most important.

## **Iterates Over**

The second element of kernel metadata is <code>ITERATES\_OVER</code>. This specifies that the Kernel has been written with the assumption that it is iterating over grid points of the specified type. By default the supported values are: <code>INTERNAL\_PTS</code>, <code>EXTERNAL\_PTS</code> and <code>ALL\_PTS</code>. These may be understood by considering the following diagram of an example model configuration:

10.4. Kernel 93



outside the domain and ALL\_PTS encompasses all grid points in the model. The chosen value is specified in the kernel-meta data like so:

```
integer :: iterates_over = INTERNAL_PTS
```

A user can use a config file (see *Configuration*) to add additional iteration spaces to PSyclone.

#### **Index Offset**

The third element of kernel metadata, INDEX\_OFFSET, specifies the index-offset that the kernel uses. This is the same quantity as supplied to the grid constructor (see the *Grid* Section for a description).

The GOcean 1.0 API supports two different offset schemes; OFFSET\_NE, OFFSET\_SW. The scheme used by a kernel is specified in the metadata as, e.g.:

```
integer :: index_offset = OFFSET_NE
```

Currently all kernels used in an application must use the same offset scheme which must also be the same as passed to the grid constructor.

#### **Procedure**

The fourth and final type of metadata is procedure metadata. This specifies the name of the Kernel Fortran subroutine that this metadata describes.

For example:

```
procedure :: my_kernel_code
```

## 10.4.2 Subroutine

#### **Rules**

Kernel arguments follow a set of rules which have been specified for the GOcean 1.0 API. These rules are encoded in the gen\_code() method of the GOKern class in the gocean1p0.py file. The rules, along with PSyclone's naming conventions, are:

- 1. Every kernel has the indices of the current grid point as the first two arguments, i and j. These are integers and have intent in.
- 2. For each field/scalar/grid property in the order specified by the meta\_args metadata:
  - (a) For a field; the field array itself. A field array is a real array of kind wp and rank two. The first rank is longitude (x) and the second latitude (y).
  - (b) For a scalar; the variable itself. A real scalar is of kind wp.
  - (c) For a grid property; the array or variable (see the earlier table) containing the specified property.

**Note:** Grid properties are not passed from the Algorithm Layer. PSyclone generates the necessary lookups in the PSy Layer and includes the resulting references in the arguments passed to the kernel.

As an example, consider the bc\_solid\_u kernel that is used in the gocean2d program shown earlier. The metadata for this kernel is:

10.4. Kernel 95

The interface to the subroutine containing the implementation of this kernel is:

As described above, the first two arguments to this subroutine specify the grid-point at which the computation is to be performed. The third argument is the field that this kernel updates and the fourth argument is the T-point mask. The latter is a property of the grid and is provided to the kernel call from the PSy Layer.

Comparing this interface definition with the use of the kernel in the Invoke call:

we see that in the Algorithm Layer the user need only provide the field(s) (and possibly scalars) that a kernel operates on. The index of the grid point and any grid properties are provided in the (generated) PSy Layer where the kernel subroutine proper is called.

## 10.5 Built-ins

The GOcean 1.0 API does not support any built-in operations.

## 10.6 Conventions

There is a convention in the GOcean 1.0 API kernel code that if the name of the operation being performed is <name> then a kernel file is <name>\_mod.[fF90], the name of the module inside the kernel file is <name>\_mod, the name of the kernel metadata in the module is <name>\_type and the name of the kernel subroutine in the module is <name>\_code. PSyclone does not require this convention to be followed in the GOcean 1.0 API.

The contents of the metadata is also usually declared private but this does not affect PSyclone.

Finally, the procedure metadata (located within the kernel metadata) usually has nopass specified but again this is ignored by PSyclone.

# 10.7 Configuration

The configuration file (see *Configuration*) used by PSyclone can contain GOcean 1.0 specific options. For example, after the default section the GOcean 1.0 specific section looks like this:

The supported keys are listed in the next section.

## 10.7.1 Iteration-spaces

This section lists additional iteration spaces that can be used in a kernel metadata declaration to allow PSyclone to create a loop with different loop boundaries. Each line of the iteration—spaces declaration contains 7 values, separated by ':'. The fields are:

Field	Description	Details
1	Index Offset	See Index Offset.
2	grid-point types	See Grid point types.
3	Iterates Over	See Iterates Over.
4	Start index of outer loop	Start index of North-South loop.
5	End index of outer loop	End index of North-South loop.
6	Start index of inner loop	Start index of East-West loop.
7	End index of inner loop	End index of East-West loop.

Two special variables can be used in an iteration space: {start} and {stop}. These values will be replaced by PSyclone with the correct loop boundaries for the inner points of a grid (i.e. the non-halo area). This means that the depth-1 halo region can be specified using {start}-1 and {stop}+1.

For example, given the iteration-spaces declaration above, a kernel declared with iterates\_over=internal\_ns\_halo for a field type ct and index offset offset\_sw would create the following loop boundaries:

```
DO j=2-1, jstop+1

DO i=2, istop

CALL (i, j, ...)

END DO

END DO
```

**Warning:** With user defined iteration spaces it is possible that PSyclone will create code that does not compile: if you specify syntactically correct, but semantically incorrect boundary definitions, the PSyclone internal tests will accept the new iteration space, but the compiler will not. For example if one of the loop boundaries contains the name of a variable that is not defined, compilation will fail. It is the responsibility of the user to make sure that valid loop boundaries are specified in a new iteration space definition.

## 10.8 Transformations

In this section we describe the transformations that are specific to the GOcean 1.0 API. For an overview of transformations in general see *Transformations*.

10.7. Configuration 97

#### class psyclone.transformations.GOceanLoopFuseTrans

Performs error checking (that the loops are over the same grid-point type) before calling the LoopFuseTrans.apply() method of the base class in order to fuse two GOcean loops.

## apply (node1, node2)

Fuse the two GOcean loops represented by node1 and node2.

#### **Parameters**

- node1 (psyclone.gocean1p0.GOLoop) A node representing a GOLoop.
- node2 (psyclone.gocean1p0.GOLoop) A node representing a GOLoop.

**Raises** *TransformationError* – if the supplied node2 can not be fused, e.g. not all nodes are loops, don't have the same parent, are not next to each other or have different iteration spaces.

#### name

Returns the name of this transformation as a string.

## class psyclone.transformations.GOceanOMPParallelLoopTrans(omp\_schedule='static')

GOcean specific OpenMP Do loop transformation. Adds GOcean specific validity checks (that supplied Loop is an inner or outer loop). Actual transformation is done by base class.

**Parameters** omp\_schedule - The omp schedule to be created. Must be one of 'runtime', 'static', 'dynamic', 'guided' or 'auto'.

### apply (node)

Perform GOcean-specific loop validity checks then call OMPParallelLoopTrans.apply().

Parameters node (psyclone.psyGen.Loop) - A Loop node from an AST.

**Raises** *TransformationError* – if the supplied node is not an inner or outer loop.

### name

Returns the name of this transformation as a string.

#### class psyclone.transformations.GOceanOMPLoopTrans(omp schedule='static')

GOcean-specific orphan OpenMP loop transformation. Adds GOcean specific validity checks (that the node is either an inner or outer Loop). Actual transformation is done by base class.

**Parameters** omp\_schedule - The omp schedule to be created. Must be one of 'runtime', 'static', 'dynamic', 'guided' or 'auto'.

### apply (node)

Perform GOcean specific loop validity checks then call OMPLoopTrans.apply().

Parameters node (psyclone.psyGen.Loop.) - The loop to parallelise using OMP Do.

### name

Returns the name of this transformation as a string.

#### class psyclone.transformations.GOConstLoopBoundsTrans

Switch on (or off) the use of constant loop bounds within a GOSchedule. In the absence of constant loop bounds, PSyclone will generate loops where the bounds are obtained by de-referencing a field object, e.g.:

DO j =  $my_field$ %grid%internal%ystart,  $my_field$ %grid%internal%ystop

Some compilers are able to produce more efficient code if they are provided with information on the relative trip-counts of the loops within an Invoke. With constant loop bounds switched on, PSyclone generates code like:

```
ny = my_field%grid%simulation_domain%ystop
...
DO j = 1, ny-1
```

In practice, the application of the constant loop bounds looks something like, e.g.:

```
>>> from psyclone.parse import parse
>>> from psyclone.psyGen import PSyFactory
>>> import os
>>> TEST_API = "gocean1.0"
>>> _, info = parse(os.path.join("tests", "test_files", "gocean1p0",
                                 "single_invoke.f90"),
>>>
>>>
                    api=TEST_API)
>>> psy = PSyFactory (TEST_API).create(info)
>>> invoke = psy.invokes.get('invoke_0_compute_cu')
>>> schedule = invoke.schedule
>>>
>>> from psyclone.transformations import GOConstLoopBoundsTrans
>>> clbtrans = GOConstLoopBoundsTrans()
>>>
>>> newsched, _ = clbtrans.apply(schedule)
>>> # or, to turn off const. looop bounds:
>>> # newsched, _ = clbtrans.apply(schedule, const_bounds=False)
>>>
>>> newsched.view()
```

#### apply (node, const\_bounds=True)

Switches constant loop bounds on or off for all loops in the schedule node. Default is 'on'.

### **Parameters**

- node (psyclone.gocean1p0.GOSchedule) The schedule of which all loops will get the constant loop bounds switched on or off.
- const\_bounds (bool) If the constant loop should be used (True) or not (False). Default is True.

#### name

Return the name of the Transformation as a string.

## $\textbf{class} \hspace{0.1cm} \texttt{psyclone.trans} for \texttt{mations.GOLoopSwapTrans}$

Provides a loop-swap transformation, e.g.:

```
DO j=1, m
DO i=1, n
```

## becomes:

```
DO i=1, n
DO j=1, m
```

This transform is used as follows:

10.8. Transformations 99

```
>>> from parse import parse
>>> from psyGen import PSyFactory
>>> ast,invokeInfo=parse("shallow_alg.f90")
>>> psy=PSyFactory("gocean1.0").create(invokeInfo)
>>> schedule=psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>>
>>> from transformations import GOLoopSwapTrans
>>> swap=GOLoopSwapTrans()
>>> new_schedule,memento=swap.apply(schedule.children[0])
>>> new_schedule.view()
```

## apply (outer)

The argument outer must be a loop which has exactly one inner loop. This transform then swaps the outer and inner loop.

Parameters outer (psyclone.psyGen.Loop) - The node representing the outer loop.

**Returns** A tuple consistent of the new schedule, and a Memento.

Raises *TransformationError* — if the supplied node does not allow a loop swap to be done.

#### name

Returns the name of this transformation as a string.

**CHAPTER** 

## **ELEVEN**

## **NEMO API**

In contrast to the other APIs supported by PSyclone, the NEMO API is designed to work with source code that does *not* follow the PSyKAl separation of concerns. Instead, the NEMO source code is treated as if it were a manually written PSy layer with all kernels in-lined. This approach relies upon the NEMO Coding Conventions [nem13] in order to reason about the code being processed. Rather than construct a Schedule for the PSy layer from scratch (as is done for other APIs), the Schedule is constructed by parsing the supplied Fortran code and generating a higher-level representation.

Note: the NEMO API is currently only a prototype. The known issues are listed in *Limitations*.

# 11.1 Algorithm

Since NEMO source is treated as a pre-existing PSy layer, this API does not have the concept of an Algorithm layer.

# 11.2 Constructing the PSylRe

Transformations in PSyclone are applied to an Internal Representation, the "PSyIRe." In contrast to the other APIs where the PSyIRe is constructed from scratch, for NEMO PSyclone must parse the existing Fortran and create a higher-level representation of it. This is done using rules based upon the NEMO Coding Conventions [nem13]. These rules are described in the following sections.

## 11.2.1 Loops

## **Explicit**

PSyclone recognises the following loop types, based on the name of the loop variable:

Loop type	Loop variable
Vertical levels	jk
Latitude	ji
Longitude	jj
Tracer species	jn

PSyclone currently assumes that each of these loop types may be safely parallelised. In practice this will not always be the case (e.g. when performing a tri-diagonal solve) and this implementation will need to be refined.

## **Implicit**

The use of Fortran array notation is encouraged in the NEMO Coding Conventions and is employed throughout the NEMO code base. PSyclone therefore also recognises the loops implied by this notation. The type of loop is inferred from the position of the colon in the array subscripts. Since NEMO uses (*ji,jj,jk*) index ordering (longitude, latitude, levels) this means that loop types are determined according to the following table:

Array index	Loop type	Loop limits
1	Longitude	1, jpi
2	Latitude	1, jpj
3	Vertical levels	1, jpk

Currently PSyclone will convert every implicit loop it encounters in one of these dimensions into an explicit loop over the full sub-domain. This is because it has been found by CMCC and Intel that the Intel compiler is better able to OpenMP-parallelise explicit loops. However, as with all compiler-specific optimisations, this situation may change and will depend on the compiler being used.

If an implicit loop is encountered in an array index other than 1-3 then currently PSyclone will raise an error.

# 11.3 Example

A typical fragment of NEMO source code (taken from the traldf\_iso routine) is shown below:

```
DO jn = 1, kjpt

zdit (1,:,:) = 0._wp ; zdit (jpi,:,:) = 0._wp

zdjt (1,:,:) = 0._wp ; zdjt (jpi,:,:) = 0._wp

DO jk = 1, jpkm1

DO jj = 1, jpjm1

DO ji = 1, fs_jpim1

zdit(ji,jj,jk) = ( ptb(ji+1,jj ,jk,jn) - ptb(ji,jj,jk,jn) ) * umask(ji,jj,

jk)

zdjt(ji,jj,jk) = ( ptb(ji ,jj+1,jk,jn) - ptb(ji,jj,jk,jn) ) * vmask(ji,jj,

jk)

END DO

END DO

END DO

END DO
```

PSyclone uses fparser2 to parse such source code and then generates an internal representation of it:

```
Loop[type='tracers', field_space='None', it_space='None']

Loop[type='levels', field_space='None', it_space='None']

Loop[type='lat', field_space='None', it_space='None']

KernCall[Implicit]

Loop[type='levels', field_space='None', it_space='None']

KernCall[Implicit]

Loop[type='levels', field_space='None', it_space='None']

Loop[type='lat', field_space='None', it_space='None']

KernCall[Implicit]

Loop[type='levels', field_space='None', it_space='None']

Loop[type='levels', field_space='None', it_space='None']

KernCall[Implicit]

Loop[type='levels', field_space='None', it_space='None']

Loop[type='levels', field_space='None', it_space='None']

Loop[type='lat', field_space='None', it_space='None']
```

```
Loop[type='lon',field_space='None',it_space='None']
   KernCall[]
```

# 11.4 Limitations

The NEMO API is currently only a prototype implementation. Here we list the current, known limitations/issues:

- 1. When converting implicit loops into explicit loops, the declaration of the loop variables is repeated (there is an x-failing test for this);
- 2. Scalar variables inside loops are not made private when parallelising using OpenMP;
- 3. All recognised loops (levels, latitude etc.) are assumed to be parallelisable. This will not always be the case (e.g. tridiagonal solve has a loop-carried dependence in the vertical);
- 4. Loops/kernels within CASE statements are not found;
- 5. Labelled do-loops are not handled (i.e. they will probably end up being put inside a code block);
- 6. NemoKern.\_load\_from\_loop() and \_load\_from\_implicit\_loop() both need to be implemented. Currently they do nothing but they should e.g. work out which variables are private to the kernel;
- 7. Loops are currently only permitted to contain one kernel. This restriction will have to be lifted in order to permit loop fusion;
- 8. Array slices with specified bounds (e.g. umask(1:10)) are not yet supported and will raise an exception;
- 9. When generating new variable names, no attempt is made to avoid clashing with variables already present in the NEMO source.
- 10. The psyGen.Node base class now has an \_ast property to hold a pointer into the associated fparser2 AST. However, the psyGen.Kern class already has an \_fp2\_ast property that points to the whole fparser2 AST of the kernel code. This will be rationalised in #241.

11.4. Limitations 103

**CHAPTER** 

## **TWELVE**

# STUB GENERATION

# 12.1 Quick Start

- 1. Use an existing Kernel file or create a Kernel file containing a Kernel module with the required metadata and an empty Kernel subroutine with no arguments.
- 2. Run the following command

```
> genkernelstub <PATH>/my_file.f90
```

## 12.2 Introduction

PSyclone provides a kernel stub generator for the Dynamo0.3 API. The kernel stub generator takes a kernel file as input and outputs the kernel subroutine arguments and declarations. The word "stub" is used to indicate that it is only the subroutine arguments and their declarations that are generated; the subroutine has no content.

The primary reason the stub generator is useful is that it generates the correct Kernel subroutine arguments and declarations for the Dynamo0.3 API as specified by the Kernel metadata. As the number of arguments to Kernel subroutines can become large and the arguments have to follow a particular order, it can become burdensome, and potentially error prone, for the user to have to work out the appropriate argument list if written by hand.

The stub generator can be used when creating a new Kernel. A Kernel can first be written to specify the required metadata and then the generator can be used to create the appropriate (empty) Kernel subroutine. The user can then fill in the content of the subroutine.

The stub generator can also be used to check whether the arguments for an existing Kernel are correct i.e. whether the Kernel subroutine and Kernel metadata are consistent. One example would be where a Kernel is updated resulting in a change to the metadata and subroutine arguments.

