# Parallel Design Patterns-L08

Parallel Frameworks II

Course Organiser: Dr Nick Brown
nick.brown@ed.ac.uk
Bayes room 2.13

## Framework recap

- In the last lecture we considered the design of frameworks with some common HPC examples.
  - Focused around a specific aim and parallelism a secondary aim
  - These examples are commonly used on HPC systems

- Today we are going to look at some frameworks which are first and foremost there to abstract the parallelism
  - This is a much tougher area and no one framework has (yet) gained popularity in this area

- Today we are going to think about the implementation in a bit more detail

## Skeletons

- Capture the algorithmic form common to numerous applications
- A skeleton implements a parallel pattern
  - Taking care of the lower level, tricky and often uninteresting details such as communication, synchronisation and distribution
  - The programmer fleshes these out with their application logic
  - The programmer still needs to select the correct skeleton to use
- Are often integrated with higher level languages such as functional languages

*compile :: [Char] -> [Char]*

*compile = PIPE [ readfile, lex, parse, typeCheck, genCode, writefile ]*

*readfile, lex, parse, typeCheck, genCode, writefile :: [Char] -> [Char]*

## Examples skeletons

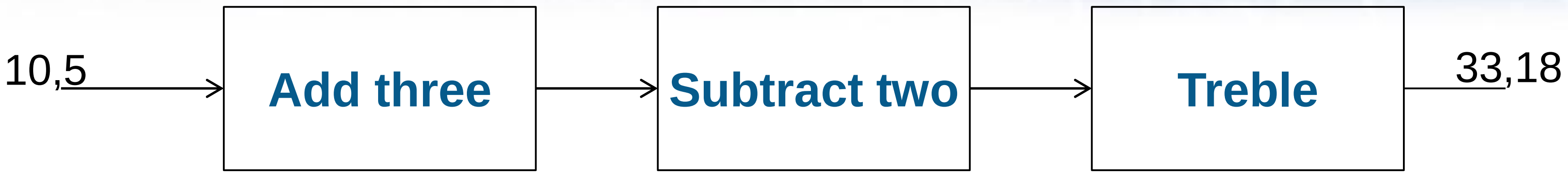
- **FARM** is also known as *master-worker* or *master-slave*.
- **PIPE** represents staged computation.
- **FOR** represents fixed iteration, where a task is executed a fixed number of times.
- **WHILE** represents conditional iteration, where a given skeleton is executed until a condition is met.
- **MAP** represents *split*, *execute*, *merge* computation. A task is divided into sub-tasks, sub-tasks are executed in parallel according to a given skeleton, and finally sub-task's results are merged to produce the original task's result.
- **D&C** represents divide and conquer parallelism.

## eSkel

- Developed by the School of Informatics
- The programmer writes C code which simply calls this as a library
- Uses MPI for underlying communications layer

http://homepages.inf.ed.ac.uk/mic/eSkel

# eSkel example pipeline



```
#include "eSkel.h"
eSkel_atom_t* addThree(eSkel_atom_t * in){
    int * data=(int*) in->data;
    data+=3;
    return in;
}
eSkel_atom_t* minusTwo(eSkel_atom_t * in){
    int * data=(int*) in->data;
    data-=2;
    return in;
}
eSkel_atom_t* treble(eSkel_atom_t * in){
    int * data=(int*) in->data;
    data*=3;
    return in;
}
```

```
int main() {
  int inputs[2]={5,10}, results[2];
  eSkel_atom_t* (*stages[3])(eSkel_atom_t*);

  stages[0]=addThree;
  stages[1]=minusTwo;
  stages[2]=treble;

  pipeline1for1(3, stages, …, inputs, ……,
                              results, ……);
}
```

