Starting MOHID Lagrangian

Ricardo Birjukovs Canelas

MARETEC, Instituto Superior Técnico, Lisbon, Portugal

Immediate concern: CleanAtlantic

Immediate concern: CleanAtlantic

- Develop modelling methodologies and capabilities to tackle a domain such as the Atlantic ocean;
- Model **several types** of litter and their evolution in time (degradation, biofouling, aging, etc);
- Identify accumulation zones, account for main sources and predict trends.

Not so immediate concerns: MOHID development

- The Lagrangian Modules are **very large** (+40 KLOCs each);
- They have aged ungracefully:
  - considerable computational and coding overhead has been introduced by continuous add-ons;
  - memory usage is not optimized
  - **not internally modular** (like most of MOHID)
- As a standalone module, they are slow

- The Lagrangian Modules are **very comprehensive**;
- Do many things:
  - Plumes and jets;
  - Water quality;
  - Sediments;
  - Oil;
  - etc
- **Integrated** in the MOHID system

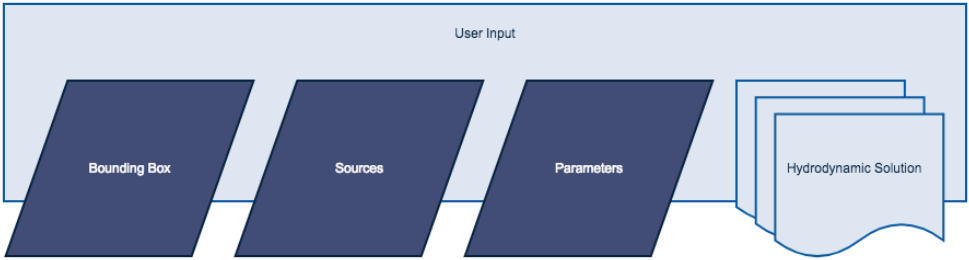
Functional Specifications

In order to be useful in the context of Maretec and the MOHID users, the new Lagrangian code should comply with:

- **Uni-directional** coupling to a hydrodynamic solution;
- Be **medium-independent**, i.e., should work across Water, Land and atmosphere;
- Present a **large collection of tracer types** and their respective methods;
- Be **efficient** ($O(10^7)$ tracers), **fast** (KISS and parallel), **extendible** (highly modular) and **readable** (it will never be finished, new coders should be kept in mind)

First looks

Our Lagrangian code should **expose** to the 'user' the following control structure



Defining a Source

Sources are complex objects:

- Geometry and location
- Lifespan
- Trajectory
- Properties to imprint on the Tracers
- Emitting rate of Tracers

Defining a useful Tracer

Our 'main' entity is probably opaque to the user. A **Pure Lagrangian Tracer** should consist of:

Parameters

- Id
- Id of the Source
- Maximum velocity
- ...

Statistics

- Average position
- Average velocity
- Average depth
- ...

State

- Active
- Age
- Position
- Velocity
- Acceleration
- Depth
- ...

Another Tracer!

A **Pure Lagrangian Tracer** is useful as a **template for other types of Tracers**. We can just make our 'plastic' tracer by adding some other properties to it:

Plastic parameters

- Density
- Degradation rate
- Size
- Particulate
- ...

Plastic statistics

- ...

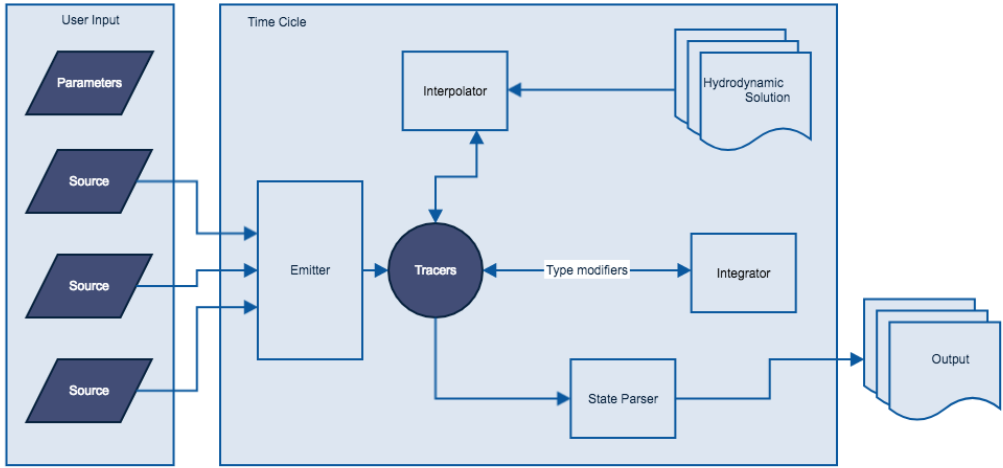
Plastic state

- Radius
- Condition
- Concentration
- ...