The Dynamo0.3 API requires Kernels to conform to a set of rules which determine the required arguments and types for a particular Kernel. These rules are required as the generated PSy layer needs to know exactly how to call a Kernel. These rules are outlined in Section *Rules*.

Therefore PSyclone has been coded with the Dynamo0.3 API rules which are then applied when reading the Kernel metadata to produce the require Kernel call and its arguments in the generated PSy layer. These same rules are used by the Kernel stub generator to produce Kernel subroutine stubs, thereby guaranteeing that Kernel calls from the PSy layer and the associated Kernel subroutines are consistent.

# 12.3 Use

Before using the stub generator, PSyclone must be installed. If you have not already done so, please follow the instructions for setting up PSyclone in Section *Getting Going*.

PSyclone will be installed in a particular location on your machine, which will be referred to as the <PSYCLONEINSTALL> directory. The stub generator script genkernelstub comes with the PSyclone installation. A quick check > which genkernelstub should return the location of the <PSYCLONEINSTALL>/bin directory.

The easiest way to use the stub generator is to run the supplied genkernelstub script.

```
> genkernelstub usage: genkernelstub [-h] [-o OUTFILE] [-api API] [-l] filename genkernelstub: error: too few arguments
```

You can get information about the genkernelstub arguments using -h or --help:

```
genkernelstub -h
usage: genkernelstub [-h] [-o OUTFILE] [-api API] [-l] filename
Create Kernel stub code from Kernel metadata
positional arguments:
 filename
                       Kernel metadata
optional arguments:
 -h, --help
                       show this help message and exit
  -o OUTFILE, --outfile OUTFILE
                       filename of output
 -api API
                       choose a particular api from ['dynamo0.3'], default
                        dynamo0.3
 -1, --limit
                       limit the fortran line length to 132 characters
```

As is indicated when using the -h option, the -api option only accepts dynamo0.3 at the moment and is redundant as this option is also the default. However the number of supported API's is expected to expand in the future.

The  $-\circ$ , or  $-\circ$ utfile option allows the user to specify that the output should be written to a particular file. If  $-\circ$  is not specified then the python print statement is used. Typically the print statement results in the output being printed to the terminal.

The -1, or --1imit option utilises the PSyclone support for wrapping of lines within the 132 character limit (please see the *Line Length* chapter for more details).

## 12.4 Kernels

Any Dynamo 0.3 kernel can be used as input to the stub generator. Example Kernels can be found in the examples/dynamo repository or, for more simple cases, in the tests/test\_files/dynamo0p3 directory. These directories are located in the <PSYCLONEHOME>/src/psyclone directory where <PSYCLONEHOME> refers to the location where you download or clone PSyclone (Getting Going).

In the tests/test\_files/dynamo0p3 directory the majority of examples start with testkern. The exceptions are: simple.f90, ru\_kernel\_mod.f90 and matrix\_vector\_mod.F90. The following test kernels can be used to generate kernel stub code (running stub generation from the <PSYCLONEHOME>/src/psyclone directory):

```
tests/test_files/dynamo0p3/testkern_chi_2.F90
tests/test_files/dynamo0p3/testkern_chi.F90
tests/test_files/dynamo0p3/testkern_operator_mod.f90
tests/test_files/dynamo0p3/testkern_operator_nofield_mod.f90
tests/test_files/dynamo0p3/testkern_orientation.F90
tests/test_files/dynamo0p3/testkern_operator_orient_mod.f90
tests/test_files/dynamo0p3/ru_kernel_mod.f90
tests/test_files/dynamo0p3/simple.f90
```

# 12.5 Example

A simple single field example of a kernel that can be used as input for the stub generator is found in tests/test\_files/dynamo0p3/simple.f90 and is shown below:

**Note:** The module name simple\_mod and the type name simple\_type share the same root simple and have the extensions \_mod and \_type respectively. This is a convention in Dynamo0.3 and is required by the kernel stub generator as it needs to determine the name of the type containing the metadata and infers this by reading the module name. If this rule is not followed the kernel stub generator will return with an error message (see Section *Errors*).

**Note:** Whilst strictly the kernel stub generator only requires the Kernel metadata to generate the appropriate stub code, the parser that the generator relies on currently requires a dummy kernel subroutine to exist.

If we run the kernel stub generator on the simple.f90 example:

```
> genkernelstub tests/test_files/dynamo0p3/simple.f90
```

we get the following kernel stub output:

```
MODULE simple_mod

IMPLICIT NONE

CONTAINS

SUBROUTINE simple_code(nlayers, field_1_w1, ndf_w1, undf_w1, map_w1)

USE constants_mod, ONLY: r_def

IMPLICIT NONE

INTEGER, intent(in) :: nlayers

INTEGER, intent(in) :: ndf_w1

INTEGER, intent(in) :: undf_w1

REAL(KIND=r_def), intent(out), dimension(undf_w1) :: field_1_w1
```

12.5. Example 107

```
INTEGER, intent(in), dimension(ndf_w1) :: map_w1
END SUBROUTINE simple_code
END MODULE simple_mod
```

The subroutine content can then be copied into the required module, used as the basis for a new module, or checked with an existing subroutine for correctness.

Note: The output does not currently conform to Met Office coding standards so must be modified accordingly.

**Note:** The code will not compile without a) providing the constants\_mod, argument\_mod and kernel\_mod modules in the compiler include path and b) adding in code that writes to any arguments declared as intent out or inout. For a quick check, the USE declaration and KIND declarations can be removed and the field\_1\_w1 array can be initialised with some value in the subroutine. At this point the Kernel should compile successfully.

**Note:** Whilst there is only one field declared in the metadata there are 5 arguments to the Kernel. The first argument nlayers specifies the number of layers in a column for a field. The second argument is the array associated with the field. The field array is dimensioned as the number of unique degrees of freedom (undf) which is also passed into the kernel (the fourth argument). The naming convention is to call each field a field, followed by its position in the (algorithm) argument list (which is reflected in the metadata ordering). The third argument is the number of degrees of freedom for the particular column and is used to dimension the final argument which is the degrees of freedom map (dofmap) which indicates the location of the required values in the field array. The naming convention for the dofmap, undf and ndf is to append the name with the space that it is associated with.

We now take a look at a more complicated example. The metadata in this example is the same as an actual Dynamo kernel, however the subroutine content and various comments have been removed. The metadata specifies that there are four fields passed by the algorithm layer, the fourth of which is a vector field of size three. All three of the spaces require a basis function and the w0 and w2 function spaces additionally require a differential basis function. The content of the Kernel is given below.

```
module ru_kernel_mod
type, public, extends(kernel_type) :: ru_kernel_type
 private
 type(arg_type) :: meta_args(6) = (/
      arg_type(GH_FIELD, GH_INC, W2),
                                                                       δ
      arg_type(GH_FIELD, GH_READ, W3),
      arg_type(GH_INTEGER, GH_READ),
      arg_type(GH_REAL, GH_READ),
      arg_type(GH_FIELD, GH_READ, W0),
       arg_type(GH_FIELD*3, GH_READ, W0)
 type(func_type) :: meta_funcs(3) = (/
       func_type(W2, GH_BASIS, GH_DIFF_BASIS),
                                                                       δ
       func_type(W3, GH_BASIS),
                                                                       S.
       func_type(W0, GH_BASIS, GH_DIFF_BASIS)
       /)
 integer :: iterates_over = CELLS
 integer :: gh_shape = gh_quadrature_XYoZ
contains
 procedure, nopass ::ru_code
end type
```

```
contains
subroutine ru_code()
end subroutine ru_code
end module ru_kernel_mod
```

If we run the kernel stub generator on this example:

```
> genkernelstub tests/test_files/dynamo0p3/ru_kernel_mod.f90
```

### we obtain the following output:

```
MODULE ru_mod
 IMPLICIT NONE
 CONTAINS
 SUBROUTINE ru_code(nlayers, field_1_w2, field_2_w3, iscalar_3, rscalar_4, field_5_
→w0, field_6_w0_v1, field_6_w0_v2, field_6_w0_v3, ndf_w2, undf_w2, map_w2, basis_w2,...
→diff_basis_w2, ndf_w3, undf_w3, map_w3, basis_w3, ndf_w0, undf_w0, map_w0, basis_w0,

→ diff_basis_w0, np_xy, np_z, weights_xy, weights_z)
   USE constants_mod, ONLY: r_def
   IMPLICIT NONE
    INTEGER, intent(in) :: nlayers
    INTEGER, intent(in) :: ndf_w2
    INTEGER, intent(in) :: undf_w2
    INTEGER, intent(in) :: ndf w3
   INTEGER, intent(in) :: undf_w3
   INTEGER, intent(in) :: ndf_w0
   INTEGER, intent(in) :: undf_w0
   REAL(KIND=r_def), intent(inout), dimension(undf_w2) :: field_1_w2
   REAL(KIND=r_def), intent(in), dimension(undf_w3) :: field_2_w3
   INTEGER, intent(in) :: iscalar_3
   REAL(KIND=r def), intent(in) :: rscalar 4
   REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_5_w0
   REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_6_w0_v1
   REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_6_w0_v2
   REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_6_w0_v3
    INTEGER, intent(in), dimension(ndf_w2) :: map_w2
   REAL(KIND=r_def), intent(in), dimension(3,ndf_w2,np_xy,np_z) :: basis_w2
   REAL(KIND=r_def), intent(in), dimension(1,ndf_w2,np_xy,np_z) :: diff_basis_w2
    INTEGER, intent(in), dimension(ndf_w3) :: map_w3
   REAL(KIND=r_def), intent(in), dimension(1,ndf_w3,np_xy,np_z) :: basis_w3
   INTEGER, intent(in), dimension(ndf_w0) :: map_w0
   REAL(KIND=r_def), intent(in), dimension(1,ndf_w0,np_xy,np_z) :: basis_w0
   REAL(KIND=r_def), intent(in), dimension(3,ndf_w0,np_xy,np_z) :: diff_basis_w0
   INTEGER, intent(in) :: np_xy, np_z
   REAL(KIND=r_def), intent(in), dimension(np_xy) :: weights_xy
   REAL(KIND=r_def), intent(in), dimension(np_z) :: weights_z
 END SUBROUTINE ru_code
END MODULE ru_mod
```

The above example demonstrates that the argument list can get quite complex. Rather than going through an explanation of each argument you are referred to Section Rules for more details on the rules for argument types and argument ordering. Regarding naming conventions for arguments you can see that the arrays associated with the fields are labelled as 1-6 depending on their position in the metadata. For a vector field, each vector results in a different array. These are distinguished by appending  $\_vx$  where x is the number of the vector.

The introduction of stencil operations on field arguments futher complicates the argument list of a kernel. An example of the use of the stub generator for a kernel that performs stencil operations is provided in examples/dynamo/eg5.

12.5. Example 109

```
> genkernelstub ../../examples/dynamo/eg5/conservative_flux_kernel_mod.F90
```

# 12.6 Errors

The stub generator has been written to provide useful errors if mistakes are found. If you run the generator and it does not produce a useful error - and in particular if it produces a stack trace - please contact the PSyclone developers.

The following tests do not produce stub kernel code either because they are invalid or because they contain functionality that is not supported in the stub generator.

```
tests/test_files/dynamo0p3/matrix_vector_mod.F90
tests/test_files/dynamo0p3/testkern_any_space_1_mod.f90
tests/test_files/dynamo0p3/testkern_any_space_4_mod.f90
tests/test_files/dynamo0p3/testkern_invalid_fortran.F90
tests/test_files/dynamo0p3/testkern_short_name.F90
tests/test_files/dynamo0p3/testkern_no_datatype.F90
tests/test_files/dynamo0p3/testkern.F90
tests/test_files/dynamo0p3/testkern_qr.F90
```

testkern\_invalid\_fortran.F90, testkern\_no\_datatype.F90, testkern\_short\_name.F90, testkern.F90 and matrix\_vector\_mod.F90 are designed to be invalid for PSyclone testing purposes and should produce appropriate errors. For example:

```
> genkernelstub tests/test_files/dynamo0p3/testkern_invalid_fortran.F90 Error: 'Parse Error: Code appears to be invalid Fortran'
```

any\_space is not currently supported in the stub generator so testkern\_any\_space\_1\_mod.f90 and testkern\_any\_space\_4\_mod.f90 should fail with appropriate warnings because of that. For example:

```
> genkernelstub tests/test_files/dynamo0p3/testkern_any_space_1_mod.f90 Error: "Generation Error: Unsupported space for basis function, expecting one of ['w3', 'wtheta', 'w2v', 'w0', 'w1', 'w2', 'w2h', 'any_w2'] but found 'any_space_1'"
```

As noted above, if the Dynamo0.3 naming convention for module and type names is not followed, the stub generator will return with an error message. For example:

```
> genkernelstub tests/test_files/dynamo0p3/testkern.F90
Error: "Parse Error: Error, module name 'testkern' does not have
'_mod' as an extension. This convention is assumed."
```

**CHAPTER** 

# **THIRTEEN**

# LINE LENGTH

By default PSyclone will generate fortran code with no consideration of fortran line length limits. As the line length limit for free-form fortran is 132 characters, the code that is output may be non-conformant.

Line length is not an issue for many compilers as they allow compiler flags to be set which allow lines longer than the fortran standard. However this is not the case for all compilers.

PSyclone therefore supports the wrapping of lines within the 132 character limit. The next two sections discuss how this is done when scripting and when working interactively respectively.

# 13.1 Script

The *psyclone* script provides the -l option to wrap lines. Please see the *Fortran line length* section for more details.

# 13.2 Interactive

When using PSyclone interactively the line lengths of the input algorithm and Kernel files can be checked by setting the parse () function's "line length" argument to "True".

```
>>> from parse import parse
>>> ast, info = parse("argspec.F90", line_length=True)
```

Similarly the "line\_length" argument can be set to "True" if calling the generator.generate() function. This function simply passes this argument on to the parse.parse() function.

```
>>> from psyclone.generator import generate
>>> alg, psy = generate("argspec.F90", line_length=True)
```

Line wrapping is performed as a post processing step, i.e. after the code has been generated. This is done by an instance of the line\_length.FortLineLength class. For example:

```
>>> from psyclone.generator import generate
>>> from line_length import FortLineLength
>>> psy, alg = generate("algspec.f90", line_length=True)
>>> line_length = FortLineLength()
>>> psy_str = line_length.process(str(psy))
>>> print psy_str
>>> alg_str = line_length.process(str(alg))
>>> print alg_str
```

# 13.3 Limitations

The line\_length.FortLineLength class is only partially aware of fortran syntax. This awareness is required so that appropriate continuation characters can be used (for example & at the end of a line and  $! \circ mp$  at the start of a line for OpenMP directives, & at the end of a line for statements and & at the end of a line and & at the beginning of a line for strings).

Whilst statements only require an & at the end of the line when line wrapping with free-form fortran they may optionally also have an & at the beginning of the subsequent line. In contrast, when splitting a string over multiple lines an & is required at both locations. Therefore an instance of the line\_length.FortLineLength class will always add & at the beginning of a continuation line for a statement, in case the line is split within a string.

One known situation that could cause an instance of the <code>line\_length.FortLineLength</code> class to fail is when an inline comment is used at the end of a line to make it longer than the 132 character limit. Whilst PSyclone does not generate such code for the PSy-layer, this might occur in Algorithm-layer code, even if the Algorithm-layer code conforms to the 132 line length limit. The reason for this is that PSyclone's internal parser concatenates lines together, thus a long line correctly split with continuation characters in the Algorithm-layer becomes a line that needs to be split by an instance of the <code>line\_length.FortLineLength</code> class.

**CHAPTER** 

## **FOURTEEN**

**API** 

# 14.1 The generator module

This module provides the PSyclone 'main' routine which is intended to be driven from the bin/psyclone executable script. 'main' takes an algorithm file as input and produces modified algorithm code and generated PSy code. A function, 'generate', is also provided which has the same functionality as 'main' but can be called from within another Python program.

Takes a GungHo algorithm specification as input and outputs the associated generated algorithm and psy codes suitable for compiling with the specified kernel(s) and GungHo infrastructure. Uses the parse.parse() function to parse the algorithm specification, the psyGen.PSy class to generate the PSy code and the algGen.Alg class to generate the modified algorithm code.

#### **Parameters**

- **filename** (str) The file containing the algorithm specification.
- **kernel\_path** (*str*) The directory from which to recursively search for the files containing the kernel source (if different from the location of the algorithm specification).
- **script\_name** (*str*) A script file that can apply optimisations to the PSy layer (can be a path to a file or a filename that relies on the PYTHONPATH to find the module).
- line\_length (bool) A logical flag specifying whether we care about line lengths being longer than 132 characters. If so, the input (algorithm and kernel) code is checked to make sure that it conforms. The default is False.
- **distributed\_memory** (bool) A logical flag specifying whether to generate distributed memory code. The default is set in the config.py file.

**Returns** 2-tuple containing fparser1 ASTs for the algorithm code and the psy code.

**Raises** IOError – if the filename or search path do not exist

For example:

```
>>> from psyclone.generator import generate
>>> alg, psy = generate("algspec.f90")
>>> alg, psy = generate("algspec.f90", kernel_path="src/kernels")
>>> alg, psy = generate("algspec.f90", script_name="optimise.py")
>>> alg, psy = generate("algspec.f90", line_length=True)
>>> alg, psy = generate("algspec.f90", distributed_memory=False)
```

# 14.2 The parse module

Module implementing classes populated by parsing either kernel meta-data or invoke()'s in the Algorithm layer

```
psyclone.parse.parse(alg_filename, api=", invoke_name='invoke', inf_name='inf', kernel_path=", line length=False, distributed memory=None)
```

Takes a GungHo algorithm specification as input and outputs an AST of this specification and an object containing information about the invocation calls in the algorithm specification and any associated kernel implementations.