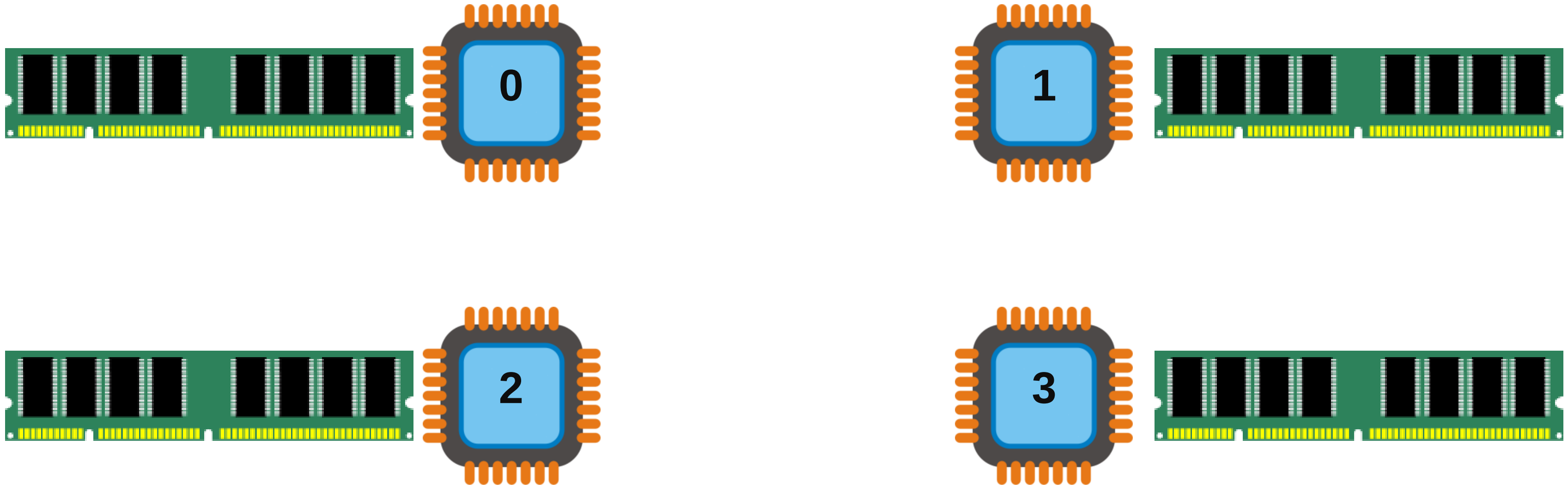
## Comments about skeletons

- A very interesting area of research
  - And one the school of informatics is involved in (*http://homepages.inf.ed.ac.uk/mic/Skeletons/index.html*)
  - A number of existing frameworks have been developed which implement similar skeletons
- Has yet to gain wide acceptance in the field of HPC
  - Issue of transparency – what is the underlying runtime actually doing in my skeleton, is it performant or scalable?
  - Requires not only a reengineering of codes but also a mind set change on behalf of the programmers (many of which are still using Fortran!)

- Aspects of these technologies can often find their way into more mainstream languages
  - Arguably Hadoop is a MapReduce skeleton

## UPC++

- UPC is an extension to C that provides the user with a Partitioned Global Address Space memory model
  - The addition of keywords into C that the UPC compiler then processes



  - Whilst this allowed the user to use an existing language, it still required an additional compiler to be installed and was inflexible for adding additional functionality

## UPC++

- UPC is an extension to C that provides the user with a Partitioned Global Address Space memory model
  – The addition of keywords into C that the UPC compiler then processes
  – Required additional compiler be installed and inflexible for adding functionality

```
shared int y[THREADS]; /* one y on each thread */
shared int a[100][THREADS]; /* one a[100] on each thread*/
shared int b[100]; /* Distributed across threads */
shared int x; /* One x in entire system */

shared int *p; /* a local item which points elsewhere */

p=upc_alloc(200);
```

- Re-engineered UPC to work with C++ and provided as a library
  – Via objects and templates

```
shared_array<unsigned long> A(100);

shared_array<int> b;
b.init(200);

global_ptr<int> ptr;
```

## UPC++

- Shows some promise but yet to gain acceptance
  - HPC programmer encouraged to use C++, might be able to wrap it
  - Have to trust that the library will do the right thing
- This idea of templates with C++ is a nice abstraction
  - Especially because additional policy could be injected
  - **`shared_array<int, even_distribution> b;`**
- The library should be made accessible so that other HPC people can go in and create specialisations of these things
  - That might abstract the partitioning and distribution of data further which UPC++ currently lacks

## Java Concurrency

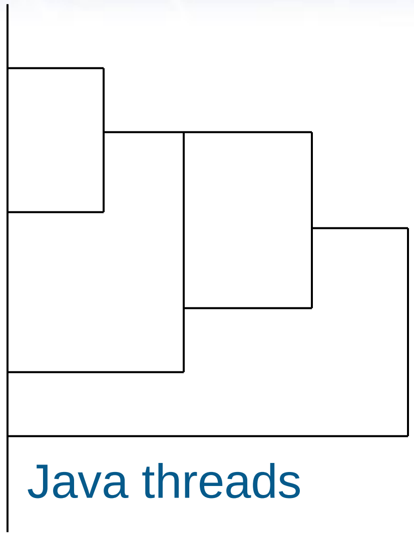
- As a framework Java carries a lot of deprecated baggage around to maintain backwards compatibility

- There is a package in Java called java.util.concurrent
  - Available since Java 1.5, so now very established
- An integral part of java, but implemented as a package
  - In other words, someone else could have written such a package
- Java has support for threads as part of the language, along with some basic mechanisms to support synchronisation between threads such as the **`synchronized`** keyword

- Whilst Java itself might not be popular in the HPC field, how Java handles these aspects finds itself into other frameworks

# Recap from Threaded Programming

- Parallelism versus concurrency?
- Java threads work on shared memory model
  - with the option to declare data to be private to a thread
- A thread is an object
- Threads are spawned in various ways:
  - Extending the **java.lang.**Thread class
  - Implementing the **java.lang.**Runnable interface
  - Using an ExecutorService and implementing the **java.lang.**Runnable interface
  - Using an ExecutorService and implementing the **java.util.concurrent.**Callable interface



