The Bespoke Framework Generator (BFG)

Rupert Ford
Graham Riley
Overview

- What is BFG?
- How is it implemented?
- Current status
- Example use
- Future work
What is BFG?

- Bespoke Framework Generator
- BFG takes as input, metadata describing individual models (e.g. subroutines or methods), their composition into a coupled model and their deployment onto resources, and generates the required wrapper code (e.g. main(s) and communication code).

- “Meta-Coupler” © Rocky
BFG Metadata

- **DCD** metadata
  - **Define** individual models characteristics
    - Coupling data it requires and can provide
    - How the data is received and provided (mechanism)
    - Entry points (subroutines/methods)
    - Language, timestep etc…
  - **Compose** individual models into a coupled model in terms of science
    - Connect outputs of models to inputs of other models
    - Specify a schedule
  - Specify how to **Deploy** a coupled model onto resources
Flexibility

- Flexibility in composition
  - Change the way the coupled model starts up - priming
  - Connect any output to any input
  - Replace one model with another
- Flexibility in deployment
  - future-proofing
  - Use most appropriate framework
- Flexibility for performance – target most appropriate framework and configuration (i.e. mapping of models to executables)
BFG1 and BFG2

• BFG1 developed to demonstrate
  • FLUME flexibility requirements could be supported
  • Separating science from coupling infrastructure (targets).
  • Use of metadata (define, compose, deploy) for code generation
  • Flexibility in composition
  • Same model code, same composition, can choose many different targets (flexible deployment)
BFG1: Deployment Flexibility

Support for many targets, therefore choose most appropriate:
  in-sequence (shared memory buffers), mpi, Oasis3, tdt, web services
No change to model code (model description or composition description)
Deployment Flexibility

Ability to choose most appropriate mapping of models to executables, with no change to model code (or composition)
BFG1 and BFG2

- BFG2 developed to test ability to use this approach for full Flume requirements (Flume – Flexible Unified Model Environment, Met Office project)
- Additions to BFG(1) (so far)
  - Argument passing as a communication mechanism – same performance as hand crafted code
  - Complex initialisation and priming of data
  - Fortran 90 module as a model
  - Multiple entry points
  - init, run and finalise phases
  - More complex control flow (arbitrary nested loops)
  - Multi-instance support
  - MPI and OASIS4 targets (no ESMF or OASIS3 yet)
  - XML gridspec definitions for OASIS4
  - No model parallelism as no parallel partition metadata
- Rewrite of code-gen software architecture due to additional complexity – multiple phase approach
A BFG2 model

- A set of one or more subroutines/methods which perform one simulation time step and associated descriptive metadata.
- A Fortran 90 illustration ...

```fortran
module exampleModule
  private
  public ts
  subroutine ts(a)
    use bfg, only : put
    ! do something clever
    call put(e,9)
  end subroutine ts
end Module exampleModule
```
Model metadata

<definition ...>
<name>exampleModule</name>
<type>scientific</type>
<language>f90</language>
<entryPoints>
  <entryPoint type="iteration" name="ts">
    <data direction="inout" form="argpass" id="1"
      dataType="real" dimension="0"/>
    <data direction="out" form="inplace" id="9"
      dataType="real" dimension="0"/>
  </entryPoint>
</entryPoints>
<timestep units="hours">1</timestep>
</definition>
BFG2: Mixed, concurrent/in-sequence

threads/processes

- call atm(a)
- call atm_chem(a)
- call put(a,tag)
- call ocn_chem(a)

mpi

threads/processes

- call get(a,tag)
- call ocn(a)
- call ocn_chem(a)
BFG2 Implementation

- Phases
- Different bits of code
  - Main program(s)
  - Target specific
  - Put/get
  - configuration
- Implemented with xslt
- Generic XML description with a final codegen phase
BFG2 model wrapping

- Container, Control, ArgPass Data, ArgPass Wrapper
- Model Code(s)
- InPlace Calls
- Concurrency
- Target Coupling Infrastructure

- Existing code/library code
- BFG-generated code
BFG2 Utilities

- bfg2check.py
- bfg1tobfg2.py
- bfg2graphmlgen.py
- bfg2makefilegen.py
- bfg2modelstubgen.py
- bfg2svggen[12].py
- runbfg2.py
Example GENIE Configuration in BFG2
CIAS 2 model Example running BFG2

Generate stub code

Generate stub code

Generate coupling code

Compile generated code
eb_go_gs_ac_bg example!