#### **Parameters**

- alg\_filename (str) The file containing the algorithm specification.
- invoke\_name (str) The expected name of the invocation calls in the algorithm specification
- **inf\_name** (str) The expected module name of any required infrastructure routines.
- **kernel\_path** (*str*) The path to search for kernel source files (if different from the location of the algorithm source).
- line\_length (bool) A logical flag specifying whether we care about line lengths being longer than 132 characters. If so, the input (algorithm and kernel) code is checked to make sure that it conforms and an error raised if not. The default is False.

**Returns** 2-tuple consisting of the fparser1 AST of the Algorithm file and an object holding details of the invokes found.

```
Return type fparser.one.block_statements.BeginSource, psyclone.parse. FileInfo
```

### Raises

- **IOError** if the filename or search path does not exist.
- **ParseError** if there is an error in the parsing.
- RuntimeError if there is an error in the parsing.

## For example:

```
>>> from parse import parse
>>> ast, info = parse("argspec.F90")
```

# 14.3 The transformations module

This module provides the various transformations that can be applied to the schedule associated with an invoke(). There are both general and API-specific transformation classes in this module where the latter typically apply API-specific checks before calling the base class for the actual transformation.

class psyclone.transformations.ACCDataTrans

Adds an OpenACC "enter data" directive to a Schedule. For example:

```
>>> from psyclone.parse import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
```

```
>>>
>>>
    from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> dtrans = t.get_trans_name('ACCDataTrans')
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>>
>>> # Add an enter-data directive
>>> newschedule, _ = dtrans.apply(schedule)
>>> newschedule.view()
```

#### apply (sched)

Adds an OpenACC "enter data" directive to the invoke associated with the supplied Schedule. Any fields accessed by OpenACC kernels within this schedule will be added to this data region in order to ensure they remain on the target device.

**Parameters** sched (sub-class of psyclone.psyGen.Schedule.) – Schedule to which to add an "enter data" directive.

**Returns** Tuple of the modified schedule and a record of the transformation.

Return type (psyclone.psyGen.Schedule, psyclone.undoredo.Memento)

#### Raises

- NotImplementedError for any API other than GOcean 1.0.
- **TransformationError** if passed something that is not a (subclass of) psyclone. psyGen.Schedule.

#### name

**Returns** the name of this transformation.

## Return type str

## class psyclone.transformations.ACCLoopTrans

Adds an OpenACC loop directive to a loop. This directive must be within the scope of some OpenACC Parallel region (at code-generation time).

#### For example:

```
>>> from psyclone.parse import parse, ParseError
>>> from psyclone.psyGen import PSyFactory, GenerationError
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> ltrans = t.get_trans_name('ACCLoopTrans')
>>> rtrans = t.get_trans_name('ACCParallelTrans')
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> new_schedule = schedule
# Apply the OpenACC Loop transformation to *every* loop
# in the schedule
```

```
>>> for child in schedule.children:
>>> newschedule, memento = ltrans.apply(child, reprod=True)
>>> schedule = newschedule
>>>
# Enclose all of these loops within a single OpenACC
# PARALLEL region
>>> rtrans.omp_schedule("dynamic,1")
>>> newschedule, memento = rtrans.apply(schedule.children)
>>>
```

## apply (node, collapse=None, independent=True)

Apply the ACCLoop transformation to the specified node in a Schedule. This node must be a Loop since this transformation corresponds to inserting a directive immediately before a loop, e.g.:

```
!$ACC LOOP

do ...
...
end do
```

At code-generation time (when psyclone.psyGen.ACCLoopDirective.gen\_code() is called), this node must be within (i.e. a child of) a PARALLEL region.

#### **Parameters**

- node (psyclone.psyGen.Loop.) the supplied node to which we will apply the Loop transformation.
- **collapse** (*int*) number of loops to collapse into single iteration space or None.
- **independent** (bool) whether to add the "independent" clause to the directive (not strictly necessary within PARALLEL regions).

Returns (psyclone.psyGen.Schedule, psyclone.undoredo.Memento)

## name

Returns the name of this transformation as a string.

### class psyclone.transformations.ACCParallelTrans

Create an OpenACC parallel region by inserting directives. This parallel region *must* come after an enter-data directive (see *ACCDataTrans*). For example:

```
>>> from psyclone.parse import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> ptrans = t.get_trans_name('ACCParallelTrans')
>>> dtrans = t.get_trans_name('ACCDataTrans')
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>>
>>> # Enclose everything within a single OpenACC PARALLEL region
>>> newschedule, _ = ptrans.apply(schedule.children)
>>> # Add an enter-data directive
```

```
>>> newschedule, _ = dtrans.apply(newschedule)
>>> newschedule.view()
```

#### name

**Returns** The name of this transformation as a string.

Return type str

#### class psyclone.transformations.ACCRoutineTrans

Transform a kernel subroutine by adding a "!\$acc routine" directive (causing it to be compiled for the OpenACC accelerator device). For example:

```
>>> from psyclone.parse import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api)
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.transformations import ACCRoutineTrans
>>> rtrans = ACCRoutineTrans()
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> kern = schedule.children[0].children[0]
>>> # Transform the kernel
>>> newkern, _ = rtrans.apply(kern)
```

### apply (kern)

Modifies the AST of the supplied kernel so that it contains an '!\$acc routine' OpenACC directive.

Parameters kern (psyclone.psyGen.Call) - The kernel object to transform.

**Returns** (transformed kernel, memento of transformation)

Return type 2-tuple of (psyclone.psyGen.Kern, psyclone.undoredo.Memento).

**Raises** *TransformationError* – if we fail to find the subroutine corresponding to the kernel object.

#### name

**Returns** the name of this transformation class.

Return type str

### class psyclone.transformations.ColourTrans

Apply a colouring transformation to a loop (in order to permit a subsequent parallelisation over colours). For example:

```
>>> invoke = ...
>>> schedule = invoke.schedule
>>>
>>> ctrans = ColourTrans()
>>>
>>> # Colour all of the loops
>>> for child in schedule.children:
>>> cschedule, _ = ctrans.apply(child)
>>>
>>> csched.view()
```

### apply (node)

Converts the Loop represented by node into a nested loop where the outer loop is over colours and the inner loop is over cells of that colour. :param node: The loop to transform. :type node: psyclone.psyGen.Loop:returns: Tuple of modified schedule and record of transformation:rtype: (psyclone.psyGen.Schedule, :py:class:`psyclone.undoredo.Memento)

#### name

Returns the name of this transformation as a string.

## class psyclone.transformations.Dynamo0p3ColourTrans

Split a Dynamo 0.3 loop over cells into colours so that it can be parallelised. For example:

```
>>> from psyclone.parse import parse
>>> from psyclone.psyGen import PSyFactory
>>> import transformations
>>> import os
>>> import pytest
>>>
>>> TEST API = "dvnamo0.3"
>>> _,info=parse(os.path.join(os.path.dirname(os.path.abspath(__file__))),
                 "tests", "test_files", "dynamo0p3",
>>>
>>>
                 "4.6_multikernel_invokes.f90"),
                 api=TEST_API)
>>> psy = PSyFactory (TEST_API).create(info)
>>> invoke = psy.invokes.get('invoke_0')
>>> schedule = invoke.schedule
>>> ctrans = Dynamo0p3ColourTrans()
>>> otrans = DynamoOMPParallelLoopTrans()
>>> # Colour all of the loops
>>> for child in schedule.children:
        cschedule, _ = ctrans.apply(child)
>>>
>>>
>>> # Then apply OpenMP to each of the colour loops
>>> schedule = cschedule
>>> for child in schedule.children:
        newsched, _ = otrans.apply(child.children[0])
>>>
>>> newsched.view()
```

Colouring in the Dynamo 0.3 API is subject to the following rules:

- Only kernels with an iteration space of CELLS and which modify a continuous field require colouring. Any other type of loop will cause this transformation to raise an exception.
- A kernel may have at most one field with 'INC' access
- A separate colour map will be required for each field that is coloured (if an invoke contains >1 kernel call)

## apply (node)

Performs Dynamo0.3-specific error checking and then uses the parent class to convert the Loop represented by node into a nested loop where the outer loop is over colours and the inner loop is over cells of that colour.

Parameters node (psyclone.dynamo0p3.DynLoop) - the loop to transform.

**Returns** 2-tuple of new schedule and memento of transform

**Return type** (psyclone.dynamo0p3.DynSchedule, psyclone.undoredo. Memento)

#### name

Returns the name of this transformation as a string.

## class psyclone.transformations.Dynamo0p30MPLoopTrans(omp\_schedule='static')

Dynamo 0.3 specific orphan OpenMP loop transformation. Adds Dynamo-specific validity checks. Actual transformation is done by base class.

#### apply (node, reprod=None)

Perform Dynamo 0.3 specific loop validity checks then call <code>OMPLoopTrans.apply()</code>.

### **Parameters**

- node (psyclone.psyGen.Node) the Node in the Schedule to check
- reprod (bool or None (default, which indicates to use the default from the config file))—if reproducible reductions should be used.

Raises TransformationError – if an OMP loop transform would create incorrect code.

#### name

Returns the name of this transformation as a string.

## class psyclone.transformations.Dynamo0p3RedundantComputationTrans

This transformation allows the user to modify a loop's bounds so that redundant computation will be performed. Redundant computation can result in halo exchanges being modified, new halo exchanges being added or existing halo exchanges being removed.

- This transformation should be performed before any parallelisation transformations (e.g. for OpenMP) to the loop in question and will raise an exception if this is not the case.
- This transformation can not be applied to a loop containing a reduction and will again raise an exception if this is the case.
- This transformation can only be used to add redundant computation to a loop, not to remove it.
- This transformation allows a loop that is already performing redundant computation to be modified, but only if the depth is increased.

### apply (loop, depth=None)

Apply the redundant computation tranformation to the loop loop. This transformation can be applied to loops iterating over 'cells or 'dofs'. if depth is set to a value then the value will be the depth of the field's halo over which redundant computation will be performed. If depth is not set to a value then redundant computation will be performed to the full depth of the field's halo.

#### **Parameters**

- loop (psyclone.psyGen.DynLoop) the loop that we are transforming
- **depth** (*int* or *None*) the depth of the stencil. Defaults to None if a depth is not provided.

#### name

Returns the name of this transformation as a string.

### class psyclone.transformations.DynamoLoopFuseTrans

Performs error checking before calling the <code>apply()</code> method of the <code>base class</code> in order to fuse two Dynamo loops.

## apply (node1, node2, same\_space=False)

Fuse the two Dynamo loops represented by node1 and node2. The optional same\_space flag asserts that an unknown iteration space (i.e. any\_space) matches the other iteration space. This is set at the users own risk.

#### **Parameters**

- node1 (psyclone.dynamo0p3.DynLoop) First Loop to fuse.
- node2 (psyclone.dynamo0p3.DynLoop) Second Loop to fuse.

Returns two-tuple of modified schedule and Memento

Return type psyclone.psyGen.Schedule, psyclone.undoredo.Memento

**Raises** *TransformationError* – if either of the supplied loops contains an inter-grid kernel.

#### name

Returns the name of this transformation as a string.

**class** psyclone.transformations.**DynamoOMPParallelLoopTrans** (*omp\_schedule='static'*)

Dynamo-specific OpenMP loop transformation. Adds Dynamo specific validity checks. Actual transformation is done by the *base class*.

## apply (node)

Perform Dynamo specific loop validity checks then call the apply () method of the base class.

#### name

Returns the name of this transformation as a string.

## class psyclone.transformations.GOConstLoopBoundsTrans

Switch on (or off) the use of constant loop bounds within a GOSchedule. In the absence of constant loop bounds, PSyclone will generate loops where the bounds are obtained by de-referencing a field object, e.g.:

```
DO j = my_field%grid%internal%ystart, my_field%grid%internal%ystop
```

Some compilers are able to produce more efficient code if they are provided with information on the relative trip-counts of the loops within an Invoke. With constant loop bounds switched on, PSyclone generates code like:

```
ny = my_field%grid%simulation_domain%ystop
...
DO j = 1, ny-1
```

In practice, the application of the constant loop bounds looks something like, e.g.:

```
>>> from psyclone.parse import parse
>>> from psyclone.psyGen import PSyFactory
>>> import os
>>> TEST_API = "gocean1.0"
>>> _, info = parse(os.path.join("tests", "test_files", "gocean1p0",
>>>
                                 "single_invoke.f90"),
>>>
                    api=TEST_API)
>>> psy = PSyFactory(TEST_API).create(info)
>>> invoke = psy.invokes.get('invoke_0_compute_cu')
>>> schedule = invoke.schedule
>>> from psyclone.transformations import GOConstLoopBoundsTrans
>>> clbtrans = GOConstLoopBoundsTrans()
>>>
>>> newsched, _ = clbtrans.apply(schedule)
>>> # or, to turn off const. looop bounds:
>>> # newsched, _ = clbtrans.apply(schedule, const_bounds=False)
>>>
>>> newsched.view()
```

#### apply (node, const\_bounds=True)

Switches constant loop bounds on or off for all loops in the schedule node. Default is 'on'.

#### **Parameters**

- node (psyclone.gocean1p0.GOSchedule) The schedule of which all loops will get the constant loop bounds switched on or off.
- const\_bounds (bool) If the constant loop should be used (True) or not (False). Default is True.

#### name

Return the name of the Transformation as a string.

## class psyclone.transformations.GOLoopSwapTrans

Provides a loop-swap transformation, e.g.:

```
DO j=1, m
DO i=1, n
```

#### becomes:

```
DO i=1, n
DO j=1, m
```

This transform is used as follows:

```
>>> from parse import parse
>>> from psyGen import PSyFactory
>>> ast,invokeInfo=parse("shallow_alg.f90")
>>> psy=PSyFactory("gocean1.0").create(invokeInfo)
>>> schedule=psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>>
>>> from transformations import GOLoopSwapTrans
>>> swap=GOLoopSwapTrans()
>>> new_schedule,memento=swap.apply(schedule.children[0])
>>> new_schedule.view()
```

## apply (outer)

The argument outer must be a loop which has exactly one inner loop. This transform then swaps the outer and inner loop.

Parameters outer (psyclone.psyGen.Loop) - The node representing the outer loop.

**Returns** A tuple consistent of the new schedule, and a Memento.

Raises TransformationError — if the supplied node does not allow a loop swap to be done.

#### name

Returns the name of this transformation as a string.

#### class psyclone.transformations.GOceanLoopFuseTrans

Performs error checking (that the loops are over the same grid-point type) before calling the LoopFuseTrans.apply() method of the base class in order to fuse two GOcean loops.

## apply (node1, node2)

Fuse the two GOcean loops represented by node1 and node2.

#### **Parameters**

- node1 (psyclone.gocean1p0.GOLoop) A node representing a GOLoop.
- node2 (psyclone.gocean1p0.GOLoop) A node representing a GOLoop.

**Raises** *TransformationError* – if the supplied node2 can not be fused, e.g. not all nodes are loops, don't have the same parent, are not next to each other or have different iteration spaces.

#### name

Returns the name of this transformation as a string.

class psyclone.transformations.GOceanOMPLoopTrans(omp\_schedule='static')

GOcean-specific orphan OpenMP loop transformation. Adds GOcean specific validity checks (that the node is either an inner or outer Loop). Actual transformation is done by base class.

**Parameters** omp\_schedule - The omp schedule to be created. Must be one of 'runtime', 'static', 'dynamic', 'guided' or 'auto'.

apply (node)

Perform GOcean specific loop validity checks then call OMPLoopTrans.apply().

Parameters node (psyclone.psyGen.Loop.) - The loop to parallelise using OMP Do.

#### name

Returns the name of this transformation as a string.

**class** psyclone.transformations.**GOceanOMPParallelLoopTrans** (*omp\_schedule='static'*) GOcean specific OpenMP Do loop transformation. Adds GOcean specific validity checks (that supplied Loop is an inner or outer loop). Actual transformation is done by base class.

**Parameters** omp\_schedule - The omp schedule to be created. Must be one of 'runtime', 'static', 'dynamic', 'guided' or 'auto'.

apply (node)

Perform GOcean-specific loop validity checks then call <code>OMPParallelLoopTrans.apply()</code>.

Parameters node (psyclone.psyGen.Loop) - A Loop node from an AST.

Raises TransformationError – if the supplied node is not an inner or outer loop.

#### name

Returns the name of this transformation as a string.

class psyclone.transformations.KernelModuleInlineTrans

Switches on, or switches off, the inlining of a Kernel subroutine into the PSy layer module. For example:

```
>>> invoke = ...
>>> schedule = invoke.schedule
>>>
>>> inline_trans = KernelModuleInlineTrans()
>>>
>>> ischedule, _ = inline_trans.apply(schedule.children[0].children[0])
>>> ischedule.view()
```

**Warning:** For this transformation to work correctly, the Kernel subroutine must only use data that is passed in by argument, declared locally or included via use association within the subroutine. Two examples where in-lining will not work correctly are:

1. A variable is declared within the module that contains the Kernel subroutine and is then accessed within that Kernel;

A variable is included via use association at the module level and accessed within the Kernel subroutine

There are currently no checks that these rules are being followed when in-lining so the onus is on the user to ensure correctness.