And then paper, tires, algae, radioactive isotopes, (...)

Once a simulation is launched

This part is mostly hidden from a 'user'



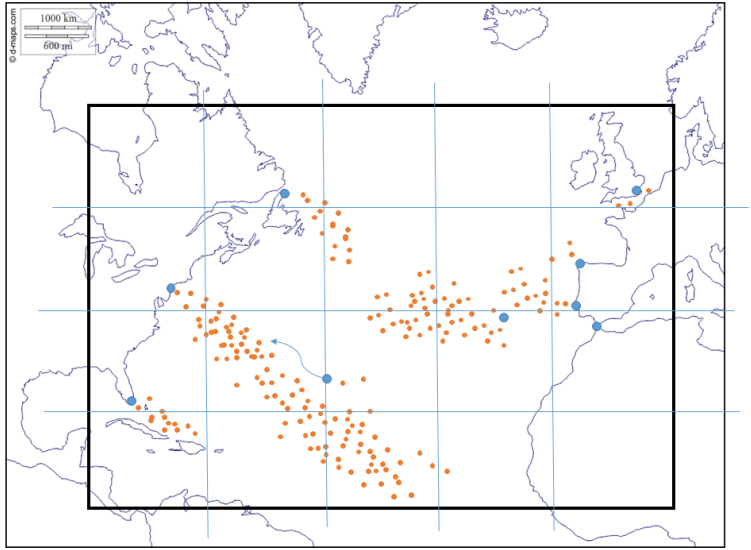
Organizing a Tracer library

A tracer type-properties library can be built over time, keeping information nice and clean:

```
<property name="bag_1">
    <particulate value="false" />
    <density value="0.7" />
    <radius value="0.2" />
    <condition value="0.95" />
    <degradation_rate value="1" />
</property>
```

A shared .xml file can be used as a library, were new entries can be added as need arises.

Ensuring scalability



Ensuring scalability - I

Employ domain decomposition even in a serial run

- Background hydrodynamic solution can be read in sub-domain blocks
- Different procedures can be done per block
- Allows higher-order optimizations on memory access patterns

Ensuring scalability - II

Finding your place in the domain takes time/memory in an arbitrary mesh:

Using memory (scales badly)

- use Ceiling()/dx and Floor()/dx on coordinates to find my grid cell
- need to store every dx in a variable resolution mesh
- what if someone gives us an unstructured mesh?

Using processor (scales at large)

- use a space filling curve and map to that
- very elegant way to decouple the problem from the mesh

Z-Order to the rescue!

Ensuring scalability - II

Finding your place in the domain takes time/memory in an arbitrary mesh:

Using memory (scales badly)

- use Ceiling()/dx and Floor()/dx on coordinates to find my grid cell
- need to store every dx in a variable resolution mesh
- what if someone gives us an unstructured mesh?

Using processor (scales at large)

- use a space filling curve and map to that
- very elegant way to decouple the problem from the mesh

Z-Order to the rescue!

So what makes our problem 'hard'?

Sources and Tracers seem bo be simple enough, as entities. Integrating a tracer in time is essentially variations on the theme

$$\frac{D\boldsymbol{x}_i}{Dt} =< \boldsymbol{u} > (\boldsymbol{x}_i, t)$$

or

$$\frac{D\boldsymbol{V}_i}{Dt} = f(< \boldsymbol{u} > (\boldsymbol{x}_i, t))$$

Most processes rely on estimating an ambient quantity: velocity, temperature, salinity, colour...

If there are many Tracers, their operations can be cheap comparing to interpolating the ambient quantities.

So what makes our problem 'hard'?

Sources and Tracers seem bo be simple enough, as entities. Integrating a tracer in time is essentially variations on the theme

$$\frac{D\boldsymbol{x}_i}{Dt} =< \boldsymbol{u} > (\boldsymbol{x}_i, t)$$

or

$$\frac{D\boldsymbol{V}_i}{Dt} = f(< \boldsymbol{u} > (\boldsymbol{x}_i, t))$$

Most processes rely on estimating an ambient quantity: velocity, temperature, salinity, colour...