- Running with 1 thread: BFG2 reproduces the same results as Genie. Both versions take the same time.
- Running with 2 threads: Asis - BFG2 reproduces the same results as Genie. Runs slightly slower due to (mpi) comms costs as no concurrency in configuration.

```xml
<sequenceUnit threads="1">
    <model name="goldstein_seaice"/>
    <model name="embm"/>
    <model name="surflux"/>
    <model name="weight_check"/>
    <model name="counter" instance="1"/>
    <model name="counter" instance="2"/>
    <model name="counter" instance="3"/>
    <model name="goldstein"/>
</sequenceUnit>
<sequenceUnit threads="1">
    <model name="bfg_increment_genie_clock"/>
    <model name="biogem"/>
    <model name="atchem" instance="1"/>
    <model name="atchem" instance="2"/>
</sequenceUnit>
```
Coupling Technologies for Earth System Modelling: Today and Tomorrow. Toulouse 15th-17th December 2010

eb_go_gs_ac_bg 500 iterations

Initialise phase

Timestepping phase

Finalise phase

Zoom

goldstein (+ embm)

biogem (+ atchem)

diag_bisogem_timeseries

8.542956829071045
eb_go_gs_ac_bg without concurrency on a dual core pentium D
Concurrent Coupled Models

- Data dependence with one priming vs. concurrent with both primed
eb_go_gs_ac_bg with concurrency (concurrent priming) on a dual core pentium D
eb_go_gs_ac_bg with concurrency (concurrent priming) on a dual core pentium D for multiple timesteps
eb_go_gs_ac_bg with concurrency (concurrent priming) on a dual core pentium D for whole run
BFG metadata changes to make composition concurrent

<priming>
<model name="biogem">
  <entryPoint name="biogem">
    <primed id="12">
      <file dataRef="R121" nlName="conc_pri" name="conc_pri.nam" type="namelist"/>
    </primed>
    <primed id="8">
      <file dataRef="R46" nlName="conc_pri" name="conc_pri.nam" type="namelist"/>
    </primed>
    <primed id="9">
      <file dataRef="R47" nlName="conc_pri" name="conc_pri.nam" type="namelist"/>
    </primed>
    ...
  </entryPoint>
</model>
</priming>
Timing results

Results from running on a dual core Pentium D. Sequential Init & Finalise costs account for 3.4s.

<table>
<thead>
<tr>
<th>Configuration of eb_go_gs_ac_bg</th>
<th>1 Deployment Unit, 1 Sequence Unit</th>
<th>1 Deployment Unit, 2 Sequence Units, original priming</th>
<th>1 Deployment Unit, 2 Sequence Units, concurrent priming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wall clock time</td>
<td>31.6s</td>
<td>33.9s</td>
<td>27.7s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Individual model timings (per call)</th>
<th>Dual core Pentium D 1DU2SU, original priming</th>
<th>Dual core Pentium D 1DU2SU, concurrent priming</th>
<th>2 machines over ethernet (2DU1SU), concurrent priming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goldstein time</td>
<td>$7 \times 10^{-2}$s</td>
<td>$10 \times 10^{-2}$s</td>
<td>$7 \times 10^{-2}$s</td>
</tr>
<tr>
<td>Biogem time</td>
<td>$1.9 \times 10^{-2}$s</td>
<td>$2.5 \times 10^{-2}$s</td>
<td>$1.9 \times 10^{-2}$s</td>
</tr>
</tbody>
</table>
BFG future directions