## apply (node, inline=True)

Checks that the node is of the correct type (a Kernel) then marks the Kernel to be inlined, or not, depending on the value of the inline argument. If the inline argument is not passed the Kernel is marked to be inlined.

#### name

Returns the name of this transformation as a string.

class psyclone.transformations.LoopFuseTrans

Provides a loop-fuse transformation. For example:

# apply (node1, node2)

Fuse the loops represented by node1 and node2.

#### name

Returns the name of this transformation as a string.

## class psyclone.transformations.MoveTrans

Provides a transformation to move a node in the tree. For example:

Nodes may only be moved to a new location with the same parent and must not break any dependencies otherwise an exception is raised.

```
apply (node, location, position='before')
```

Move the node represented by node before location location (which is also a node) by default and after if the optional *position* argument is set to 'after'. An exception is raised if the move is invalid.

#### name

Returns the name of this transformation as a string.

```
class psyclone.transformations.OMPLoopTrans(omp_schedule='static')
```

Adds an orphaned OpenMP directive to a loop. i.e. the directive must be inside the scope of some other OMP Parallel REGION. This condition is tested at code-generation time. The optional 'reprod' argument in the apply method decides whether standard OpenMP reduction support is to be used (which is not reproducible) or whether a manual reproducible reproduction is to be used.

**Parameters** omp\_schedule (str) – The OpenMP schedule to use.

For example:

```
>>> from psyclone.parse import parse, ParseError
>>> from psyclone.psyGen import PSyFactory, GenerationError
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>> print psy.invokes.names
>>>
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> ltrans = t.get_trans_name('OMPLoopTrans')
>>> rtrans = t.get_trans_name('OMPParallelTrans')
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> new_schedule = schedule
# Apply the OpenMP Loop transformation to *every* loop
# in the schedule
>>> for child in schedule.children:
       newschedule, memento = ltrans.apply(child, reprod=True)
        schedule = newschedule
# Enclose all of these loops within a single OpenMP
# PARALLEL region
>>> rtrans.omp_schedule("dynamic,1")
>>> newschedule, memento = rtrans.apply(schedule.children)
>>>
>>>
```

### apply (node, reprod=None)

Apply the OMPLoopTrans transformation to the specified node in a Schedule. This node must be a Loop since this transformation corresponds to wrapping the generated code with directives like so:

```
! $OMP DO

do ...
...
end do
! $OMP END DO
```

At code-generation time (when OMPLoopDirective.gen\_code() is called), this node must be within (i.e. a child of) an OpenMP PARALLEL region.

The optional reprod argument will cause a reproducible reduction to be generated if it is set to True, otherwise the default value (as read from the psyclone.cfg file) will be used. Note, reproducible in this case means obtaining the same results with the same number of OpenMP threads, not for different numbers of

OpenMP threads.

#### **Parameters**

- node (psyclone.psyGen.Node) the supplied node to which we will apply the OMPLoopTrans transformation
- **reprod** (Boolean or None) optional argument to determine whether to generate reproducible OpenMP reductions (True) or not (False). The default value is None which will cause PSyclone to look up a default value

**Returns** (psyclone.psyGen.Schedule, psyclone.undoredo.Memento)

#### name

Returns the name of this transformation as a string.

### omp\_schedule

Returns the OpenMP schedule that will be specified by this transformation. The default schedule is 'static'.

```
class psyclone.transformations.OMPParallelLoopTrans (omp_schedule='static') Adds an OpenMP PARALLEL DO directive to a loop.
```

For example:

```
>>> from psyclone.parse import parse
>>> from psyclone.psyGen import PSyFactory
>>> ast, invokeInfo = parse("dynamo.F90")
>>> psy = PSyFactory("dynamo0.1").create(invokeInfo)
>>> schedule = psy.invokes.get('invoke_v3_kernel_type').schedule
>>> schedule.view()
>>>
>>> from psyclone.transformations import OMPParallelLoopTrans
>>> trans = OMPParallelLoopTrans()
>>> new_schedule, memento = trans.apply(schedule.children[0])
>>> new_schedule.view()
```

#### apply (node)

Apply an OMPParallelLoop Transformation to the supplied node (which must be a Loop). In the generated code this corresponds to wrapping the Loop with directives:

```
!$OMP PARALLEL DO ...

do ...
end do
!$OMP END PARALLEL DO
```

Parameters node (psyclone.f2pygen.DoGen) - the node (loop) to which to apply the transformation.

**Returns** Two-tuple of transformed schedule and a record of the transformation.

```
Return type (psyclone.psyGen.Schedule, :py:class:`psyclone.undoredo.Memento)
```

#### name

Returns the name of this transformation as a string.

```
class psyclone.transformations.OMPParallelTrans
```

Create an OpenMP PARALLEL region by inserting directives. For example:

```
>>> from psyclone.parse import parse, ParseError
>>> from psyclone.psyGen import PSyFactory, GenerationError
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> ltrans = t.get_trans_name('GOceanOMPLoopTrans')
>>> rtrans = t.get_trans_name('OMPParallelTrans')
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>> new_schedule = schedule
>>> # Apply the OpenMP Loop transformation to *every* loop
>>> # in the schedule
>>> for child in schedule.children:
       newschedule, memento = ltrans.apply(child)
>>>
>>>
       schedule = newschedule
>>>
>>> # Enclose all of these loops within a single OpenMP
>>> # PARALLEL region
>>> newschedule, _ = rtrans.apply(schedule.children)
>>> newschedule.view()
```

#### name

**Returns** The name of this transformation as a string.

### Return type str

## class psyclone.transformations.ParallelLoopTrans

Adds an orphaned directive to a loop indicating that it should be parallelised. i.e. the directive must be inside the scope of some other Parallel REGION. This condition is tested at code-generation time.

```
apply (node, collapse=None)
```

Apply the Loop transformation to the specified node in a Schedule. This node must be a Loop since this transformation corresponds to wrapping the generated code with directives, e.g. for OpenMP:

```
! $OMP DO

do ...
end do
! $OMP END DO
```

At code-generation time (when gen\_code() is called), this node must be within (i.e. a child of) a PARAL-LEL region.

### **Parameters**

- node (psyclone.psyGen.Node.) the supplied node to which we will apply the Loop transformation.
- collapse (int) number of loops to collapse into single iteration space or None.

Returns (psyclone.psyGen.Schedule, psyclone.undoredo.Memento)

name

**Returns** the name of this transformation as a string.

Return type str

## class psyclone.transformations.ParallelRegionTrans

Base class for transformations that create a parallel region.

#### apply (nodes)

Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Loops in the schedule within a single parallel region.

Parameters nodes ((list of) psyclone.psyGen.Node.) - a single Node or a list of Nodes.

**Raises** *TransformationError* – if the nodes argument is not of the correct type.

#### name

Returns the name of this transformation as a string.

## class psyclone.transformations.ProfileRegionTrans

Create a profile region around a list of statements. For example:

```
>>> from psyclone.parse import parse, ParseError
>>> from psyclone.psyGen import PSyFactory, GenerationError
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alq.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> p_trans = t.get_trans_name('ProfileRegionTrans')
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>>
>>> # Enclose all children within a single profile region
>>> newschedule, _ = p_trans.apply(schedule.children)
>>> newschedule.view()
```

# apply (nodes)

Apply this transformation to a subset of the nodes within a schedule - i.e. enclose the specified Nodes in the schedule within a single profiler region.

**Parameters nodes** (psyclone.psygen.Node or list of psyclone.psygen.Node.) – Can be a single node or a list of nodes.

#### name

Returns the name of this transformation as a string

#### class psyclone.transformations.RegionTrans

This class is a base class for all transforms that act on list of nodes. It gives access to a \_validate function that makes sure that the nodes in the list are in the same order as in the original AST, no node is duplicated, and that all nodes have the same parent.

## exception psyclone.transformations.TransformationError(value)

Provides a PSyclone-specific error class for errors found during code transformation operations.

```
\verb"psyclone.transformations.check_intergrid" (node)
```

Utility function to check that the supplied node does not have an intergrid kernel as a child.

This is used to ensure that we reject any attempt to apply transformations to loops containing inter-grid kernels. (This restriction will be lifted in Issue #134 and this routine can then be removed.)

# TODO remove this routine once #134 is complete.

Parameters node (psyGen. Node) - the Node in the Schedule to check

Raises TransformationError – if the supplied node has an inter-grid kernel as a child

# 14.4 The psyGen module

This module provides generic support for PSyclone's PSy code optimisation and generation. The classes in this method need to be specialised for a particular API and implementation.

```
class psyclone.psyGen.PSy(invoke_info)
```

Base class to help manage and generate PSy code for a single algorithm file. Takes the invocation information output from the function parse.parse() as its input and stores this in a way suitable for optimisation and code generation.

**Parameters** invoke\_info (FileInfo) – An object containing the required invocation information for code optimisation and generation. Produced by the function parse.parse().

For example:

```
>>> import psyclone
>>> from psyclone.parse import parse
>>> ast, info = parse("argspec.F90")
>>> from psyclone.psyGen import PSyFactory
>>> api = "..."
>>> psy = PSyFactory(api).create(info)
>>> print(psy.gen)
```

### inline (module)

inline all kernel subroutines into the module that are marked for inlining. Avoid inlining the same kernel more than once.

# 14.5 The algGen module

This module provides the Alg class and supporting exception-handling to translate the original algorithm file into one that can be compiled and linked with the generated PSy code.

```
class psyclone.algGen.Alg(ast, psy)
```

Generate a modified algorithm code for a single algorithm specification. Takes the ast of the algorithm specification output from the function parse.parse() and an instance of the psyGen.PSy class as input.

#### **Parameters**

- **ast** (ast) An object containing an ast of the algorithm specification which was produced by the function parse.parse().
- psy (PSy) An object (psyGen.PSy) containing information about the PSy layer.

For example:

```
>>> from parse import parse
>>> ast,info=parse("argspec.F90")
>>> from psyGen import PSy
>>> psy=PSy(info)
>>> from algGen import Alg
```

```
>>> alg=Alg(ast,psy)
>>> print(alg.gen)
```

## gen

Generate modified algorithm code

Return type ast

# 14.6 The line\_length module

Provides support for breaking long fortran lines into smaller ones to allow the code to conform to the maximum line length limits (132 for f90 free format is the default)

```
class psyclone.line_length.FortLineLength(line_length=132)
```

This class take a free format fortran code as a string and line wraps any lines that are larger than the specified line length

## length

returns the maximum allowed line length

## long\_lines(fortran\_in)

returns true if at least one of the lines in the input code is longer than the allowed length. Otherwise returns false

## process (fortran\_in)

takes fortran code as a string as input and output fortran code as a string with any long lines wrapped appropriately

**CHAPTER** 

## **FIFTEEN**

# **PROFILING**

PSyclone has the ability to define regions that can be profiled with various performance measurement tools. The profiling can be enabled automatically using command line parameters like:

```
psyclone --profile kernels ...
```

Or more fine grained by applying a profiling transformation in a transformation script.

PSyclone can be used with a variety of existing profiling tools. It currently supports dl\_timer, and comes with a simple stand-alone timer library. An application needs to be able to find the module files for the profile wrapper, and needs to be linked with the included wrapper library that interfaces between the PSyclone API and the tool-specific API. It is the responsibility of the user to supply the corresponding compiler command line options when building the application that incorporates the PSyclone-generated code.

# 15.1 Profiling API

In order to be used with different profiling tools, PSyclone supports a simple profiling API. For each existing profiling tool a simple interface library needs to be implemented that maps the PSyclone profiling calls to the corresponding call for the profiling tool.

PSyclone utilises 4 profiling calls which are described in the following sub-sections.

# 15.1.1 ProfileInit()

This method needs to be called once to initialise the profiling tool. At this stage this call is not automatically inserted by PSyclone, so it is the responsibility of the user to add the call to an appropriate location in the application:

```
use profile_mod, only : ProfileInit
...
call ProfileInit()
```

The 'appropriate' location might depend on the profiling library used. For example, it might be necessary to invoke this before or after a call to  $\mathtt{MPI\_Init}$  ().

## 15.1.2 ProfileFinalise()

At the end of the program the function ProfileFinalise() must be called. It will make sure that the measurements are printed or written to file correctly, and that the profiling tool is closed correctly. Again at this stage it is necessary to manually insert the call at an appropriate location:

```
use profile_mod, only : ProfileFinalise
...
call ProfileFinalise()
```

And again the appropriate location might depend on the profiling library used (e.g. before or after a call to MPI Finalize()).

# 15.1.3 ProfileStart()/ProfileEnd()

The ProfileStart and ProfileEnd functions define the beginning and end of a region to be measured. In general it is up to the user what exactly a region is, arbitrary code can be enclosed, as long as it is guaranteed that each call of ProfileStart is matched with exactly one corresponding call to ProfileEnd. PSyclone supplies one saved (static) variable of type ProfileData to each matching Start/End pair.

This is the code sequence which is created by PSyclone:

```
use profile_mod, only : ProfileData, ProfileStart, ProfileEnd
...
type(ProfileData), save :: profiler_data
...
call ProfileStart("Module", "Region", profiler_data)
...
call ProfileEnd(profiler_data)
```

PSyclone guarantees that different ProfileStart/End pairs have different ProfileData variables.

# 15.2 Profiling Command Line Options

PSyclone offers two command line options to automatically instrument code with profiling regions. It can create profile regions around a full invoke (including all kernel calls in this invoke), and/or around each individual kernel.

The option —profile invokes will automatically add a call to ProfileStart and ProfileEnd at the beginning and end of every invoke subroutine created by PSyclone. All kernels called within this invoke subroutine will be included in the profiled region.

The option ——profile kernels will add a call to ProfileStart before any loops created by PSyclone, and a ProfileEnd call at the end of the loop. Two caveats:

- 1. In some APIs (for example dynamo when using distributed memory) additional minor code might get included in a profiled kernel section, for example setDirty() calls (expensive calls like HaloExchange are excluded).
- 2. If loop transforms are applied using a script, the profiling nodes added to the AST will very likely cause errors in the script or in the generated output. As example consider a case where an OMPLoop transform is applied to a loop. With profiling enabled instead of the expected loop there could be a profile node in the AST (with the loop as child). Since an OMP DO directive can only have loops inside, and it now has a call to ProfileStart, the generated code is incorrect and will not compile.

In order to avoid the second issue, automatic profiling using <code>--profile</code> is not allowed together with a transformation script. On the other hand, since it is possible to write scripts that are more flexible in handling a modified AST, you can use the command line option <code>--force-profile</code>. It takes the same parameters as <code>--profile</code>, and will allow you to combine a transformation script together with automatic profiling. Use this option at your own risk!

It is also the responsibility of the user to manually add the calls to ProfileInit and ProfileFinalise to the code base.

PSyclone will modify the schedule of each invoke to insert the profiling regions. Below we show an example of a schedule created when instrumenting invokes - all children of a Profile-Node will be part of the profiling region, including all loops created by PSyclone and all kernel calls:

```
GOSchedule[invoke='invoke_1',Constant loop bounds=True]

[Profile]

Loop[type='outer',field_space='cu',it_space='internal_pts']

Loop[type='inner',field_space='cu',it_space='internal_pts']

KernCall compute_unew_code(unew_fld, uold_fld, z_fld, cv_fld, h_fld, tdt,

dy) [module_inline=False]

Loop[type='outer',field_space='cv',it_space='internal_pts']

Loop[type='inner',field_space='cv',it_space='internal_pts']

KernCall compute_vnew_code(vnew_fld,vold_fld,z_fld,cu_fld,h_fld,tdt,

dy) [module_inline=False]

Loop[type='outer',field_space='ct',it_space='internal_pts']

Loop[type='inner',field_space='ct',it_space='internal_pts']

KernCall compute_pnew_code(pnew_fld,pold_fld,cu_fld,cv_fld,tdt,dx,dy)

[module_inline=False]
```

And now the same schedule when instrumenting kernels. In this case each loop nest and kernel call will be contained in a separate region:

```
GOSchedule[invoke='invoke_1', Constant loop bounds=True]
    [Profile]
        Loop[type='outer', field_space='cu', it_space='internal_pts']
            Loop[type='inner', field_space='cu', it_space='internal_pts']
                KernCall compute_unew_code(unew_fld, uold_fld, z_fld, cv_fld, h_fld, tdt,
→dy) [module_inline=False]
    [Profile]
        Loop[type='outer',field_space='cv',it_space='internal_pts']
            Loop[type='inner', field_space='cv', it_space='internal_pts']
                KernCall compute_vnew_code(vnew_fld,vold_fld,z_fld,cu_fld,h_fld,tdt,
→dy) [module_inline=False]
    [Profile]
        Loop[type='outer',field_space='ct',it_space='internal_pts']
            Loop[type='inner',field_space='ct',it_space='internal_pts']
                KernCall compute pnew_code (pnew_fld, pold_fld, cu_fld, cv_fld, tdt, dx, dy)...
→ [module inline=False]
```

Both options can be specified at the same time:

```
GOSchedule[invoke='invoke_1', Constant loop bounds=True]
    [Profile]
        [Profile]
            Loop[type='outer',field_space='cu',it_space='internal_pts']
                Loop[type='inner',field_space='cu',it_space='internal_pts']
                    KernCall compute unew code (unew fld, uold fld, z_fld, cv_fld, h_fld,
→tdt,dy) [module_inline=False]
        [Profile]
            Loop[type='outer', field_space='cv', it_space='internal_pts']
                Loop[type='inner',field_space='cv',it_space='internal_pts']
                    KernCall compute_vnew_code(vnew_fld,vold_fld,z_fld,cu_fld,h_fld,
→tdt,dy) [module_inline=False]
        [Profile]
            Loop[type='outer', field_space='ct', it_space='internal_pts']
                Loop[type='inner',field_space='ct',it_space='internal_pts']
                    KernCall compute_pnew_code(pnew_fld,pold_fld,cu_fld,cv_fld,tdt,dx,
→dy) [module_inline=False]
```

# 15.3 Profiling in Scripts - ProfileRegionTransform

The greatest flexibility is achieved by using the profiler transformation explicitly in a transformation script. The script takes either a single AST Node or a list of AST Nodes as argument, and will insert a Profile Node into the AST, with the specified nodes as children. At code creation time the listed children will all be enclosed in one profile region. As an example:

```
from psyclone.transformations import ProfileRegionTrans

t=TransInfo()
p_trans= ProfileRegionTrans()
schedule=psy.invokes.get('invoke_0').schedule
schedule.view()

# Enclose all children within a single profile region
newschedule, _ = p_trans.apply(schedule.children[1:3])
newschedule.view()
```

**Warning:** It is the responsibility of the user to make sure that a profile region is only created inside a multi-threaded region if the profiling library used is thread-safe!