If there are many Tracers, their operations can be cheap comparing to interpolating the ambient quantities.

Interpolation - I

Assuming decent memory management, interpolation is our 'hard' problem in offline mode:

- file reading for every step
- find tracer positions on the mesh
- access mesh variables per interesting cell, per tracer
- implicit and/or higher-order schemes require several ambient time-steps...

Simple for a few Tracers, doesn't scale for large numbers. Fortunately it is open for optimizations and maps well to a parallel problem!

Interpolation - I

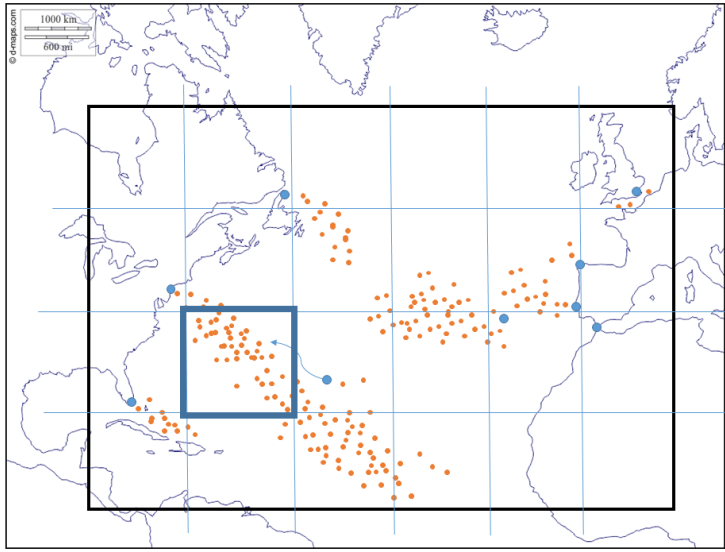
Assuming decent memory management, interpolation is our 'hard' problem in offline mode:

- file reading for every step
- find tracer positions on the mesh
- access mesh variables per interesting cell, per tracer
- implicit and/or higher-order schemes require several ambient time-steps...

Simple for a few Tracers, doesn't scale for large numbers. Fortunately it is open for optimizations and maps well to a parallel problem!

Interpolation - II

Both file 'import' and the interpolation can be done on a block basis



Interpolation - III

Once the block is in memory, two general approaches can be considered

Tracer oriented

- typically bi-linear interpolation
- another z-order curve can be built to optimize mesh node access
- simple, easy and robust
- one interpolation/quantity/Tracer/time step
- limited optimizations available

Block oriented

- interpolate all the tile!
- splines, Lagrange interpolators, etc
- one interpolation/quantity/layer/time step
- further abstracts the Tracers from the ambient mesh
- may be useful for many Tracers
- available libraries

Interpolation - III

Once the block is in memory, two general approaches can be considered

Tracer oriented

- typically bi-linear interpolation
- another z-order curve can be built to optimize mesh node access
- simple, easy and robust
- one interpolation/quantity/Tracer/time step
- limited optimizations available

Block oriented

- interpolate all the tile!
- splines, Lagrange interpolators, etc
- one interpolation/quantity/layer/time step
- further abstracts the Tracers from the ambient mesh
- may be useful for many Tracers
- available libraries

Go Parallel

We described a very parallel problem

Distributed memory (MPI style)

- domain decomposition based
- blocks can be mapped to processes
- communication is minimized - just send tracers from one block to another
- load balancing is not trivial, but we might be limited to the domain decomposition of the hydrodynamic solution already stored, so we have limited responsibility...

Shared memory (OMP style)

- each process (block or block group) can be further subdivided into cells
- cells re-use all the block code
- load balancing is trivial and with plenty of options
- because memory becomes more coalesced, efficiency should increase
- directive based acceleration should be sufficient

Go Parallel

We described a very parallel problem

Distributed memory (MPI style)

- domain decomposition based
- blocks can be mapped to processes
- communication is minimized - just send tracers from one block to another
- load balancing is not trivial, but we might be limited to the domain decomposition of the hydrodynamic solution already stored, so we have limited responsibility...