*Core language*

*java.util.concurrent package*

## Spawning Threads

- Extending the **java.lang.**Thread class
  - Extend the class java.lang.Thread
  - Override the run method
  - Start a Thread
- Implementing the **java.lang.**Runnable interface
  - Implement the java.lang.Runnable interface
  - Create a new instance of type MyClass
  - Create a new Thread object wrapping the runnable

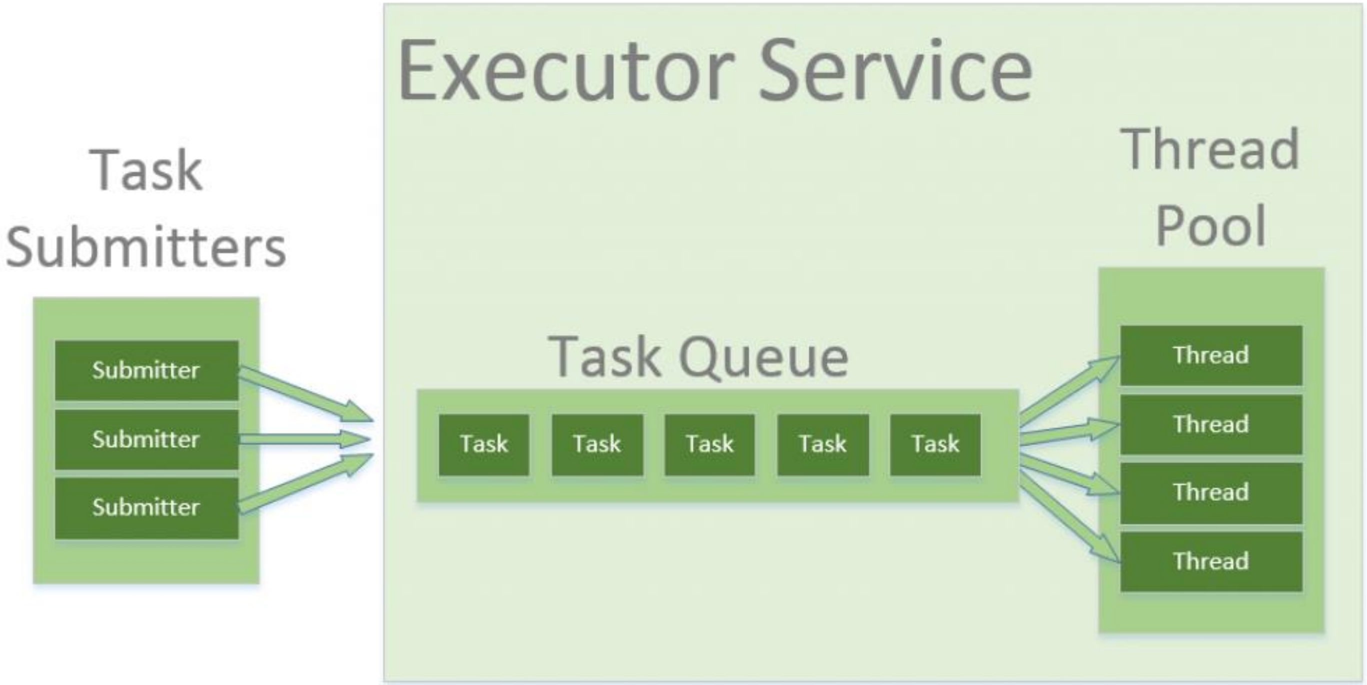
*Core language*

- ExecutorService
  - with java.lang.Runnable
    - **`run()`** method does not return values
  - with java.util.concurrent.Callable
    - more flexible

*java.util.concurrent package*

## Java Interface ExecutorService

- **ExecutorService** interface
  - Provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling,



- The submit method returns a Future that can be used to manage the task (e.g. cancel execution, wait for completion).

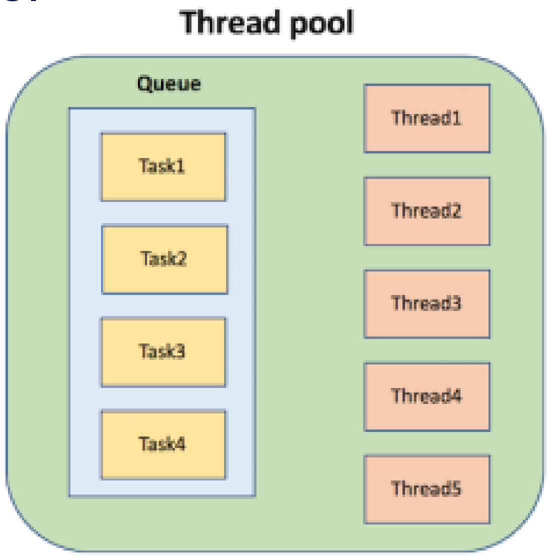
## Java Future

- **Future** interface
  - `public interface **Future**<V>`
    - `Future f=my_executor.submit(new MyClass());`