# 15.4 Interface to Third Party Profiling Tools

PSyclone comes with wrapper libraries to support usage of dl\_timer and a simple non-thread-safe timing library. Support for DrHook and TAU will be added in the future.

Any user can create similar wrapper libraries for other profiling tools by providing a corresponding Fortran module. The four profiling calls described in the section about the *ProfilingAPI* must be implemented, and an opaque, user-defined type ProfileData needs to be provided in the module.

Note that the ProfileEnd call does not have the module or region name as an argument. If this is required by the profiling library, this data must be stored in the ProfileData object so that it is available in the ProfileEnd call.

The examples in the lib/profiling directory show various ways in which the opaque data type can be used to interface with existing profiling tools - for example by storing an index used by the profiling tool in ProfileData, or by storing pointers to the profiling data to be able to print all results in a ProfileFinalise() subroutine.

**CHAPTER** 

# SIXTEEN

# SYSTEM-SPECIFIC SET-UP

This chapter is split into two sections: *User set-up* describes the setup for a user of PSyclone. It includes all steps necessary to be able to use PSyclone. And while you could obviously do some development, none of the required tools for testing or documentation creation will be installed.

Developer set-up describes the additional installation of all required tools to run tests and create documentation.

Both sections have detailed instructions for Ubuntu 16.04.2 and OpenSUSE 42.2 - if you are working with a different Linux distribution some adjustments will be necessary.

# 16.1 User set-up

This section provides system-specific information on how to set-up your system to use PSyclone for users. It has been tested with a vanilla installation of Ubuntu 16.04.2 and OpenSUSE 42.2. You need a terminal window for entering the commands into.

# 16.1.1 Installing dependencies

Most required dependencies are installed from the Python Package Index (https://packaging.python.org/installing/) using the program pip ("PIP Installs Packages"). Besides pip it is also recommended to install the graphviz package to be able to visualise dependency graphs. This is optional and the associated routine will silently return if the graphviz bindings are not installed.

## Installing dependencies on Ubuntu

On Ubuntu pip and graphviz are installed using apt-get. Remember that graphviz is optional and that you'll need to install the graphviz package in addition to the Python bindings.

```
> sudo apt-get install python-pip graphviz
```

## Installing dependencies on OpenSUSE

The vanilla OpenSUSE installation includes pip for Python 3. Note that the graphviz package is installed by default.

```
> sudo zypper install python-pip
```

# 16.1.2 Installing PSyclone

Change your working directory to where you would like to place the code and download the latest stable release of PSyclone.

```
> cd <PSYCLONEHOME>
> wget https://github.com/stfc/PSyclone/archive/1.6.0.tar.gz
> gunzip 1.6.0.tar.gz
> tar xf 1.6.0.tar
> rm 1.6.0.tar
> cd PSyclone-1.6.0
> export PYTHONPATH=`pwd`/src:${PYTHONPATH}
> export PATH=`pwd`/bin:${PATH}
```

This sets up your python path and path appropriately. You may want to set these paths permanently (e.g. by editing your \${HOME}/.bashrc file if you run the BASH shell). You can also use the latest version using git, as described in *Installing PSyclone From Git*.

## 16.1.3 Common installation

To avoid warnings during the dependency installation, it is recommended to update pip to the latest version:

```
> sudo pip install --upgrade pip
```

Next you need to install the fparser and pyparsing packages:

```
> sudo pip install fparser pyparsing
```

**Tip:** With pip it is possible to install packages either system-wide (which requires root privileges) as above, or for a single user only (in ~/.local). While the latter is only useful for one particular user, it means that PSyclone can be installed using pip without needing root privileges. In order to install a package for a user, add the –user command line option to all pip commands. This flag requests that the packages be installed locally for the current user rather than requiring root access:

```
> pip install --user fparser pyparsing
```

You may remove the use of sudo and add the --user option to all pip commands described in this document.

Uninstalling is simply a matter of doing:

```
> sudo pip uninstall fparser pyparsing
```

PSyclone supports the ability to output a schedule dependency graph using the graphviz package. This is optional and the associated routine will silently return if the graphviz bindings are not installed. If you have the graphviz package installed (see especially section *Installing dependencies on Ubuntu* if you are on Ubuntu), you also need to install the python bindings to the graphviz package:

```
> sudo pip install graphviz
```

If you just want to use PSyclone then you've installed all you need and you are ready to go to the getting-going *Run* section.

# 16.2 Developer set-up

This section adds software that is used to develop and test PSyclone. It includes all packages for testing and creation of documentation in html and pdf. We assume you have already installed the software described in the *User set-up* section.

# 16.2.1 Installing PSyclone From Git

For development it is recommended to get a copy of PSyclone using git to get access to the latest development version.

## Installing git for Ubuntu

You need to install the git package:

```
> sudo apt-get install git
```

## Installing git on OpenSUSE

You need to install the git package:

```
>> sudo zypper --no-recommends install git
```

## Cloning PSyclone using git

Cloning PSyclone from git and setting up your environment is done as follows:

```
> cd <PSYCLONEHOME>
> git clone https://github.com/stfc/PSyclone.git
> cd PSyclone
> pip install --user -e .
```

Note that the "-e" flag causes the project to be installed in 'editable' mode so that any changes to the PSyclone source take effect immediately.

**Warning:** On OpenSUSE it is necessary to add \$HOME/.local/bin to your \$PATH if you have done a user-local install.

# 16.2.2 Installing documentation tools

Install Sphinx along with bibtex support for creating PSyclone documentation:

```
> sudo pip install sphinx sphinxcontrib.bibtex
```

You can now build html documentation:

```
> cd doc
> make html
```

The latex package is required to create the pdf documentation for PSyclone. Installing the full dependencies can take up several GB, the instructions for Ubuntu and OpenSUSE only install a minimal subset.

## **Installing LaTeX on Ubuntu**

The following three packages need to be installed to create the pdf documentation. It is recommended to install the packages in one apt-get command, since otherwise, depending on your filesystem, unnecessary snapshots might be created that take up additional space. The --no-install-recommends option significantly reduces the number of installed packages:

```
> sudo apt-get install --no-install-recommends texlive \ texlive-latex-extra latexmk
```

## Installing LaTeX on OpenSUSE

The following command installs the minimum number of packages in order to create the pdf documentation - around 130 packages all in all, requiring approximately 300 MB.

**Warning:** It is important to install the packages in one zypper command, since otherwise, depending on your filesystem, unnecessary snapshots might be created after each package, which can add up to several GB of data.

```
> sudo zypper install --no-recommends texlive-latex texlive-latexmk \
texlive-babel-english texlive-cmap texlive-psnfss texlive-fncychap \
texlive-fancyhdr texlive-titlesec texlive-tabulary texlive-varwidth \
texlive-framed texlive-fancyvrb texlive-float texlive-wrapfig \
texlive-parskip texlive-upquote texlive-capt-of texlive-needspace \
texlive-metafont texlive-makeindex texlive-times texlive-helvetic \
texlive-courier texlive-dvips
```

### Creating pdf documentation

You can now build the pdf documentation using

```
> cd doc
> make latexpdf
```

# 16.2.3 Installing testing tools

The following modules are recommended to get access to testing and formatting tools:

```
> sudo pip install pytest pep8 pylint==1.6.5 pytest-cov pytest-pep8 \ pytest-pylint pytest-flakes pytest-pep257
```

**Warning:** It appears that the 1.7 series of pylint has a bug (at least up to 1.7.2) and does not work properly with PSyclone - it aborts with a "maximum recursion depth exceeded" error message. It is therefore recommended to use version 1.6.5, as specified in the above pip command.

You can now run the PSyclone python tests:

```
> cd PSyclone.git
> py.test
```

In order to see whether the Python code conforms to the pep8 standards, use:

> pep8 code.py

Warning: From release 1.7.1 pep8 returns a runtime user warning to install and use pycodestyle instead.

Verifying the pylint standards is done with:

> pylint code.py

OK, you're all set up.

**CHAPTER** 

# **SEVENTEEN**

# CONFIGURATION

PSyclone reads various run-time configuration options from the psyclone.cfg file. As described in *Configuration*, the default psyclone.cfg configuration file is installed in <python-base-prefix>/share/psyclone/during the installation process. (The original version of this file may be found in the PSyclone/config directory of the PSyclone distribution.)

At execution-time, the user can specify a custom configuration file to be used. This can either be done with the --config command line option, or by specifying the (full path to the) configuration file to use via the PSYCLONE\_CONFIG environment variable. If the specified configuration file is not found then PSyclone will fall back to searching in a list of default locations.

The ordering of these locations depends upon whether PSyclone is being run within a Python virtual environment (such as venv). If no virtual environment is detected then the locations searched, in order, are:

```
1. ${PWD}/.psyclone/
```

- 2. \${HOME}/.local/share/psyclone/
- 3. <python-base-dir>/share/psyclone/

where <python-base-dir> is the path stored in Python's sys.prefix.

If a virtual environment is detected then it is assumed that the share directory will be a part of that environment. In order to maintain isolation of distinct virtual environments this directory is then checked *before* the user's home directory. i.e. the list of locations searched is now:

```
1. ${PWD}/.psyclone/
```

- 2. <python-base-dir>/share/psyclone/
- 3. \${HOME}/.local/share/psyclone/

# 17.1 Options

The configuration file is read by the Python ConfigParser class (https://docs.python.org/3/library/configparser.html) and must be formatted accordingly. It currently consists of a DEFAULT section e.g.:

```
[DEFAULT]

API = dynamo0.3

DEFAULTSTUBAPI = dynamo0.3

DISTRIBUTED_MEMORY = true

REPRODUCIBLE_REDUCTIONS = false

REPROD_PAD_SIZE = 8
```

and a an optional API specific section, for example for dynamo0.3 section:

```
[dynamo0.3]
COMPUTE_ANNEXED_DOFS = false
```

#### or for gocean1.0:

The meaning of the various entries is described in the following sub-sections.

Note that ConfigParser supports various forms of boolean entry including "true/false", "yes/no" and "1/0". See https://docs.python.org/3/library/configparser.html#supported-datatypes for more details.

## 17.1.1 DEFAULT Section

This section contains entries that are, in principle, applicable to all APIs supported by PSyclone.

Entry	Description	
API	The API that PSyclone assumes an Algorithm/Kernl conforms to if no API	
	is specified. Must be one of the APIs supported by PSyclone	
	(gunghoproto, dynamo0.1, dynamo0.3, gocean0.1, gocean1.0). If there is	
	no API specified and there is only one API-specific section in the config	
	file loaded, this API will be used. This value can be overwritten by the	
	command line option '-api'. If there is no API entry in the config file, and	
	'-api' is not specified on the command line, dynamo0.3 is used as default.	
DEFAULTSTUBAPI	The API that the kernel-stub generator assumes by default. Must be one of	
	the stub-APIs supported by PSyclone (dynamo0.3 only at this stage).	
DISTRIBUTED_MEMORY	Whether or not to generate code for distributed-memory parallelism by	
	default. Note that this is currently only supported for the dynamo0.3 API.	
REPRODUCIBLE_REDUCTIONS	Whether or not to generate code for reproducible OpenMP reductions (see	
	Reductions) by default.	
REPROD_PAD_SIZE	If generating code for reproducible OpenMP reductions, this setting	
	controls the amount of padding used between elements of the array in	
	which each thread accumulates its local reduction. (This prevents false	
	sharing of cache lines by different threads.)	

# 17.1.2 dynamo0.3 Section

This section contains configuration options that are only applicable when using the Dynamo 0.3 API.

Entry	Description
COMPUTE_ANNEXED_DOFS	Whether or not to perform redundant computation over annexed dofs in order
	to reduce the number of halo exchanges. See <i>Dof iterators</i> in the Developers'
	guide.

# 17.1.3 gocean1.0 Section

This section contains configuration options that are only applicable when using the Gocean 1.0 API.

Entry	Description
iteration-spaces	This contains definitions of additional iteration spaces used by PSyclone. A detailed
	description can be found in the <i>Configuration</i> section of the GOcean 1.0 chapter.

17.1. Options 143

**CHAPTER** 

# **EIGHTEEN**

# **DEVELOPERS' GUIDE**

# 18.1 Working With PSyclone from GitHub

A PSyclone developer will, by definition, be working with the GitHub PSyclone repository rather than installing a released version from pypi (using e.g. pip install psyclone). This section describes the set-up necessary when using PSyclone in this way. It also describes some of the development practises of the PSyclone project.

### 18.1.1 Installation

Although PSyclone releases always work with a released version of fparser, the same is not always true of other versions (e.g. the HEAD of the master branch). For those versions of PSyclone requiring fparser functionality that is not yet in a release, we use the git submodule feature such that the PSyclone repository always has a link to the correct version of fparser. In order to obtain this version the PSyclone repository must be cloned with the --recursive flag:

```
> git clone --recursive https://github.com/stfc/PSyclone.git
```

Alternatively, if you already have a local clone of the PSyclone github repository then doing:

```
> cd <PSYCLONEHOME>
> git submodule init
> git submodule update
```

will fetch the fparser submodule. (Failure to do this will mean that the <PSYCLONEHOME>/external/fparser directory will be empty.)

Note that after cloning the repository from GitHub, the local copy will be on the master branch. If you are working with some other branch then this must be checked out by doing:

```
> cd <PSYCLONEHOME>
> git checkout <BRANCH_NAME>
```

Once the above steps have been performed, the <PSYCLONEHOME>/external/fparser directory will contain the correct version of the fparser code. This can then be installed using pip:

```
> cd <PSYCLONEHOME>/external/fparser
> pip install --user .
```

Once you have the correct version of fparser installed you are ready to install PSyclone itself. Again, the simplest way of doing this is to use pip:

```
> cd <PSYCLONEHOME>
> pip install -e --user .
```

where -e requests an 'editable' installation so that changes to the PSyclone source are immediately reflected in the installed package. (For alternatives to using pip please see the *Getting Going* section.)

### 18.1.2 Test Suite

The PSyclone test suite is integral to the development process and all new code must be covered (i.e. executed) by one or more tests. As described in *Getting Going*, the test suite is written for use with pytest.

## Coverage

The easiest and most user-friendly way of checking the coverage of any new code is to use CodeCov (https://codecov. io/gh/stfc/PSyclone) which is integrated with GitHub. Coverage for Pull Requests is automatically reported and will appear as a comment on the Pull Request. This comment is then automatically updated whenever new code is pushed to the associated branch.

For checking test coverage on your local machine you will need to install the cov plugin (pip install pytest-cov). You can then request various types of coverage report when running the test suite. e.g. to ask for a terminal report of missed lines for the dynamo0p3 module you would do:

```
> cd <PSYCLONEHOME>
> py.test --cov-report term-missing --cov psyclone.dynamo0p3
```

This will produce output along the lines of:

showing the line numbers which are not covered.

## **Parallel execution**

The size of the test suite is such that running all of it in serial can take many minutes, especially if you have requested a coverage report. It is therefore very helpful to run it in parallel and pytest provides support for this via the xdist plugin (pip install pytest-xdist). Once you have this plugin, the test suite may be run in parallel simply by providing the number of cores to use via the -n flag:

```
> cd <PSYCLONEHOME>
> py.test -n 4
```

Running the test suite in parallel also changes the order in which tests are run which can reveal any problems resulting from tests not being sufficiently isolated from one another.

#### **Gotchas**

Note that pytest will not complain if two tests (within a module) have the same name - it will just silently ignore one of them! The best way of checking for this is to run pylint on any modified test modules. (This needs to be done anyway as one of the requirements of the *Code Review* is that all new code be pylint-clean.)

## **Compilation testing**

The test suite provides support for testing that the code generated by PSyclone is valid Fortran. This is performed by writing the generated code to file and then invoking a Fortran compiler. This testing is not performed by default since it requires a Fortran compiler and significantly increases the time taken to run the test suite. Since configuration of these tests uses a pytest fixture, the tests must be run from the tests directory when requesting that compilation checks be performed:

```
> cd <PSYCLONEHOME>/src/psyclone/tests
> py.test --compile
```

The Gnu Fortran compiler (gfortran) is used by default. If you wish to use a different compiler and/or supply specific flags then these are specified by further command-line flags:

```
> py.test --compile --f90=ifort --f90flags="-03"
```

Note that compilaton testing is currently only supported for the "dynamo0.3" API. Since the code generated by PSyclone makes calls to the LFRic infrastructure, we maintain a stub implementation of the necessary routines in tests/test\_files/dynamo0p3/infrastructure.