- Current funding from FP7 IS-ENES, FP7 METAFORE and FP7 ERMITAGE
- Develop and use CIM metadata to drive BFG2
- Simplification of processing
- Robustness, documentation
- Parallel models (requires partition metadata)
- Levels of compliance (models with their own control code)
- Support for coupling with “native” models
- Multiple frameworks (adaptor based?)
- Composite models (model hierarchies)
- Support for other languages (adaptor based?)
- Other targets (ESMF, Oasis3, Web Services, …)
- Support for service models
- BFG2 as a web service
- Metadata/code consistency
Future look

- Multiple frameworks
  - Probably in ESM
  - Very likely between domains
- More concurrent components
- More modular code (more sharing of code)
- Re-use of code in different environments (web service ... tightly coupled ... standalone)
- No (performance) overhead couplers
- Standard Metadata descriptions (from code or defining code?) – associated consistency issue
- Framework interoperability?
  - Standard Metadata descriptions
  - Adaptor code
- Generative approach
(Generated?) Adaptor code

ESMF component 1

ESMF component 2

O2E

E2O

OASIS component 1

Coupling Technologies for Earth System Modelling : Today and Tomorrow. Toulouse 15th-17th December 2010
Thanks ...
Point to Point vs Set Notation

module m1model
  ...
  subroutine m1init()
    ! Do things
    call put(a,3)
  end subroutine m1init
  ...
  subroutine m1()
    call get(x,6)
    ! Do things
    call put(y,3)
  end subroutine m1
  ...
end module m1model

module m2model
  ...
  subroutine m2init()
    ! Do things
    call get(c,2)
  end subroutine m2init
  ...
  subroutine m2()
    call get(a,1)
    ! Do things
    call put(b,1)
  end subroutine m2
  ...
end module m2model
Point to Point vs Set Notation

<modelChannel outModel="m1model"
inModel="m2model">
  <epChannel outEP="m1init" inEP="m2init">
    <connection outID="3" inID="2"/>
  </epChannel>
  <epChannel outEP="m1" inEP="m2">
    <connection outID="3" inID="1"/>
  </epChannel>
</modelChannel>

<modelChannel outModel="m2model"
inModel="m1model">
  <epChannel outEP="m2" inEP="m1">
    <connection outID="1" inID="6"/>
  </epChannel>
</modelChannel>
Point to Point vs Set Notation

• Simplicity of Set Notation for arg passing

```plaintext
call m1init(a)
do i=1,n
   if (mod(i,3).eq.0) call m1(a)
   if (mod(i,2).eq.0) call m2(a)
   call m3(a)
end do

<set name="a">
   <field modelName="m1model" epName="m1init" id="1"/>
   <field modelName="m1model" epName="m1" id="1"/>
   <field modelName="m2model" epName="m2" id="1"/>
   <field modelName="m3model" epName="m3" id="1"/>
</set>
```
Progress since last meeting

- Adapted the front-end of the gcc fortran compiler, gfortran, to produce model definition metadata from source code:
  - E.g.
    
    ```bash
    $ gfortran -fdump-parse-tree (…+ other compilation options)
    ```
  - Definition metadata no longer needs to be written manually.
  - Goldstein ocean metadata was generated using this tool.
- Worked on XML build and configuration system for GENIE
CIAS, FLUME, BFG

- CIAS (Community Integrated Assessment System) developed by Tyndall Centre. Uses BFG1 (BFG2 being integrated)
- Models written to the BFG1 rules can be used in BFG2
- Graham and I were consultants to the Met Office on FLUME.
- BFG1 and BFG2 were originally implemented to test out the ideas being developed in FLUME.
- It is hoped that FLUME models will be compatible with BFG2. The current plan is to follow the same model coding rules and to ensure that the metadata describing models are at least compatible and hopefully the same.
- FLUME plans to use Oasis4 (the latest generation PRISM Coupler) to couple models (plan to use wrapping code approach).
Running in Sequence?

