

PSyKAI: a Code Generation to Performance Portability

[Extended Abstract]

Mike Ashworth

STFC Daresbury Laboratory
Sci-Tech Daresbury
Warrington WA4 4AD UK
+44-1925-603178
mike.ashworth@stfc.ac.uk

Rupert Ford

STFC Daresbury Laboratory
Sci-Tech Daresbury
Warrington WA4 4AD UK
+44-1925-603178
rupert.ford@stfc.ac.uk

Andrew Porter

STFC Daresbury Laboratory
Sci-Tech Daresbury
Warrington WA4 4AD UK
+44-1925-603178
andrew.porter@stfc.ac.uk

Chris Maynard

The Met Office
Fitzroy Road
Exeter EX1 3PB UK
+44 1392 885680
christopher.maynard@metoffice.gov.uk

Thomas Melvin

The Met Office
Fitzroy Road
Exeter EX1 3PB UK
+44 1392 885680
thomas.melvin@metoffice.gov.uk

ABSTRACT

This paper presents a domain specific approach to performance portability and code complexity reduction in finite element and finite difference codes. This approach has been developed for the Met Office's next generation atmospheric model which uses finite elements on a quasi-uniform grid, and has also been prototyped on two finite difference Ocean model benchmarks, one of which is based on the NEMO ocean model. The approach is outlined and the API's for the atmosphere and ocean models are discussed.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors – *code generation, parsing, preprocessors*.

General Terms

Algorithms, Performance, Languages.

Keywords

GungHo, PSyKAI, PSyclone, atmospheric modelling, ocean modelling.

1. INTRODUCTION

The Met Office's numerical weather prediction and climate model code, the Unified Model (UM), is almost 25 years old. Up to the present day the UM has been able to be run efficiently on many of the world's most powerful computers, helping to keep the Met Office at the forefront of climate prediction and weather forecasting.

However, with performance increases from each new generation of computers now being primarily provided by an increase in the amount of parallelism rather than an increase in the clock-speed of the processors themselves, running higher resolutions of the UM now faces the double challenge of code scalability and numerical accuracy.

The UM's atmospheric dynamical core makes use of a finite-difference scheme on a regular latitude-longitude grid. The regular latitude-longitude grid results in an increasingly disparate grid resolution as the resolution increases, due to lines of longitude converging at the poles. For example, a 10km resolution at mid-latitudes would result in a 12m resolution at the poles. The difference in resolution leads to increased communication at the

poles and load balance issues which are known to impair scalability; it also leads to issues with numerical accuracy and smaller time-steps due to the difference in scale.

To address this problem the Met Office, NERC and STFC initiated the GungHo project. The primary aim of this project is to deliver a scalable, numerically accurate dynamical core. This dynamical core is scheduled to become operational around the year 2022. The project is currently investigating the use of quasi-uniform meshes, such as triangular, icosahedral and cubed-sphere meshes, using finite element methods.

The associated GungHo software infrastructure is being developed to support multiple meshes and element types thus allowing for future model development. GungHo is also proposing a novel separation of concerns for Dynamo - 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 PSyKAI (pronounced as 'cycle').

Rather than writing the PSy layer manually, the GungHo project is developing a code generation system called PSyclone which can generate correct code and 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.

In the GOcean project the PSyKAI approach and the PSyclone system have been extended for use with finite difference models and applied to two Ocean benchmarks, a shallow water model and a cut down version of the NEMO ocean model.

In the following sections the PSyKAI approach is discussed in more detail, the PSyclone system is introduced and the Dynamo and GOcean API's are outlined

2. PSYKAI

The PSyKAI approach separates code into three layers, the Algorithm layer, the PSy layer and the Kernel layer. Whilst this approach is general we have applied it to Atmosphere and Ocean models written in Fortran where domain decomposition is typically performed in the latitude-longitude direction, leaving columns of elements on each domain-decomposed partition.

The top layer, in terms of calling hierarchy, is the Algorithm layer. This layer specifies the algorithm that the scientist would like to perform (in terms of calls to kernel and infrastructure routines) and logically operates on full fields. We say logically here as the fields may be domain decomposed, however the algorithm layer is not aware of this. It is the scientists responsibility to write this algorithm layer.