# 18.1.3 Continuous Integration

The PSyclone project uses Travis (https://travis-ci.org/stfc/PSyclone) for continuous integration. GitHub triggers Travis to execute the test suite whenever there is a push to the repository. The work performed by Travis is configured by the .travis.yml file in the root directory of the repository. Currently Travis is configured to run the test suite for both Python 2.7 and 3.6. It also runs all of the examples using the check\_examples script in the examples directory.

By default, the Travis configuration uses pip to install the dependencies required by PSyclone before running the test suite. This works well when PSyclone only depends upon released versions of other packages. However, PSyclone relies heavily upon fparser which is also under development. Occasionally it may be that a given branch of PSyclone requires a version of fparser that is not yet released. As described in *Installation*, PSyclone has fparser as a git submodule. In order to configure Travis to use that version of fparser instead of a release, the .travis.yml file must be edited and the line executing the "install\_optional.sh" script (in the before\_install section) must be edited to pass in the "fparser submodule" argument:

```
- ./bin/install_optional.sh fparser_submodule
```

Note that this functionality is only for development purposes. Any release of PSyclone must work with a released version of fparser and therefore the "fparser\_submodule" argument must be removed before making a release.

Given that a run of the test-suite on Travis uses approximately 45 minutes of CPU time, it is good practise to avoid triggering it unnecessarily (e.g. if you know that a certain commit won't pass). This can be achieved by appending "[skip ci]" (without the quotes) to the end of the associated git commit message.

## 18.1.4 Code Review

Before a branch can be merged to master it must pass code review. The guidelines for performing a review (i.e. what is expected from the developer) are available on the GitHub PSyclone wiki pages: https://github.com/stfc/PSyclone/wiki.

# 18.2 Generic Code

PSyclone is designed to be configurable so that new front-ends (called API's) can be built, re-using as much existing code as possible. The generic code is kept in the *psyGen.py* file for psy-code generation.

# 18.2.1 Dependence Analysis

Dependence Analysis in PSyclone produces ordering constraints between instances of the *Argument* class within a PSyIRe.

The *Argument* class is used to specify the data being passed into and out of instances of the *Call* class, *HaloExchange* Class and *GlobalSum* class (and their subclasses).

As an illustration consider the following invoke:

```
invoke( & kernel1(a,b), & kernel2(b,c))
```

where the metadata for *kernel1* specifies that the 2nd argument is written to and the metadata for *kernel2* specifies that the 1st argument is read.

In this case the PSyclone dependence analysis will determine that there is a flow dependence between the second argument of *Kernel1* and the first argument of *Kernel2* (a read after a write).

Information about arguments is aggregated to the PSyIRe node level (*kernel1* and *kernel2* in this case) and then on to the parent *loop* node resulting in a flow dependence (a read after a write) between a loop containing *kernel1* and a loop containing *kernel2*. This dependence is used to ensure that a transformation is not able to move one loop before or after the other in the PSyIRe schedule (as this would cause incorrect results).

Dependence analysis is implemented in PSyclone to support functionality such as adding and removing halo exchanges, parallelisation and moving nodes in a PSyIRe schedule. Dependencies between nodes in a PSyIRe schedule can be viewed as a DAG using the dag() method within the *Node* base class.

#### **DataAccess Class**

The *DataAccess* class is at the core of PSyclone data dependence analysis. It takes an instance of the *Argument* class on initialisation and provides methods to compare this instance with other instances of the *Argument* class. The class is used to determine 2 main things, called *overlap* and *covered*.

### Overlap

*Overlap* specifies whether accesses specified by two instances of the *Argument* class access the same data or not. If they do access the same data their accesses are deemed to *overlap*. The best way to explain the meaning of *overlap* is with an example:

Consider a one dimensional array called A of size 4 (A(4)). If one instance of the Argument class accessed the first two elements of array A and another instance of the Argument class accessed the last two elements of array A then they

would both be accessing array A but their accesses would *not overlap*. However, if one instance of the *Argument* class accessed the first three elements of array A and another instance of the *Argument* class accessed the last two elements of array A then their accesses would *overlap* as they are both accessing element A(3).

Having explained the idea of *overlap* in its general sense, in practice PSyclone currently assumes that *any* two instances of the *Argument* class that access data with the same name will always *overlap* and does no further analysis (apart from halo exchanges and vectors, which are discussed below). The reason for this is that nearly all accesses to data, associated with an instance of the *Argument* class, start at index 1 and end at the number of elements, dofs or some halo depth. The exceptions to this are halo exchanges, which only access the halo and boundary conditions, which only access a subset of the data. However these subset accesses are currently not captured in metadata so PSyclone must assume subset accesses do not exist.

If there is a field vector associated with an instance of an *Argument* class then all of the data in its vector indices are assumed to be accessed when the argument is part of a *Call* or a *GlobalSum*. However, in contrast, a *HaloExchange* only acts on a single index of a field vector. Therefore there is one halo exchange per field vector index. For example:

```
Schedule[invoke='invoke_0_testkern_stencil_vector_type' dm=True]
... HaloExchange[field='f1', type='region', depth=1, check_dirty=True]
... HaloExchange[field='f1', type='region', depth=1, check_dirty=True]
... HaloExchange[field='f1', type='region', depth=1, check_dirty=True]
... Loop[type='',field_space='w0',it_space='cells', upper_bound='cell_halo(1)']
... KernCall testkern_stencil_vector_code(f1,f2) [module_inline=False]
```

In the above PSyIRe schedule, the field fI is a vector field and the Call testkern\_stencil\_vector\_code is assumed to access data in all of the vector components. However, there is a separate HaloExchange for each component. This means that halo exchanges accessing the same field but different components do not overlap, but each halo exchange does overlap with the loop node. The current implementation of the overlaps() method deals with field vectors correctly.

## Coverage

The concept of *coverage* naturally follows from the discussion in the previous section.

Again consider a one dimensional array called A of size 4 (A(4)). If one instance (that we will call the *source*) of the *Argument* class accessed the first 3 elements of array A (i.e. elements 1 to 3) and another instance of the *Argument* class accessed the first two elements of array A then their accesses would *overlap* as they are both accessing elements A(1) and A(2) and elements A(1) and A(2) would be *covered*. However, access A(3) for the *source Argument* class would not yet be *covered*. If a subsequent instance of the *Argument* class accessed the 2nd and 3rd elements of array A then all of the accesses A(1), A(2) and A(3) would now be *covered* so the *source argument* would be deemed to be covered.

In PSyclone the above situation occurs when a vector field is accessed in a kernel and also requires halo exchanges e.g.:

```
Schedule[invoke='invoke_0_testkern_stencil_vector_type' dm=True]

HaloExchange[field='f1', type='region', depth=1, check_dirty=True]

HaloExchange[field='f1', type='region', depth=1, check_dirty=True]

HaloExchange[field='f1', type='region', depth=1, check_dirty=True]

Loop[type='',field_space='w0',it_space='cells', upper_bound='cell_halo(1)']

KernCall testkern_stencil_vector_code(f1,f2) [module_inline=False]
```

In this case the PSyIRe loop node needs to know about all 3 halo exchanges before its access is fully *covered*. This functionality is implemented by passing instances of the *Argument* class to the *DataAccess* class *update\_coverage()* method and testing the access.covered property until it returns *True*.

18.2. Generic Code 149

```
# this example is for a field vector 'f1' of size 3
# f1_index[1,2,3] are halo exchange accesses to vector indices [1,2,3] respectively
access = DataAccess(f1_loop)
access.update_coverage(f1_index1)
result = access.covered # will be False
access.update_coverage(f1_index2)
result = access.covered # will be False
access.update_coverage(f1_index3)
result = access.covered # will be True
access.reset_coverage()
```

Note the *reset\_coverage()* method can be used to reset internal state so the instance can be re-used (but this is not used by PSyclone at the moment).

The way in which halo exchanges are placed means that it is not possible for two halo exchange with the same index to depend on each other in a schedule. As a result an exception is raised if this situation is found.

Notice there is no concept of read or write dependencies here. Read or write dependencies are handled by classes that make use of the DataAccess class i.e. the \_field\_write\_arguments() and \_field\_read\_arguments() methods, both of which are found in the Arguments class.

# 18.3 New APIs

**TBD** 

# 18.4 Existing API's

# 18.4.1 Dynamo0.3

#### Mesh

The Dynamo0.3 API supports meshes that are unstructured in the horizontal and structured in the vertical. This is often thought of as a horizontal 2D unstructured mesh which is extruded into the vertical. The LFRic infrastructure represents this mesh as a list of 2D cells with a scalar value capturing the number of levels in the vertical "column".

#### Cells

The Dynamo0.3 API currently assumes that all kernels which support iterating over cells work internally on a column of cells. This means that PSyclone need only be concerned with iterating over cell-columns in the horizontal. As a result the LFRic infrastructure presents the mesh information to PSyclone as if the mesh were 2-dimensional. From now on this 2D view will be assumed i.e. a cell will actually be a column of cells. The LFRic infrastructure provides a global 2D cell index from 1 to the number of cells.

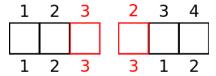
For example, a simple quadrilateral element mesh with 4 cells might be indexed in the following way.

_1	2	3	4

When the distributed memory option is switched on in the Dynamo0.3 API (see the *Distributed Memory* Section) the cells in the model are partitioned amongst processors and halo cells are added at the boundaries to a depth determined

by the LFRic infrastructure. In this case the LFRic infrastructure maintains the global cell index and adds a unique local cell index from 1 to the number of cells in each partition, including any halo cells.

An example for a depth-1 halo implementation with the earlier mesh split into 2 partitions is given below, with the halo cells being coloured red. An example local indexing scheme is also provided below the cells. Notice the local indexing scheme is set up such that owned cells have lower indices than halo cells.



#### **Dofs**

In the LFRic infrastracture the degrees-of-freedom (dofs) are indexed from 1 to the total number of dofs. The infrastructure also indexes dofs so that the values in a column are contiguous and their values increase in the vertical. Thus, given the dof indices for the "bottom" cell, the rest of the dof indices can be determined for the column. This set of dof indices for the bottom cell is called a dofmap.

Dofs represent a field's values at various locations in the mesh. Fields can either be continuous or discontinuous. Continuous fields are so named because their values are continuous across cell boundaries. Dofs that represent continuous fields are shared between neighbouring cells. Discontinuous fields have values that are not necessarily related between neighbouring cells (there can be discontinuities across cell boundaries). Dofs that represent discontinuous fields are local to a cell.

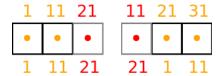
#### **Discontinuous Dofs**

A simple example of discontinuous dofs is given below. In this case each cell contains 1 dof and there are 10 cells in a column. We only show the bottom cells and their corresponding dof indices. As explained earlier, the dof indices increase contiguously up the column, so the cell above the cell containing dof index 1 contains dof index 2 and the cell above that contains dof index 3 etc.

1	11	21	31
•	•	•	•

As discussed in the previous section, when the distributed memory option is switched on in the Dynamo0.3 API (see the *Distributed Memory* Section) the cells in the model are partitioned amongst processors and halo cells are added at the boundaries to a depth determined by the LFRic infrastructure. This results in the dofs being replicated in the halo cells, leading to a dof halo. As for cells, the LFRic infrastructure maintains the global dof indexing scheme and adds a local dof indexing scheme from 1 to the number of dofs in each partition, including any halo dofs.

An example for a depth-1 halo implementation with the earlier mesh split into 2 partitions is given below, with the halo cells drawn in grey and halo dofs coloured red. An example local partition indexing scheme is also provided below the dofs. As with cells, notice the local indexing scheme ensures that owned dofs have lower indices than halo dofs.



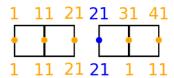
#### **Continuous Dofs**

A simple continuous dof example is given below for the same mesh as before. In this case dofs are on cell edges in the horizontal and there are 10 cells in a column. Again we only show the bottom cells and their corresponding dof indices. As explained earlier, the dof indices increase contiguously up the column, so the cell above the cell containing dof index 1 contains dof index 2 and the cell above that contains dof index 3 etc.

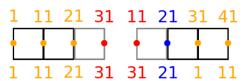


As already explained, when the distributed memory option is switched on in the Dynamo0.3 API (see the *Distributed Memory* Section) the cells in the model are partitioned amongst processors and halo cells are added at the boundaries.

In the example below we ignore the additional halo cells and just look at the partitioning of cells amongst processors (with the same mesh and 2 partitions as shown earlier). It can be seen that the dofs shared between cells which are on different partitions now need to be replicated if fields on continuous dofs are going to be able to be computed locally on each partition. This concept is different to halos as there are no halo cells here, the fact that the cells are partitioned has meant that continuous dofs on the edge of the partition are replicated. The convention used in Dynamo0.3 is that the cell with the lowest global id determines which partition owns a dof and which has the copy. Dofs which are copies are called *annexed*. Annexed dofs are coloured blue in the example:



If we now extend the above example to include the halo cells (coloured grey) then we get:



An example for a depth-1 halo implementation with the earlier mesh split into 2 partitions is given below, with the halo cells drawn in grey and halo dofs coloured red. An example local indexing scheme is also provided below the dofs. Notice the local indexing scheme ensures that owned dofs have lower indices than annexed dofs, which in turn have lower indices than halo dofs.

## **Cell and Dof Ordering**

Cells in a partition are sequentially indexed by the LFRic infrastructure, starting at 1, so that local cells occur first, then level-1 halo cells, then level-2 halo cells etc. A benefit of this layout is that it makes it easy for PSyclone to specify the required iteration space for cells as a single range, allowing a single Fortran do loop (or other language construct as required) to be generated. The LFRic infrastructure provides an API that returns the index of the last owned cell, the index of the last halo cell at a particular depth and the index of the last halo cell, to support PSyclone code generation.

Dofs on a partition are also sequentially indexed by the LFRic infrastructure, starting at 1, so that local dofs occur first, then annexed dofs (if the field is continuous), then level-1 halo dofs, then level-2 halo dofs etc. Again, this layout makes it easy for PSyclone to specify the required iteration space for dofs as a single range. As before, the LFRic infrastructure provides an API that returns the index of the last owned dof, the index of the last annexed dof, the index of the last halo dof at a particular depth and the index of the last halo dof, to support PSyclone code generation.

## **Multi-grid**

The Dynamo 0.3 API supports kernels that map fields between meshes of different horizontal resolutions; these are termed "inter-grid" kernels. As indicated in Fig. 18.1 below, the change in resolution between each level is always a factor of two in both the x and y dimensions.

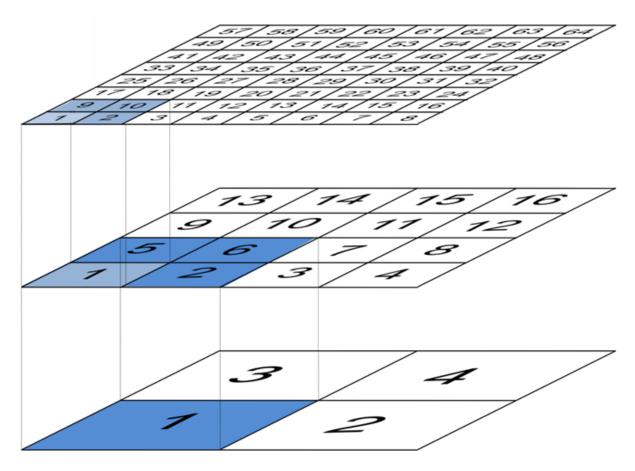


Fig. 18.1: The arrangement of cells in the multi-grid hierarchy used by LFRic. (Courtesy of R. Wong, Met Office.)

Inter-grid kernels are only permitted to deal with fields on two, neighbouring levels of the mesh hierarchy. In the context of a single inter-grid kernel we term the coarser of these meshes the "coarse" mesh and the other the "fine" mesh.

There are two types of inter-grid operation; the first is "prolongation" where a field on a coarse mesh is mapped onto a fine mesh. The second is "restriction" where a field on a fine mesh is mapped onto a coarse mesh. Given the factor of two difference in resolution between the fine and coarse meshes, the depth of any halo accesses for the field on the fine mesh must automatically be double that of those on the coarse mesh.

### **Loop iterators**

In the current implementation of the Dynamo0.3 API it is possible to iterate (loop) either over cells or dofs. At the moment all coded kernels are written to iterate over cells and all builtin kernels are written to iterate over dofs, but that does not have to be the case.

The loop iteration information is specified in the kernel metadata. In the case of builtin's there is kernel metadata but it is part of PSyclone and is specified in *src/psyclone/dynamo0p3\_builtins\_mod.f90*.

18.4. Existing API's 153

For inter-grid kernels, it is the coarse mesh that provides the iteration space. (The kernel is passed a list of the cells in the fine mesh that are associated with the current coarse cell.)

#### **Cell iterators: Continuous**

When a kernel is written to iterate over cells and modify a continuous field, PSyclone always computes dofs on owned cells and redundantly computes dofs in the level-1 halo (or to depth 2 if the field is on the fine mesh of an intergrid kernel - see *Multi-grid*). Users can apply a redundant computation transformation to increase the halo depth for additional redundant computation but it must always at least compute the level-1 halo. The reason for this is to ensure that the shared dofs on cells on the edge of the partition (both owned and annexed) are always correctly computed. Note that the outermost halo dofs are not correctly computed and therefore the outermost halo of the modified field is dirty after redundant computation. Also note that if we do not know whether a modified field is discontinuous or continuous then we must assume it is continuous.