- Defined in java.util.concurrent
  - Represents the result of an asynchronous computation
  - Objects conforming to this interface are passed around a program instead of the values themselves, thereby removing the need to wait for an asynchronous computation to complete until the last minute
- Methods are provided to interrogate the progress
  - f.isDone() to test for completion
  - f.isCancelled() to test for cancelation
- Get will retrieve the result (optional max time) and cancel will cancel the request

## ScheduledThreadPoolExecutor

- Implements the ExecutorService interface
  - Instantiated with the number of threads in the pool
- Useful for resource management
  - Typically you want to map tasks to threads rather than create a new thread for each task
  - If done wrongly this can be quite a performance bottleneck
  - The API determines the best way of doing this, best way of queuing up tasks



- Useful for resource management and performance
  - Typically you want to map tasks to threads rather than create a new thread for each task
  - If done wrongly this can be quite a performance bottleneck
  - The API determines the best way of doing this, best way of queuing up tasks

## These communication frameworks

- We have seen some examples of the parallelism being abstracted from the programmer
  - Just a few examples of very many in this field
  - None of these have (yet) reached any mainstream success in HPC

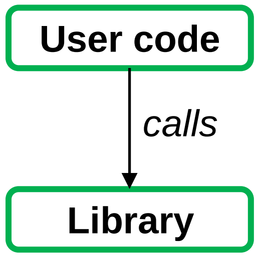
- A variety of reasons why we still use C/Fortran with MPI or OpenMP
  - Abstraction makes too many assumptions causing poor performance
  - Limited maturity
  - Limited flexibility
  - User reticence for learning new technologies

## Common ways of code reuse

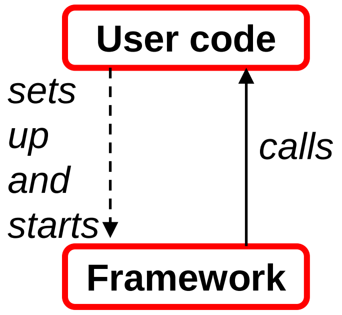
Loosely integrated

Heavily integrated

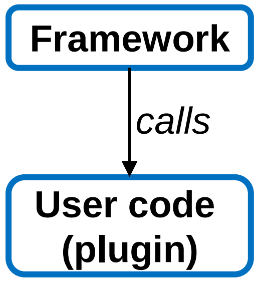
1. Calling functions in a library via an API
   - Entirety of functionality is pre-existing
   - Either has some side effect (such as writing a file/sending a message) or returns data that you then work with



2. Injecting own code into the framework
   - In C or Fortran via function pointer args to API
   - Sometimes known as a callback



3. Oriented around user provided plugins
   - Structured user defined behaviour
   - Main entry point is the framework and it calls the user plugins as appropriate



## 1 - Calling functions in a library via an API

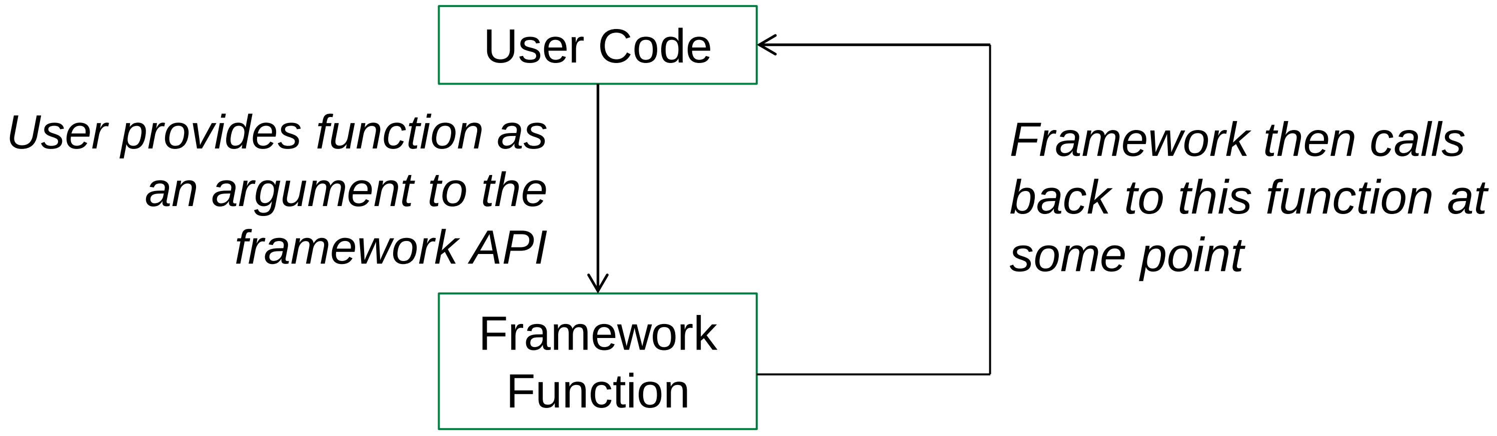
- A specific API call should do one think and do it well
  - Avoid magic arguments or mode bits
- APIs should be as small as possible but no smaller
- The API should be separate from the implementation
- Names and consistency matter
  - Self explanatory
  - Consistency in your API and with the community
- Documentation
  - Inline documentation
  - Examples
- Consider error handling