Shared memory (OMP style)

- each process (block or block group) can be further subdivided into cells
- cells re-use all the block code
- load balancing is trivial and with plenty of options
- because memory becomes more coalesced, efficiency should increase
- directive based acceleration should be sufficient

I/O formats

We should stay agnostic and portable - standard formats

Input files

- Demand NetCDF files (HDF5 is not a format...)
- Not sure about geometries, shapefiles

Output files

- again, NetCDF should be default
- There are people working exclusively on post-processors - use them.

I/O formats

We should stay agnostic and portable - standard formats

Input files

- Demand NetCDF files (HDF5 is not a format...)
- Not sure about geometries, shapefiles

Output files

- again, NetCDF should be default
- There are people working exclusively on post-processors - use them.

Mapping the problem to Fortran

Follow the golden rules of Fortran and use OOP (Fortran 2008+):

- Minimize global variables (parameters and nothing else if possible)
- Use external libraries to handle strings, vectors, file I/O and guarantee KIND portability across systems
- Be paranoid about compartmentalization and re-usability of code
- Avoid pointers
- Be pleasant to compilers and future coders
- ...

Fortran is very good with OOP nowadays

- 'Everything' is an object from a given class
- Take advantage of unlimited polymorphic classes to reduce code size and hence bug probability
- already have some library APIs ready to share

Mapping the problem to Fortran

Follow the golden rules of Fortran and use OOP (Fortran 2008+):

- Minimize global variables (parameters and nothing else if possible)
- Use external libraries to handle strings, vectors, file I/O and guarantee KIND portability across systems
- Be paranoid about compartmentalization and re-usability of code
- Avoid pointers
- Be pleasant to compilers and future coders
- ...

Fortran is very good with OOP nowadays

- 'Everything' is an object from a given class
- Take advantage of unlimited polymorphic classes to reduce code size and hence bug probability
- already have some library APIs ready to share

Start with the basics - file structure

The code should be organized as both a **library** and an **application**, an implementation of the library using its API (Application Programming Interface). So:

Project tree

- Application
  - MOHIDLagrangian.exe
    - MOHIDLagrangian.f90
- Library
  - MOHIDLagrangian.lib
    - tracers.f90
    - interpolation.f90
    - sources.f90
    - ...

This is inherently **modular**, **clean** and generates a .lib that can be used inside MOHID for online solutions.

Start with the basics - code documentation

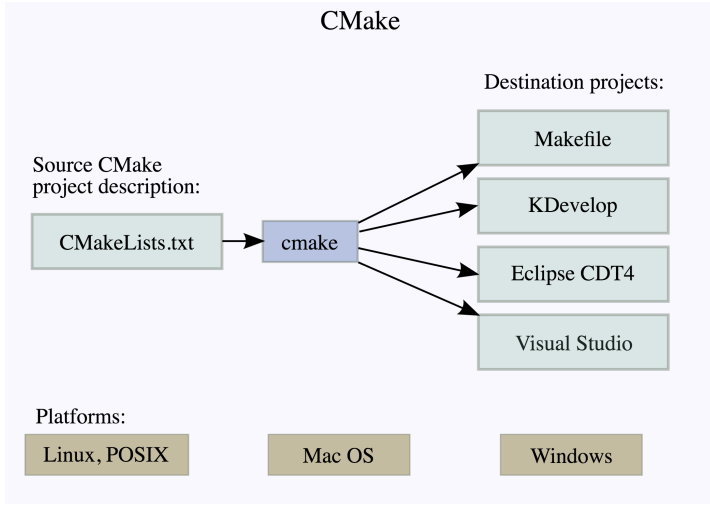
The code should be **self documented**:

```
type :: paper_state_class      !<Type - State variables of a
tracer object representing a paper material
real(prec) :: radius            !< Tracer radius (m)
real(prec) :: condition         !< Material condition (1-0)
real(prec) :: concentration     !< Particle concentration
end type
```

I've been using Doxygen, that interprets these comments and generates documentation. There are many options so we can structure the documentation to our liking. Right now I'm generating **html** and **latex** documentation.

Start with the basics - build solutions

I use Cmake to generate my solutions:



Start with the basics - Continuous Integration, tests, GIT use

- Use continuous integration (automatic builds)
- Use unit tests and grow you higher order tests
- Be paranoid about commits (my average is 8-10 a day) and branch (one per feature)

Next few months

- Continue implementation of basic framework tools
- Arrive at a barebones working model
- Test the interpolation ideas (compare bilinear to continuous in quality/performance)
- Search for turbulent diffusion models to include
- Search/implement closure models for different tracer types, starting with litter