An alternative solution could have been adopted in Dynamo0.3 whereby no redundant computation is performed and partial-sum results are shared between processors in a communication pattern similar to halo exchanges. However, a decision was made to always perform redundant computation.

A downside of performing redundant computation in the level-1 halo is that any fields being read by the kernel must have their level-1 halo clean (up-to-date), which can result in halo exchanges. Note that this is not the case for the modified field, it does not need its halo to be clean, however, at the moment a halo exchange is added in this case. This unecessary halo exchange will be removed in a future release of PSyclone.

#### **Cell iterators: Discontinuous**

When a kernel is written to iterate over cells and modify a discontinuous field, PSyclone only needs to compute dofs on owned cells. Users can apply a redundant computation transformation to redundantly compute into the halo but this is not done by default.

### **Dof iterators**

When a kernel that is written to iterate over dofs modifies a field, PSyclone must ensure that all dofs in that field are updated. If the distributed memory flag is set to false then PSyclone must iterate over all dofs. PSyclone simply needs to create a loop that iterates from 1 to the total number of dofs. The latter value is provided by the LFRic API.

If the distributed memory flag is set to true then PSyclone must ensure that each partition only iterates over owned dofs. Again PSyclone just needs to create a loop that iterates from 1 to the total number of owned dofs on that partition. The latter value is provided by the LFRic API.

When the distributed memory flag is set to true an aditional configuration option can be set which makes PSyclone always create loops which iterate over both owned and annexed dofs. Whilst this is not necessary for correctness, it can improve performance by reducing the number of halo exchanges required (at the expense of computing annexed dofs redundantly). The only change for PSyclone is that it calls a different LFRic routine which returns the index of the last annexed dof. This iteration space will necessarily also include all owned dofs due to the ordering of dof indices discussed earlier.

The configuration variable is called *COMPUTE\_ANNEXED\_DOFS* and is found in the the *dynamo0.3* section of the *psyclone.cfg* configuration file (see *Configuration*). If it is true then annexed dofs are always computed in loops that iterate over dofs and if it is false then annexed dofs are not computed. The default in PSyclone is false.

The computation of annexed dofs could have been added as a transformation optimisation. The reason for using a configuration switch is that it is then guaranteed that annexed dofs are always computed for loops that iterate over dofs which then allows us to always remove certain halo exchanges without needing to add any new ones.

If we first take the situation where annexed dofs are not computed for loops that iterate over dofs i.e. (COM-PUTE\_ANNEXED\_DOFS is false), then a field's annexed dofs will be dirty (out-of-date) after the loop has completed. If a following kernel needs to read the field's annexed dofs, then PSyclone will need to add a halo exchange to make them clean.

There are 3 cases to consider:

- 1. the field is read in a loop that iterates over dofs,
- 2. the field is read in a loop that iterates over owned cells and level-1 halo cells, and
- 3. the field is read in a loop that iterates over owned cells

In case 1) the annexed dofs will not be read as the loop only iterates over owned dofs so a halo exchange is not required. In case 2) the full level-1 halo will be read (including annexed dofs) so a halo exchange is required. In case 3) the annexed dofs will be read so a halo exchange will be required.

If we now take the case where annexed dofs are computed for loops that iterate over dofs (*COM-PUTE\_ANNEXED\_DOFS* is true) then a field's annexed dofs will be clean after the loop has completed. If a following kernel needs to read the field's annexed dofs, then PSyclone will no longer need a halo exchange.

We can now guarantee that annexed dofs will always be clean after a continuous field has been modified by a kernel. This is because loops that iterate over either dofs or cells now compute annexed dofs and there are no other ways for a continuous field to be updated.

We now consider the same three cases. In case 1) the annexed dofs will now be read, but annexed dofs are guaranteed to be clean, so no halo exchange is required. In case 2) the full level-1 halo is read so a halo exchange is still required. Note, as part of this halo exchange we will update annexed dofs that are already clean. In case 3) the annexed dofs will be read but a halo exchange is not required as the annexed dofs are guaranteed to be clean.

Therefore no additional halo exchanges are required when *COMPUTE\_ANNEXED\_DOFS* is changed from false to true i.e. case 1) does not require a halo exchange in either situation and case 2) requires a halo exchange in both situations. We also remove halo exchanges for case 3) so the number of halo exchanges may be reduced.

If a switch were not used and it were possible to use a transformation to selectively perform computation over annexed dofs for loops that iterate over dofs, then we would no longer be able to guarantee that annexed dofs would always be clean. In this situation, if the dofs were known to be dirty then PSyclone would need to add a halo exchange and if it were unknown whether the dofs were dirty or not, then a halo exchange would need to be added that uses the run-time flags to determine whether a halo exchange is required. As run-time flags are based on whether the halo is dirty or not (not annexed dofs) then a halo exchange would be performed if the halo were dirty, even if the annexed dofs were clean, potentially resulting in more halo exchanges than are necessary.

### **Halo Exchange Logic**

Halo exchanges are required when the *DISTRIBUTED\_MEMORY* flag is set to true in order to make sure any accesses to a field's halo or to its annexed dofs receive the correct value.

### **Operators and Halo Exchanges**

Halo Exchanges are only created for fields. This causes an issue for operators. If a loop iterates over halos to a given depth and the loop includes a kernel that reads from an operator then the operator must have valid values in the halos to that depth. In the current implementation of PSyclone all loops which write to, or update an operator are computed redundantly in the halo up to depth-1 (see the *load()* method in the *DynLoop* class). This implementation therefore requires a check that any loop which includes a kernel that reads from an operator is limited to iterating in the halo up to depth-1. PSyclone will raise an exception if an optimisation attempts to increase the iteration space beyond this (see the *gen\_code()* method in the *DynKern* class).

18.4. Existing API's 155

To alleviate the above restriction one could add a configurable depth with which to compute operators e.g. operators are always computed up to depth-2, or perhaps up to the maximum halo depth. An alternative would be to halo exchange operators as required in the same way that halo exchanges are used for fields.

#### **First Creation**

When first run, PSyclone creates a separate schedule for each of the invokes found in the algorithm layer. A schedule includes all required loops and kernel calls that need to be generated in the PSy layer for the particular invoke call. Once the loops and kernel calls have been created then (if the *DISTRIBUTED\_MEMORY* flag is set to true) PSyclone adds any required halo exchanges and global sums. This work is all performed in the *DynInvoke* constructor (\_\_init\_\_) method.

In PSyclone we apply a lazy halo exchange approach (as opposed to an eager one), adding a halo exchange just before it is required.

It is simple to determine where halo exchanges should be added for the initial schedule. There are three cases:

- 1. loops that iterate over cells and modify a continuous field will access the level-1 halo. This means that any field that is read within such a loop must have its level-1 halo clean and therefore requires a halo exchange. Note, at the moment PSyclone adds a halo exchange for the modified field (as it is specified as *GH\_INC* which requires a read before a write), however this is not required if there is only one field updated in the kernel. This is because we only care about updating owned and annexed dofs, therefore it does not matter what the values of any halo dofs are.
- 2. continuous fields that are read from within loops that iterate over cells and modify a discontinuous field must have their annexed dofs clean. Currently the only way to make annexed dofs clean is to perform a halo exchange. If the COMPUTE\_ANNEXED\_DOFS configuration variable is set to true then no halo exchange is required as annexed dofs will always be clean. If the COMPUTE\_ANNEXED\_DOFS configuration variable is set to false then a halo exchange must be added if the previous modification of the field is known to be from within a loop over dofs, or if the previous modification of the field is unknown (i.e. outside the invoke) as the previous modification may have been from within a loop over dofs.
- 3. fields that have a stencil access will access the halo and need halo exchange calls added.

Halo exchanges are created separately (for fields with halo reads) for each loop by calling the *create\_halo\_exchanges()* method within the *DynLoop* class.

In the situation where a field's halo is read in more than one kernel in different loops, we do not want to add too many halo exchanges - one will be enough as long as it is placed correctly. To avoid this problem we add halo exchange calls for loops in the order in which they occur in the schedule. A halo exchange will be added before the first loop for a field but the same field in the second loop will find that there is a dependence on the previously inserted halo exchange so no additional halo exchange will be added.

The algorithm for adding the necessary halo exchanges is as follows: For each loop in the schedule, the <code>create\_halo\_exchanges()</code> method iterates over each field that reads from its halo (determined by the <code>unique\_fields\_with\_halo\_reads()</code> method in the <code>DynLoop</code> class).

For each field we then look for its previous dependencies (the previous writer(s) to that field) using PSyclone's dependence analysis. Three cases can occur: 1) there is no dependence, 2) there are multiple dependencies and 3) there is one dependence.

1. If no previous dependence is found then we add a halo exchange call before the loop (using the internal helper method \_add\_halo\_exchange()). If the field is a vector field then a halo exchange is added for each component. The internal helper method \_add\_halo\_exchange itself uses the internal helper method \_add\_halo\_exchange class for the field in question and adds it to the schedule before the loop. You might notice that this method then checks that the halo exchange is actually required and removes it again if not. In our current situation the halo exchange will always

be needed so this check is not required but in more complex situations after transformations have been applied to the schedule this may not be the case. We discuss this type of situation later.

- 2. If multiple previous dependencies are found then the field must be a vector field as this is the only case where this can occur. We then choose the closest one and treat it as a single previous dependency (see 3).
- 3. If a single previous dependency is found and it is a halo exchange then we do nothing, as it is already covered. This will only happen when more than one reader depends on a writer, as discussed earlier. If the dependence is not a halo exchange then we add one.

After completing the above we have all the halo exchanges required for correct execution.

Note that we do not need to worry about halo depth or whether a halo is definitely required, or whether it might be required, as this is determined by the halo exchange itself at code generation time. The reason for deferring this information is that it can change as transformations are added.

## Modifying the Schedule

Transformations modify the schedule. At the moment only one of these transformations - the *Dynamo0p3RedundantComputationTrans* class in *transformations.py* - affects halo exchanges. This transformation can mean there is a requirement for new halo exchanges, it can mean existing halo exchanges are no longer required and it can mean that the properties of a halo exchange (e.g. depth) can change.

The redundant computation transformation is applied to a loop in a schedule. When this is done the *update\_halo\_exchanges()* method for that loop is called - see the *apply()* method in *Dynamo0p3RedundantComputationTrans*.

The first thing that the *update\_halo\_exchanges()* method does is call the *create\_halo\_exchanges()* method to add in any new halo exchanges that are required before this loop, due to any fields that now have a read access to their halo when they previously did not. For example, a loop containing a kernel that writes to a certain field might previously have iterated up to the number of owned cells in a partition (*ncells*) but now iterates up to halo depth 1.

However, a field that has its halo read no longer guarantees that a halo exchange is required, as the previous dependence may now compute redundantly to halo depth 2, for example. The solution employed in *create\_halo\_exchanges()* is to add a halo exchange speculatively and then remove it if it is not required. The halo exchange itself determines whether it is required or not via the *required()* method. The removal code is found at the end of the *\_add\_halo\_exchange\_code()* method in the *DynLoop()* class.

The second thing that the *update\_halo\_exchanges()* method does is check that any halo exchanges after this loop are still required. It finds all relevant halo exchanges, asks them if they are required and if they are not it removes them.

We only need to consider adding halo exchanges before the loop and removing halo exchanges after the loop. This is because redundant computation can only increase the depth of halo to which a loop computes so can not remove existing halo exchanges before a loop (as an increase in depth will only increase the depth of an existing halo exchange before the loop) or add existing halo exchanges after a loop (as an increase in depth will only make it more likely that a halo exchange is no longer required after the loop).

### Colouring

If a loop contains one or more kernels that write to a field on a continuous function space then it cannot be safely executed in parallel on a shared-memory device. This is because fields on a continuous function space share dofs between neighbouring cells. One solution to this is to 'colour' the cells in a mesh so that all cells of a given colour may be safely updated in parallel (Fig. 18.2).

The loop over colours must then be performed sequentially but the loop over cells of a given colour may be done in parallel. A loop that requires colouring may be transformed using the <code>DynamoOp3ColourTrans</code> transformation.

18.4. Existing API's 157

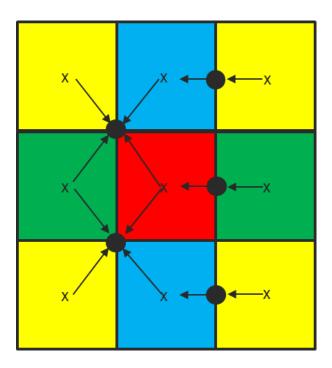


Fig. 18.2: Example of the colouring of the horizontal cells used to ensure the thread-safe update of shared dofs (black circles). (Courtesy of S. Mullerworth, Met Office.)

Each mesh in the multi-grid hierarchy is coloured separately (https://code.metoffice.gov.uk/trac/lfric/wiki/LFRicInfrastructure/MeshColouring) and therefore we cannot assume any relationship between the colour maps of meshes of differing resolution.

However, the iteration space for inter-grid kernels (that map a field from one mesh to another) is always determined by the coarser of the two meshes. Consequently, it is always the colouring of this mesh that must be used. Due to the set-up of the mesh hierarchy (see Fig. 18.1), this guarantees that there will not be any race conditions when updating shared quantities on either the fine or coarse mesh.

## 18.4.2 GOcean1.0

**TBD** 

# 18.5 Modules

This section describes the functionality of the various Python modules that make up PSyclone.

# 18.5.1 Module: f2pygen

f2pygen provides functionality for generating Fortran code from scratch (i.e. when not modifying existing source).

### **Variable Declarations**

Three different classes are provided to support the creation of variable declarations (for intrinsic, character and derived-type variables). An example of their use might be:

```
>>> from psyclone.f2pygen import ModuleGen, SubroutineGen, DeclGen,
CharDeclGen, TypeDeclGen
>>> module = ModuleGen(name="testmodule")
>>> sub = SubroutineGen(module, name="testsubroutine")
>>> module.add(sub)
>>> sub.add(DeclGen(sub, datatype="integer", entity_decls=["my_int"]))
>>> sub.add(CharDeclGen(sub, length="10", entity_decls=["my_char"]))
>>> sub.add(TypeDeclGen(sub, datatype="field_type", entity_decls=["ufld"]))
>>> gen = str(module.root)
>>> print(gen)
 MODULE testmodule
   IMPLICIT NONE
   CONTAINS
    SUBROUTINE testsubroutine()
      TYPE (field_type) ufld
     CHARACTER (LEN=10) my_char
     INTEGER my_int
   END SUBROUTINE testsubroutine
 END MODULE testmodule
```

The full interface to each of these classes is detailed below:

```
class psyclone.f2pygen.DeclGen (parent, datatype=", entity_decls=None, intent=", pointer=False, kind=", dimension=", allocatable=False, save=False, target=False, initial_values=None)
```

Generates a Fortran declaration for variables of various intrinsic types (integer, real and logical). For character variables CharDeclGen should be used.

#### **Parameters**

- parent (psyclone.f2pygen.BaseGen.) node to which to add this declaration as a child.
- datatype (str) the (intrinsic) type for this declaration.
- entity decls (list) list of variable names to declare.
- **intent** (*str*) the INTENT attribute of this declaration.
- **pointer** (bool) whether or not this is a pointer declaration.
- **kind** (str) the KIND attribute to use for this declaration.
- dimension (str) the DIMENSION specifier (i.e. the xx in DIMENSION(xx)).
- allocatable (bool) whether this declaration is for an ALLOCATABLE quantity.
- **save** (bool) whether this declaration has the SAVE attribute.
- target (bool) whether this declaration has the TARGET attribute.
- initial\_values (list of str with same no. of elements as entity\_decls.) Initial value to give each variable.

Raises RuntimeError – if datatype is not one of DeclGen.SUPPORTED\_TYPES.

Generates a Fortran declaration for character variables.

#### **Parameters**

18.5. Modules 159

- parent (psyclone.f2pygen.BaseGen.) node to which to add this declaration as a child.
- **entity\_decls** (list) list of variable names to declare.
- **intent** (*str*) the INTENT attribute of this declaration.
- **pointer** (bool) whether or not this is a pointer declaration.
- **kind** (str) the KIND attribute to use for this declaration.
- **dimension** (str) the DIMENSION specifier (i.e. the xx in DIMENSION(xx)).
- allocatable (bool) whether this declaration is for an ALLOCATABLE quantity.
- **save** (bool) whether this declaration has the SAVE attribute.
- target (bool) whether this declaration has the TARGET attribute.
- length (str) expression to use for the (len=xx) selector.
- initial\_values (list of str with same no. of elements as entity\_decls. Each of these can be either a variable name or a literal, quoted string (e.g. "'hello'")) Initial value to give each variable.

```
class psyclone.f2pygen.TypeDeclGen (parent, datatype=", entity_decls=None, intent=", pointer=False, dimension=", allocatable=False, save=False, target=False)
```

Generates a Fortran declaration for variables of a derived type.

#### **Parameters**

- parent (psyclone.f2pygen.BaseGen) node to which to add this declaration as a child
- datatype (str) the type for this declaration
- entity\_decls (list) list of variable names to declare
- **intent** (*str*) the INTENT attribute of this declaration
- pointer (bool) whether or not this is a pointer declaration
- **dimension** (str) the DIMENSION specifier (i.e. the xx in DIMENSION(xx))
- allocatable (bool) whether this declaration is for an ALLOCATABLE quantity
- **save** (bool) whether this declaration has the SAVE attribute
- target (bool) whether this declaration has the TARGET attribute

## 18.5.2 Module: configuration

PSyclone uses the Python ConfigParser class (https://docs.python.org/3/library/configparser.html) for reading the configuration file. This is managed by the psyclone.configuration module which provides a Config class. This class is a singleton, which can be (created and) accessed using Config.get (). Only one such instance will ever exist:

```
class psyclone.configuration.Config
```

Handles all configuration management. It is implemented as a singleton using a class \_instance variable and a get() function.