Based on http://www.cs.bc.edu/~muller/teaching/cs102/s06/lib/pdf/api-design.pdf

## API external and internal data types

- APIs can often require structured data as arguments
  - As a C structure or Fortran derived type
- It can be quite common to differentiate between the data types in your external interface and those used internally
  - Avoids exposing the internals of your framework implementation to the user (can avoid miss use.)
  - Can provide increased flexibility, where your framework implementation can change independently of the user code that calls it
- However a “glue layer” is required to convert from the external data types to internal data types
  - Can be a source of bugs and increased complexity
  - Might have a performance impact

## 2 - Injecting code into the framework


User Code
User provides function as an argument to the framework API
Framework Function
Framework then calls back to this function at some point

- User function can be either mandatory or optional, overriding some sensible default provided by the framework
- Can be a good way of providing complex functionality to the framework
  – The alternative being a very complex API
- In C and Fortran implemented via function pointers

## 3 - User provided plugins

- A plugin performs some high level behaviour
  - Often a collection of many functions
  - Often some sort of description about the module
- Independent of each other
- The framework defines a standard plugin template
  - Which each user provided plugin must follow
- The framework provides ways in which these can be managed
- Any inter-plugin communication follows a standard, organised, approach
- Plugins either present during compilation or plugged in at runtime

## MONC

- A cloud resolution model employed by UK weather and climate communities
  - Space is decomposed via the geometric decomposition pattern
  - Time progresses in time steps
- A key requirement was that scientists could easily add, remove or modify functionality
  - Modelling of specific systems requires code changes
  - But how to do this in a way that avoided poor code polluting the rest of the model?
- Architected as a set of components (plugins)
  - With optional call backs for initialisation, timestepping and finalisation

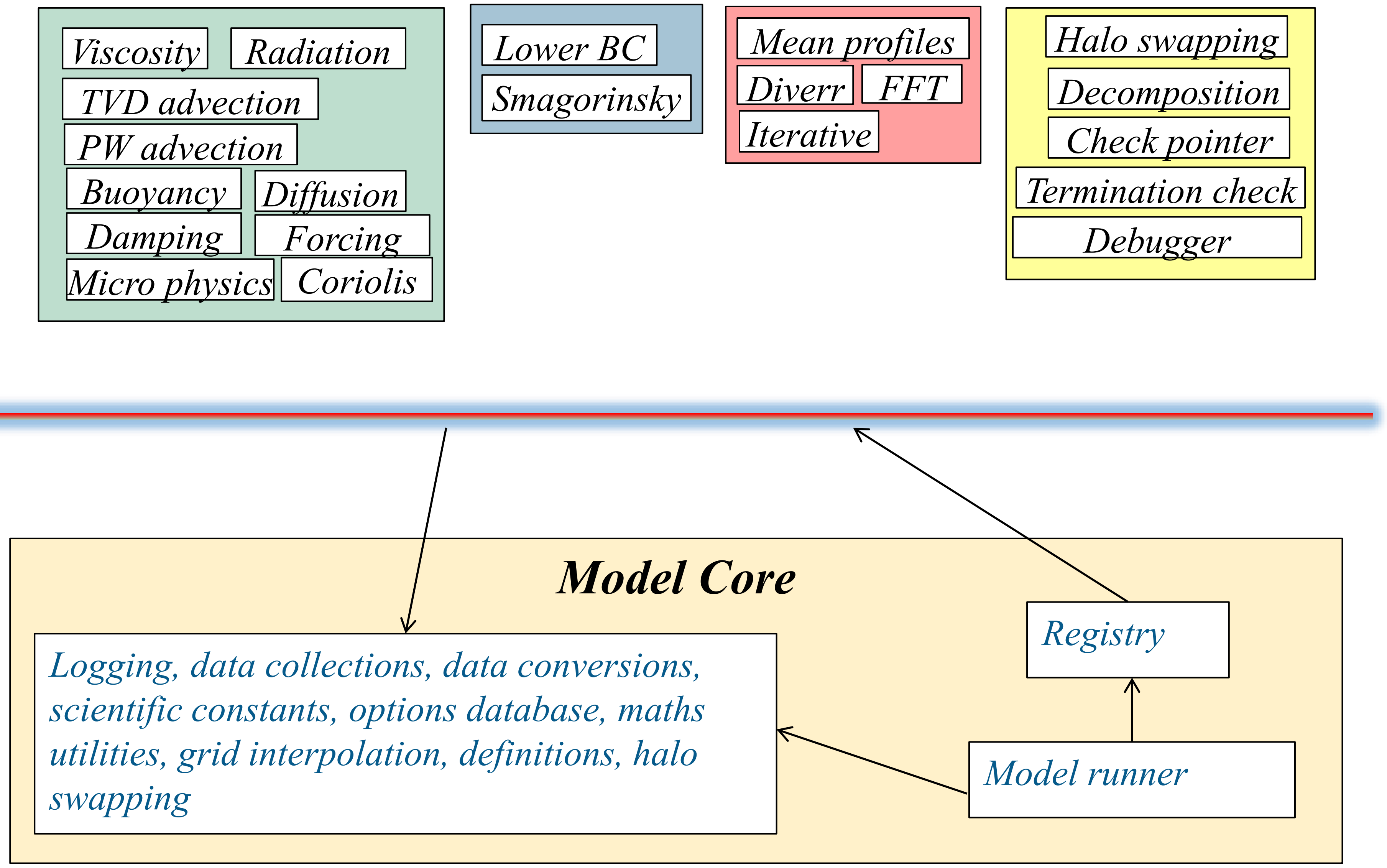
## MONC plugin template

```
type(component_descriptor_type) function test_get_descriptor()
    test_get_descriptor%name="test_component"
    test_get_descriptor%version=0.1
    test_get_descriptor%initialisation=>initialisation_callback
    test_get_descriptor%timestep=>timestep_callback
end function test_get_descriptor
```

```
subroutine initialisation_callback(current_state)
    type(model_state_type), target, intent(inout) :: current_state
    ………………
end subroutine initialisation_callback
```