The bottom layer, in terms of calling hierarchy, is the Kernel layer. The Kernel layer implements the science that the Algorithm layer calls, as a set of subroutines. These kernels operate on local fields (a set of elements, a single column of elements, or a set of columns, depending on the kernel). Again the scientist is responsible for writing this layer and there is no parallelism specified here, but there is likely to be input from an HPC expert and/or some coding rules to help make sure the kernels compile into efficient code.

The PSy layer sits in-between the Algorithm and Kernel layers and its functional role is to link the algorithm calls to the associated kernel subroutines. As the Algorithm layer works on logically global fields and Kernel layer works on local fields the PSy layer is responsible for iterating over columns. It is also responsible for including any required distributed memory operations, such as halo swaps and reductions.

As the PSy layer iterates over columns, the potential parallelism within this iteration space can be optimised and parallelised. The PSy layer can therefore 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.

As an example, consider the following “traditional” code fragment:

```
!$OMP PARALLEL DO ...
do i = 1, nlat
  do j = 1, nlon
    do k = 1, levels
      a(k,j,i) = ...
      b(k,j,i) = ...
    end do
  end do
end do
!$END OMP PARALLEL DO ...
```

In the PSyKAI approach this code would be split into the algorithm layer:

```
call psy_...(a,b,...)
```

the PSy layer:

```
subroutine psy_...(a,b,...)
!$OMP PARALLEL DO ...
do i = 1, nlat
  do j = 1, nlon
    call kern1(a,...)
    call kern2(b,...)
  end do
end do
!$END OMP PARALLEL DO ...
end subroutine psy_...
```

and the kernel layer:

```
subroutine kern1(a,...)
do k = 1, levels
  a(k) = ...
end do
end subroutine kern1
```

```
subroutine kern2(b,...)
do k = 1, levels
  b(k) = ...
end do
end subroutine kern2
```

In this way parallelism is encapsulated in the PSy layer and the latitude-longitude iteration space can be parallelised in different ways: the directives could be changed, the loops could be split, halo calls could be added etc. None of these changes would require modification to the Algorithm or Kernel layers.

One difference can be observed here. As kernels contain whole columns, it is not possible to manually merge the “k” loops into one (which was implemented in the original example code fragment). Such a modification would have to be performed by the compiler, if the compiler considered it appropriate.

Clearly the splitting of code into separate layers will have an effect on performance. This overhead and how to get back to the performance of the “traditional” code, and potentially improve on it, will be discussed in the talk.

A potentially useful way to think of the PSyKAI approach is that traditional codes are like a set of lego bricks which have been glued together. Typically these lego bricks have been joined in a way that works well on certain architectures, however, that form may not work so well on different architectures and/or compilers. The PSyKAI approach splits code into individual lego bricks and allows these bricks to be put back together in the most appropriate way for a particular architecture/compiler combination

3. PSYCLONE

PSyclone is a code generation system which takes algorithm and kernel code as input, and outputs modified algorithm code and a generated PSy layer. PSyclone also supports transformations to the PSy layer to improve performance. The code output from PSyclone can be compiled and linked with the associated kernels to produce correct performant code.

At the time of writing, the PSyclone system has been integrated with the Met Office's Dynamo software and this integration is in its final review stage. Once this review is complete the Dynamo software will be built using the PSyclone system to generate the PSy layer.

In the example in the previous section it can be seen that a single call from the algorithm layer results in two calls to kernels. Algorithm code writers will not want to manually create calls to the PSy layer and determine what arguments to pass to them. However, for performance reasons it is desirable to capture the fact that multiple kernels can be called together as this gives the most flexibility for performance improvement in the PSy layer. Therefore PSyclone supports an Application Programmer Interface (API) where multiple calls from the Algorithm layer can be put together in a single “invoke” call. As an illustration, the algorithm example in the previous section becomes:

```
call invoke(kern1_type(a,...), &
           kern2_type(b,...))
```

The *invoke* call specifies that a particular set of kernels need to be called, but does not dictate their order. The PSy layer is free to re-order calls within an *invoke* as required as long as any data dependencies are honoured. For the best chance to obtain good performance the strategy at the algorithm layer should be to place as many kernels as possible within as few *invokes* as possible.