#### api

Getter for the API selected by the user.

**Returns** The name of the selected API.

Return type str

## $\mathtt{api\_conf}(api)$

Getter for the object holding API-specific configuration options.

**Parameters** api (str) – the API for which configuration details are required.

Returns object containing API-specific configuration

**Return type** One of psyclone.configuration.DynConfig, psyclone. configuration.GOceanConfig or None.

#### Raises

- ConfigurationError if api is not in the list of supported APIs.
- **ConfigurationError** if the config file did not contain a section for the requested API.

#### default\_api

Getter for the default API used by PSyclone.

**Returns** default PSyclone API

Return type str

#### default\_stub\_api

Getter for the default API used by the stub generator.

**Returns** default API for the stub generator

Return type str

## distributed\_memory

Getter for whether or not distributed memory is enabled

**Returns** True if DM is enabled, False otherwise

Return type bool

#### filename

Getter for the full path and name of the configuration file used to initialise this configuration object.

**Returns** full path and name of configuration file

Return type str

#### static find file()

Static method that searches various locations for a configuration file. If the full path to an existing file has been provided in the PSYCLONE\_CONFIG environment variable then that is returned. Otherwise, we search the following locations, in order:

- \${PWD}/.psyclone/
- \${HOME}/.local/share/psyclone/
- <system-install-prefix>/share/psyclone/

**Returns** the fully-qualified path to the configuration file

Return type str

Raises ConfigurationError – if no config file is found

18.5. Modules 161

```
static get(do not load file=False)
```

Static function that if necessary creates and returns the singleton config instance.

**Parameters** do\_not\_load\_file (bool) – If set it will not load the default config file. This is used when handling the command line so that the user can specify the file to load.

### get\_default\_keys()

Returns all keys from the default section. :returns list: List of all keys of the default section as strings.

```
load(config file=None)
```

Loads a configuration file.

**Parameters** config\_file (str) – Override default configuration file to read.

**Raises** ConfigurationError – if there are errors or inconsistencies in the specified config file.

# reprod\_pad\_size

Getter for the amount of padding to use for the array required for reproducible OpenMP reductions

**Returns** padding size (no. of array elements)

Return type int

## reproducible\_reductions

Getter for whether reproducible reductions are enabled.

**Returns** True if reproducible reductions are enabled, False otherwise.

Return type bool

### supported\_apis

Getter for the list of APIs supported by PSyclone.

**Returns** list of supported APIs

**Return type** list of str

## supported\_stub\_apis

Getter for the list of APIs supported by the stub generator.

Returns list of supported APIs.

Return type list of str

The Config class is responsible for finding the configuration file (if no filename is passed to the constructor), parsing it and then storing the various configuration options. It also stores the list of supported APIs (Config.\_supported\_api\_list) and the default API to use if none is specified in either a config file or the command line (Config.\_default\_api.)

It also performs some basic consistency checks on the values it obtains from the configuration file.

Since the PSyclone API to use can be read from the configuration file, it is not possible to have API-specifc sub-classes of Config as we don't know which API is in use before we read the file. However, the configuration file can contain API-specific settings. These are placed in separate sections, named for the API to which they apply, e.g.:

```
[dynamo0.3]
COMPUTE_ANNEXED_DOFS = false
```

Having parsed and stored the options from the default section of the configuration file, the Config constructor then creates a dictionary using the list of supported APIs to provide the keys. The configuration file is then checked for API-specific sections (again using the API names from the default section) and, if any are found, an API-specific subclass is created using the parsed entries from the corresponding section. The resulting object is stored in the dictionary under the appropriate key. The API-specific values may then be accessed as, e.g.:

```
Config.get().api_conf("dynamo0.3").compute_annexed_dofs
```

The API-specific sub-classes exist to provide validation/type-checking and encapsulation for API-specific options. They do not sub-class Config directly but store a reference back to the Config object to which they belong.

#### 18.5.3 Module: transformations

As one might expect, the transformations module holds the various transformation classes that may be used to modify the Schedule of an Invoke and/or the kernels called from within it.

The base class for any transformation must be the class Transformation:

```
class psyclone.psyGen.Transformation
```

Abstract baseclass for a transformation. Uses the abc module so it can not be instantiated.

```
_validate(*args)
```

Method that validates that the input data is correct. It will raise exceptions if the input data is incorrect. This function needs to be implemented by each transformation.

**Parameters args** (*Type depends on actual transformation*.) – Arguments for the applying the transformation - specific to the actual transform used.

```
apply (*args)
```

Abstract method that applies the transformation. This function must be implemented by each transform.

**Parameters args** (Type depends on actual transformation.) – Arguments for the transformation - specific to the actual transform used.

**Returns** A tuple of the new schedule, and a momento.

Return type Tuple.

#### name

Returns the name of the transformation.

Those transformations that work on a region of code (e.g. enclosing multiple kernel calls within an OpenMP region) must sub-class the RegionTrans class:

```
class psyclone.transformations.RegionTrans
```

This class is a base class for all transforms that act on list of nodes. It gives access to a \_validate function that makes sure that the nodes in the list are in the same order as in the original AST, no node is duplicated, and that all nodes have the same parent.

```
_validate(node_list)
```

Test if the nodes in node\_list are in the original order.

```
Parameters node_list (list) - List of nodes.
```

**Raises** *TransformationError* – If the nodes in the list are not in the original order in which they are in the AST, a node is duplicated or the nodes have different parents.

#### **Kernel Transformations**

Kernel transformations work on the fparser2 AST of the target kernel code. This AST is obtained by converting the fparser1 AST (stored when the kernel code was originally parsed to process the meta-data) back into a Fortran string and then parsing that with fparser2. (Note that in future we intend to adopt fparser2 throughout PSyclone so that this translation between ASTs will be unnecessary.) The *ast* property of the *psyclone.psyGen.Kern* class is responsible for performing this translation the first time it is called. It also stores the resulting AST in

18.5. Modules 163

*Kern.\_fp2\_ast* for return by future calls. Transforming a kernel is then a matter of manipulating this AST. (See *psyclone.transformations.ACCRoutineTrans* for an example.)

## **OpenACC Support**

PSyclone is able to generate code for execution on a GPU through the use of OpenACC. Support for generating OpenACC code is implemented via *Transformations*. The specification of parallel regions and loops is very similar to that in OpenMP and does not require any special treatment. However, a key feature of GPUs is the fact that they have their own, on-board memory which is separate from that of the host. Managing (i.e. minimising) data movement between host and GPU is then a very important part of obtaining good performance.

Since PSyclone operates at the level of Invokes it has no information about when an application starts and thus no single place in which to initiate data transfers to a GPU. (We assume that the host is responsible for model I/O and therefore for populating fields with initial values.) Fortunately OpenACC provides support for this kind of situation with the enter data directive. This may be used to "define scalars, arrays and subarrays to be allocated in the current device memory for the remaining duration of the program" [ope17]. The ACCDataTrans transformation adds an enter data directive to an Invoke:

class psyclone. transformations. ACCDataTrans

Adds an OpenACC "enter data" directive to a Schedule. For example:

```
>>> from psyclone.parse import parse
>>> from psyclone.psyGen import PSyFactory
>>> api = "gocean1.0"
>>> filename = "nemolite2d_alg.f90"
>>> ast, invokeInfo = parse(filename, api=api, invoke_name="invoke")
>>> psy = PSyFactory(api).create(invokeInfo)
>>>
>>> from psyclone.psyGen import TransInfo
>>> t = TransInfo()
>>> dtrans = t.get_trans_name('ACCDataTrans')
>>>
>>> schedule = psy.invokes.get('invoke_0').schedule
>>> schedule.view()
>>>
>>> # Add an enter-data directive
>>> newschedule, _ = dtrans.apply(schedule)
>>> newschedule.view()
```

The resulting generated code will then contain an enter data directive protected by an IF (this is the first time in this Invoke) block, e.g. (for the GOcean1.0 API):

```
! Ensure all fields are on the device and
! copy them over if not.
IF (first_time) THEN
  !$acc enter data &
   !$acc& copyin(sshn_t,sshn_t*data,un*grid,un*grid*tmask,...)
   first_time = .false.
   ssha_t*data_on_device = .true.
   ...
```

Note that the IF block is not strictly required as the OpenACC run-time identifies when a reference is already on the device and does not copy it it over again. However, when profiling an application, it was seen that there was a small overhead associated with doing the enter data, even when the data was already on the device. The IF block eliminates this.

Of course, a given field may already be on the device (and have been updated) due to a previous Invoke. In this case,

the fact that the OpenACC run-time does not copy over the now out-dated host version of the field is essential for correctness.

18.5. Modules 165

# **BIBLIOGRAPHY**

[nem13] *NEMO Coding Conventions*. 2013. URL: https://forge.ipsl.jussieu.fr/nemo/attachment/wiki/Literature/NEMO\_coding.conv\_v3.pdf.

[ope17] The OpenACC Application Programming Interface, Version 2.6. 2017. URL: http://www.openacc.org.

168 Bibliography

# **PYTHON MODULE INDEX**

# р

```
psyclone.algGen, 128
psyclone.generator, 113
psyclone.line_length, 129
psyclone.parse, 114
psyclone.psyGen, 128
psyclone.transformations, 114
```

170 Python Module Index

# **INDEX**

Symbols _validate() (psyclone.psyGen.Transformation method),	apply() (psyclone.transformations.GOLoopSwapTrans method), 121
163	apply() (psyclone.transformations.KernelModuleInlineTrans method), 123
A	apply() (psyclone.transformations.LoopFuseTrans
ACCDataTrans (class in psyclone.transformations), 114, 164	method), 123 apply() (psyclone.transformations.MoveTrans method), 123
ACCLoopTrans (class in psyclone.transformations), 115 ACCParallelTrans (class in psyclone.transformations), 116	apply() (psyclone.transformations.OMPLoopTrans method), 124
ACCRoutineTrans (class in psyclone.transformations),	apply() (psyclone.transformations.OMPParallelLoopTrans method), 125
Alg (class in psyclone.algGen), 128	apply() (psyclone.transformations.ParallelLoopTrans method), 126
api (psyclone.configuration.Config attribute), 160 api_conf() (psyclone.configuration.Config method), 161	apply() (psyclone.transformations.ParallelRegionTrans
apply() (psyclone.psyGen.Transformation method), 163 apply() (psyclone.transformations.ACCDataTransmethod), 115	method), 127 apply() (psyclone.transformations.ProfileRegionTransmethod), 127
apply() (psyclone.transformations.ACCLoopTrans method), 116	С
apply() (psyclone.transformations.ACCRoutineTrans method), 117	check_intergrid() (in module psyclone.transformations), 127
apply() (psyclone.transformations.ColourTrans method), 117	coloured_text (psyclone.psyGen.Schedule attribute), 25 ColourTrans (class in psyclone.transformations), 117
apply() (psyclone.transformations.Dynamo0p3ColourTransmethod), 118	_
apply() (psyclone.transformations.Dynamo0p3OMPLoopT	rd <del>u</del> ds
method), 119 apply() (psyclone.transformations.Dynamo0p3RedundantC	dag_name (psyclone.psyGen.Schedule attribute), 25
method), 119	default_stub_api (psyclone.configuration.Config at-
apply() (psyclone.transformations.DynamoLoopFuseTrans	tribute), 161
method), 119 apply() (psyclone.transformations.DynamoOMPParallelLo	distributed_memory (psyclone.configuration.Config at- opTrans tribute), 161
method), 120	Dynamo0p3ColourTrans (class in psy-
apply() (psyclone.transformations.GOceanLoopFuseTrans	clone.transformations), 118
method), 121	Dynamo0p3OMPLoopTrans (class in psy-
apply() (psyclone.transformations.GOceanOMPLoopTrans	
method), 122 apply() (psyclone.transformations.GOceanOMPParallelLoc	Dynamo0p3RedundantComputationTrans (class in psy- ppTrans clone.transformations), 119
method), 122	DynamoLoopFuseTrans (class in psy-
apply() (psyclone.transformations.GOConstLoopBoundsTr method), 120	

DynamoOMPParallelLoopTrans (class in psyclone.transformations), 120	name (psyclone.transformations.ACCDataTrans attribute), 115
F	name (psyclone.transformations.ACCLoopTrans attribute), 116
filename (psyclone.configuration.Config attribute), 161 find_file() (psyclone.configuration.Config static method),	name (psyclone.transformations.ACCParallelTrans attribute), 117
161	name (psyclone.transformations.ACCRoutineTrans at-
first_access() (psyclone.psyGen.Invoke method), 24 FortLineLength (class in psyclone.line_length), 129	tribute), 117 name (psyclone.transformations.ColourTrans attribute),
G	name (psyclone.transformations.Dynamo0p3ColourTrans
gen (psyclone.algGen.Alg attribute), 129	attribute), 118
generate() (in module psyclone.generator), 113	name (psyclone.transformations.Dynamo0p3OMPLoopTrans attribute), 119
get() (psyclone.configuration.Config static method), 161 get_default_keys() (psyclone.configuration.Config	name (psyclone.transformations.Dynamo0p3RedundantComputationTrans
method), 162	attribute), 119
get_trans_name() (psyclone.psyGen.TransInfo method),	name (psyclone.transformations.DynamoLoopFuseTrans attribute), 120
get_trans_num() (psyclone.psyGen.TransInfo method),	name (psyclone.transformations.DynamoOMPParallelLoopTrans attribute), 120
GOceanLoopFuseTrans (class in psyclone.transformations), 121	name (psyclone.transformations.GOceanLoopFuseTrans attribute), 122
GOceanOMPLoopTrans (class in psy-	name (psyclone.transformations.GOceanOMPLoopTrans
clone.transformations), 122 GOceanOMPParallelLoopTrans (class in psy- clone.transformations), 122	attribute), 122 name (psyclone.transformations.GOceanOMPParallelLoopTrans attribute), 122
GOConstLoopBoundsTrans (class in psy- clone.transformations), 120	name (psyclone.transformations.GOConstLoopBoundsTrans attribute), 121
GOLoopSwapTrans (class in psyclone.transformations), 121	name (psyclone.transformations.GOLoopSwapTrans attribute), 121
I	name (psyclone.transformations.KernelModuleInlineTrans attribute), 123
inline() (psyclone.psyGen.PSy method), 128 Invoke (class in psyclone.psyGen), 24	name (psyclone.transformations.LoopFuseTrans attribute), 123
Invokes (class in psyclone.psyGen), 24	name (psyclone.transformations.MoveTrans attribute), 123
KernelModuleInlineTrans (class in psy-	name (psyclone.transformations.OMPLoopTrans attribute), 125
clone.transformations), 122	name (psyclone.transformations.OMPParallelLoopTrans attribute), 125
L	name (psyclone.transformations.OMPParallelTrans at-
length (psyclone.line_length.FortLineLength attribute), 129	tribute), 126 name (psyclone.transformations.ParallelLoopTrans at-
list (psyclone.psyGen.TransInfo attribute), 29 load() (psyclone.configuration.Config method), 162	tribute), 126 name (psyclone.transformations.ParallelRegionTrans at-
long_lines() (psyclone.line_length.FortLineLength method), 129	tribute), 127 name (psyclone.transformations.ProfileRegionTrans at-
LoopFuseTrans (class in psyclone.transformations), 123	tribute), 127
M	num_trans (psyclone.psyGen.TransInfo attribute), 29
MoveTrans (class in psyclone.transformations), 123	0
N	omp_schedule (psyclone.transformations.OMPLoopTrans attribute), 125
name (psyclone psyGen Transformation attribute) 163	OMPLoopTrans (class in psyclone.transformations), 124

172 Index

```
OMPParallelLoopTrans
                             (class
                                          in
                                                   psy-
         clone.transformations), 125
OMPParallelTrans (class in psyclone.transformations),
Р
ParallelLoopTrans (class in psyclone.transformations),
ParallelRegionTrans (class in psyclone.transformations),
         127
parse() (in module psyclone.parse), 114
process() (psyclone.line_length.FortLineLength method),
ProfileRegionTrans (class in psyclone.transformations),
         127
PSy (class in psyclone.psyGen), 128
psyclone.algGen (module), 128
psyclone.generator (module), 113
psyclone.line_length (module), 129
psyclone.parse (module), 114
psyclone.psyGen (module), 128
psyclone.transformations (module), 114
RegionTrans (class in psyclone.transformations), 127
reprod pad size
                  (psyclone.configuration.Config
         tribute), 162
reproducible_reductions (psyclone.configuration.Config
         attribute), 162
S
Schedule (class in psyclone.psyGen), 24
supported_apis (psyclone.configuration.Config attribute),
supported_stub_apis (psyclone.configuration.Config at-
         tribute), 162
Transformation (class in psyclone.psyGen), 163
TransformationError, 127
TransInfo (class in psyclone.psyGen), 29
U
unique_declarations() (psyclone.psyGen.Invoke method),
         24
unique_declns_by_intent()
                              (psyclone.psyGen.Invoke
         method), 24
V
view() (psyclone.psyGen.Schedule method), 25
```

Index 173