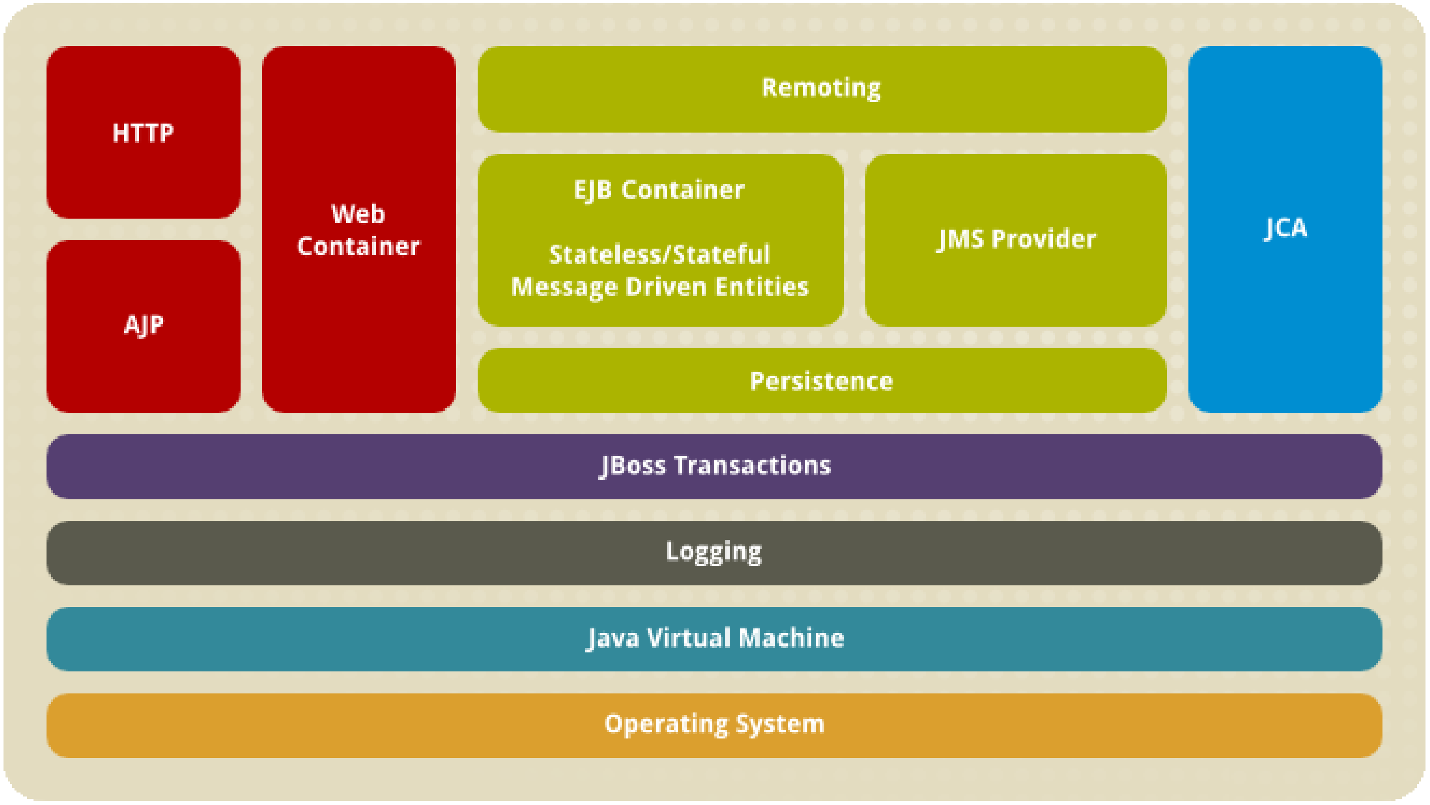
```
subroutine timestep_callback(current_state)
    type(model_state_type), target, intent(inout) :: current_state
    ………………
end subroutine timestep_callback
```