The *invoke* syntax is not necessary; however it does ensure that the algorithm developer does not accidentally place code in between

two kernel calls. In effect, it identifies multi-kernel sections of potentially parallel optimisable code.

PSyclone parses the algorithm layer and replaces any invoke calls with a single direct call to the PSy layer. As PSyclone also generates the PSy layer it is in control of the names it uses to do this.

Therefore PSyclone would change the invoke call into something like the following:

```
call psy_...(a,b,...)
```

In the above invoke syntax the kernel identifiers are *kern1_type* and *kern2_type*. However, in the kernel code these identifiers do not exist. This is because these identifiers point to metadata that describes the behaviour of the kernels. PSyclone uses this metadata and the algorithm *invoke* set of calls to generate correct PSy layer code.

In GungHo and GOcean it was decided to encode and maintain kernel metadata as a fortran type within a kernel. Therefore the *kern1_type* and *kern2_type* identifiers exist as types in the respective kernel modules. For example

```
module kern1_mod
  type kern1_type
  ...
end type
subroutine kern1(a,...)
  do k = 1, levels
    a(k) = ...
  end do
end subroutine kern1
end module kern1_mod
```

The specific content of the kernel metadata depends on the particular API. Kernel metadata will be discussed in the Dynamo and GOcean sections.

Users of a code generation system such as PSyclone will not want to write kernels for all types of functionality. For example, take the case of well-known operations, such as matrix multiply, which already have efficient implementations. Further, simple operations, such as assignment should probably not need to be implemented as kernels. Therefore PSyclone is designed so that a particular API can recognise certain names and add in appropriate code without it existing in the kernel layer. For example, in the following algorithm code:

```
call invoke(kern1_info(a,...), &
           matvec(a,b,...), &
           kern2_info(b,...))
```

the *matvec* call is recognised by the particular API in PSyclone, and within the generated PSy layer an appropriate call is made to a library, or some specific code, as chosen by the user. In PSyclone these calls are called *intrinsic* calls as they are managed by the system rather than needing explicit kernel implementations, much like intrinsics in languages such as Fortran.

Once PSyclone has parsed the algorithm code and kernel metadata it creates an internal tree representation of the PSy layer which consists of a schedule containing loops and calls. This tree can output Fortran code when requested but also serves as a structure that can be manipulated for code optimisations.

A part of PSyclone separate from the code parsing and code generation supports transformations. Transformations modify PSyclone's internal tree. At the present time there are a small number of transformations provided, including, OpenMP parallel do directives, OpenMP parallel region directives, loop splitting, loop merging and loop colouring. However adding new transformations is not arduous and this list will grow as new optimisations are identified.

Thus, PSyclone is able to parse algorithm code and kernel metadata, generate an internal representation of the PSy layer, optimise the PSy layer for a particular architecture via a user written recipe of transformations, and then generate appropriate PSy layer fortran code.

Optimisations have been purposely made available to users (primarily HPC experts) rather than attempting to optimise code automatically from the outset. The reason for this is that it is the authors' experience that the required optimisations for a code vary from one architecture to another and from one compiler to another and further, a small changes in code structure can give very large performance differences. Of course automation is not precluded and one place where automation is expected to help in the short term is in searching the space of optimisations. The authors plan to add support for this in the near future.

Lastly, PSyclone has been designed to support multiple API's. A set of base classes can be extended to support a particular API. The majority of the code, including the transformations, is independent of the API. Two such API's, one for the Met Office's next generation atmosphere model written to use finite elements (called Dynamo) and the other for a cut down benchmark of the NEMO ocean model, written in the GOcean project, are discussed in the following sections.

4. DYNAMO

Dynamo is the Met Office's next generation atmosphere model which is currently being developed as part of the GungHo project. The Dynamo API has been changing as new functionality has been developed. However, recently the API was considered mature enough to be frozen into a version and the decision was made to move to using the PSyclone system. The current API is 0.3 and PSyclone can generate correct sequential code for this.