**BFG (In-place) style control**

```fortran
program mycoupledmodel
  use m1model
  use m2model
  call m1init()
  call m2init()
  do i=1,nts
    call m1()
    call m2()
  end do
end program mycoupledmodel
```

**Hand-crafted Arg-passing comms (and data allocation)**

```fortran
program mycoupledmodel
  use m1model
  use m2model
  real :: a,b,c
  call m1init(a)
  call m2init(a)
  do i=1,nts
    call m1(b,c)
    call m2(b,c)
  end do
end program mycoupledmodel
```
module mlmodel
  ...
  real :: a, x, y
  ...
  subroutine m1init()
    ! Do things
    call put(a, 3)
  end subroutine m1init
  ...
  subroutine ml()
    call get(x, 6)
    ! Do things
    call put(y, 3)
  end subroutine ml
  ...
end module mlmodel

module mlmodel
  ...
  subroutine m1init(a)
    real, intent(out):: a
    ! Do things
  end subroutine m1init
  ...
  subroutine ml(x, y)
    real, intent(in) :: x
    real, intent(out) :: y
    ! Do things
  end subroutine ml
  ...
end module mlmodel
module m1model
...

subroutine m1init(a)
  real, intent(out):: a
  ! Do things
end subroutine m1init
...

subroutine m1(x,y)
  real, intent(in) :: x
  real, intent(out):: y
  ! Do things
end subroutine m1
...

end module m1model

module m1modelwrap
...

use m1model
...

real :: a,x,y
...

subroutine m1initwrap()
  call m1init(a)
  call put(a,3)
end subroutine m1initwrap
...

subroutine m1wrap()
  call get(x,6)
  call m1(x,y)
  call put(y,3)
end subroutine m1wrap
...

end module m1modelwrap
Choose most appropriate for model developer and for required use

- e.g. in-place for some diagnostics

```fortran
module m1model
  ...
  real :: y
  ...
  subroutine m1init(a)
    real, intent(out):: a
    ! Do things
  end subroutine m1init
  ...
  subroutine m1(x)
    real, intent(in) :: x
    ! Do things
    call put(y,3)
  end subroutine m1
  ...
end module m1model
```

BFG2: Mixed Arg-passing/In-place Communication
Conclusions

Flexibility in deployment allows
  choice of most efficient “target” infrastructure
  choice of most efficient mapping of models to “main’s”

Argument passing interface allows
  as efficient generated code (in memory and time) as hand crafted code
    when running in sequence
  all models to be run in-sequence, some models to be run in-sequence and
    some concurrently, all models to be run concurrently. Can choose most
    appropriate target and mapping to mains for concurrent models.
  (potentially) More fine grain coupling without loss of performance
  (potentially) Coupling for both NWP and ESM.
Why BFG?

- Flexibility in model ordering and connectivity, with no change to model code.
- Support for many targets: in-sequence, MPI, Oasis4, ESMF, (Web Services, PerCo)
- Ability to choose most appropriate target and mapping of models to executables, with no change to model code (or composition).
- Same performance as “native” implementations (e.g. genie.F).
- Interoperability of GENIE, CIAS and FLUME models (from a computer science perspective).
- Support for model “swapping” where standards exist.
Introduction: Model Wrapping

Main(s), Control, Coupling wrapper

Atm | Ocn | Sea-ice

Coupling wrapper

Target Coupling Infrastructure

- Existing code/library code
- BFG-generated wrapper code
(f90) Models and In-Place Communication

```fortran
module m1model
    ...
    real :: a, x, y
    ...
    subroutine m1init()
        ! Do things
        call put(a,3)
    end subroutine m1init
    ...
    subroutine m1()
        call get(x,6)  ! Do things
        call put(y,3)
    end subroutine m1
    ...
end module m1model

module m2model
    ...
    real :: a, b, c
    ...
    subroutine m2init()
        call get(c,2)  ! Do things
    end subroutine m2init
    ...
    subroutine m2()
        call get(a,1)  ! Do things
        call put(b,1)
    end subroutine m2
    ...
end module m2model
```

Coupling Technologies for Earth System Modelling : Today and Tomorrow. Toulouse 15th-17th December 2010