# MONC Components



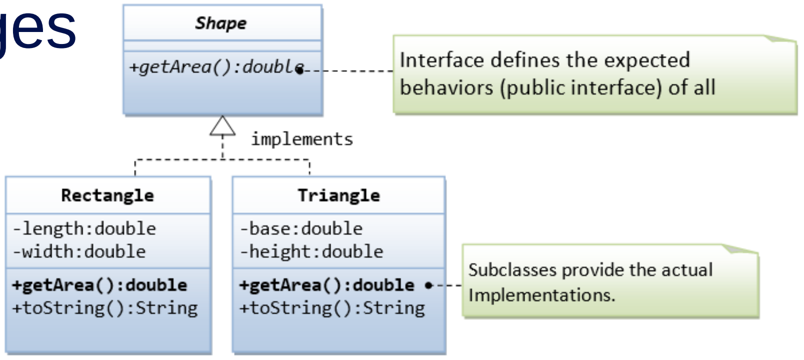
## JBoss (or Wildfly)

- An application server that allows one to easily publish J2E web applications
  - User applications are packaged into a predefined format and deployed. As well as the code themselves they also contain configuration files



# So how do we write a framework?

- In C++, Java or other OO languages there are various techniques for generic programming
  - Most important here is inheritance
  - Extend an existing class or implement an interface



- In C, Fortran (which tend to fit more closely with MPI and OpenMP), a useful idea is that of function pointers
  - Pass a function to a function, instead of just variables
  - As a very basic example, you could have a function *invoke* which takes as arguments the integers *a* and *b,* and a mathematical function to perform on these
  - C variants (C++, Objective C) and “modern” Fortran (2003 onwards) also have OO features

## Function Pointers in C

- To declare the general function
  - `int invoke(int (*fn_pointer)(int, int), int a, int b);`
- To declare the function that will be run
  - `int add_fn(int a, int b) {…}`
  - `int subtract_fn(int a, int b) {…}`
- To call the function
  - `int result = invoke(add_fn, 2, 8);`
  - `int result2 = invoke(subtract_fn, 2, 8);`

## Example in C

```
#include <stdio.h>

int add_fn(int a, int b) {
  return a+b;
}

int subtract_fn(int a, int b) {
  return a-b;
}

int invoke(int (*fn_pointer)(int, int), int a, int b) {
  return (*fn_pointer)(a, b);
}

int main() {
  int result = invoke(add_fn, 2, 8);
  int result2 = invoke(subtract_fn, 2, 8);
  printf("%d and %d\n", result, result2);
  return 0;
}
```

# Generic arguments in C

- What about supporting both:
  - `int add_fn(int a, int b) {…}`
  - `int abplusc_fn(double a, double b, double c) {…}`

```
#include <stdarg.h>

int add_fn(va_list valist) {
  return va_arg(valist, int)+va_arg(valist, int);
}

int abplusc_fn(va_list valist) {
 return (int) (va_arg(valist, double) * va_arg(valist, double)) + va_arg(valist, double);
}

int invoke(int (*fn_pointer)(va_list), ...) {
  va_list valist;
  va_start(valist, fn_pointer);
  int result = fn_pointer(valist);
  va_end(valist);
  return result;
}

int main() {
  int result = invoke(add_fn, 2, 8);
  int result2 = invoke(abplusc_fn, 2.3, 8.5, 7.4);
  return 0;
}
```

## Function Pointer in Fortran

- To define the "function pointer"

  -
    ```
    interface
      integer function fn_pointer(a,b)
        integer, intent(in) :: a, b
      end function fn_pointer
    end interface
    ```
- To declare the general procedure

  -
    ```
    subroutine invoke(provided_function, x, n)
      integer, intent(in) :: n, x
      procedure(fn_pointer) :: provided_function
    end subroutine invoke
    ```
- To call the procedure
  - `call invoke(add_fn, n, x)`
  - `call invoke(subtract_fn, n, x)`

# Example in Fortran

```
module b
  implicit none

  interface
    integer function fn_pointer(a, b)
      integer, intent(in) :: a, b
    end function fn_pointer
  end interface
contains
  integer function add_fn(a, b)
    integer, intent(in) :: a, b
    add_fn =a+b
  end function fn1

  integer function subtract_fn(a ,b)
    integer, intent(in) :: a, b
    subtract_fn =a-b
  end function fn2

  integer function & invoke(provided_function, a, b)
    procedure(fn_pointer) :: provided_function
    integer, intent(in) :: a, b

    invoke=provided_function(a, b)
    end function invoke
end module b
```

```
program a
  use b
  implicit none

  integer :: result, result2
  result =invoke(add_fn, 2, 8)
  result2=invoke(subtract_fn, 2, 8)
  print *, result, result2
end program a
```