Below is an example of a dynamo algorithm invoke call for the 0.3 API taken from the code. This invoke only specifies one kernel:

```
call invoke(rtheta_kern_type(rt,u,chi,qr))
```

The associated kernel metadata for the 0.3 API is given below:

```
type, ... :: rtheta_kern_type
  type(arg_type)::meta_args(3)=(/ &
    arg_type(GH_FIELD,GH_INC,W0), &
    arg_type(GH_FIELD,GH_READ,W2), &
    arg_type(GH_FIELD*3,GH_READ,W0) &
  /)
  type(func_type)::meta_funcs(2)=(/ &
    func_type(W0,GH_BASIS,GH_DIFF_BASIS), &
    func_type(W2,GH_BASIS,GH_ORIENTATION) &
  /)
  integer::iterates_over=CELLS
contains
  procedure,nopass::rtheta_code
end type
```

The three *meta_args* entries describe the use of the first three arguments passed from the algorithm layer (*rt,u,chi*). Each

`meta_args` entry has three values. Examining the first one of these we see that it is a field object (as they all are in this case, but there are other options), it is accessed as an increment ($rt=rt+...$) in the kernel and the kernel expects the argument to be on the W0 finite element function space. The metadata for the third argument (*chi*) specifies *3. This indicates that the kernel expects *chi* to be a vector of fields of size 3.

The *meta_funcs* entries provide information about the function space that the kernel requires internally. Thus, for the W0 function space the kernel requires basis-function information and differential-basis-function information.

As the kernel requires basis-function information the user needs to specify the required quadrature rule (quadrature can vary in the code and could change over time). Therefore an additional quadrature rule object is required to be passed from the algorithm layer. This is called *qr* in the above example.

The *iterates_over* value specifies what the kernel expects the Psy layer to iterate over – this can be cell, vertices, etc. This information is used by PSyclone to determine whether colouring is needed when parallelising in the Psy layer.

Finally, the procedure specification specifies the name of the actual kernel code.

5. GOCEAN

GOcean was a proof-of-principle project which aimed to determine whether the PSyKAl and PSyclone approach would be suitable for Ocean models. It concentrated on finite difference based ocean models as it was expected that finite element based ocean models were likely to be amenable.

Two benchmarks were developed and were hand optimised with and without using the PSyKAl separation of layers on a range of architectures and compilers. These results will be presented in the talk.

More recently PSyclone has been extended to support the GOcean API and work is ongoing to make the PSyclone generated code as efficient as the hand optimised code by adding in appropriate transformations.

Below is a cut down example of the kernel metadata for the 1.0 API, the algorithm interface is not included as it is similar to that used in Dynamo:

```

type, ... :: continuity
  type(arg)::meta_args(10)=(/ &
    arg(WRITE,CT,POINTWISE), &
    arg(READ,CU,POINTWISE), &
    arg(READ,CV,POINTWISE), &
    ...
    arg(READ, TIME_STEP), &
    arg(READ, GRID_AREA_T) &
  /)
  integer :: ITERATES_OVER = DOFS
  integer :: index_offset = OFFSET_NE
contains
  procedure,nopass:: &
    code=>continuity_code
end type continuity

```

In the above example the *meta_args* entries with three values describe the kernels expectation and use of the arguments passed from the Algorithm layer. Examining the first entry we see that it is written to in the kernel and its C-grid staggering position is on T points. The pointwise argument is ignored at the moment.

The final two *meta_args* entries only have two arguments and describe additional information required by the kernel that is not provided by the algorithm layer. In the first case it is the current timestep and in the second case is a grid area property.

The *iterates_over* argument is ignored at the moment and the *index_offset* information specifies the assumed relative index space offsets for the different staggings. This information, when combined with the staggering positions of the arguments, allows PSyclone to generate correct offsets when creating and optimising the Psy layer. The fact the user does not need to worry about such indexing is expected to make coding much easier and should introduce fewer bugs.

Finally, as with the Dynamo API, the procedure specification specifies the name of the actual kernel code.

6. ACKNOWLEDGEMENTS

The GungHo work reported here was funded by the STFC Hartree Centre, and the GOcean work was funded by NERC under grant reference NE/L012111/1. The work presented here has also benefitted from the input of other members of the GungHo and GOcean teams