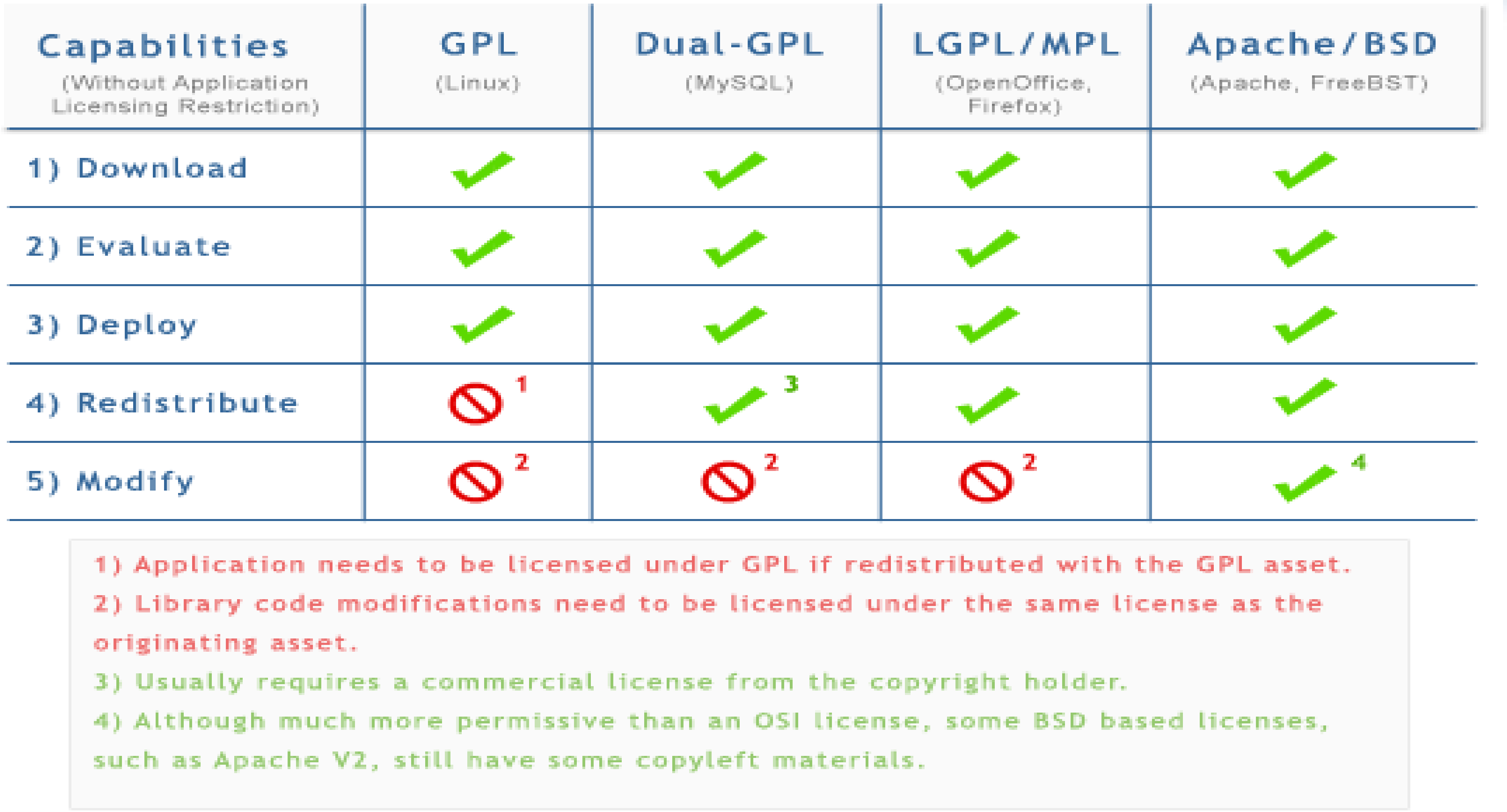
## Compiler Support for Fortran 2003

- The example on the previous slides is valid Fortran 2003
- Not all compilers are fully compliant
  - With different levels of maturity

- Current Support for Procedure Pointers
  - Derived partially from *Fortran 2003 status,* in Fortran Wiki http://fortranwiki.org/fortran/show/Fortran+2003+status (Accessed 2015-01-16)

| Compiler | Support |
|---|---|
| Absoft | Y |
| Cray | Y |
| GNU (gfortran) | Y |
| g95 | Y |
| HP | Y |
| IBM | Y |
| Intel | Y |
| NAG | Y |
| Oracle | Y |
| Pathscale | N |
| PGI | Y |

# Licencing



| Capabilities (Without Application Licensing Restriction) | GPL (Linux) | Dual-GPL (MySQL) | LGPL/MPL (OpenOffice, Firefox) | Apache/BSD (Apache, FreeBST) |
|---|---|---|---|---|
| 1) Download | ✓ | ✓ | ✓ | ✓ |
| 2) Evaluate | ✓ | ✓ | ✓ | ✓ |
| 3) Deploy | ✓ | ✓ | ✓ | ✓ |
| 4) Redistribute | 🚫 [1] | ✓ [3] | ✓ | ✓ |
| 5) Modify | 🚫 [2] | 🚫 [2] | 🚫 [2] | ✓ [4] |

1) Application needs to be licensed under GPL if redistributed with the GPL asset.
2) Library code modifications need to be licensed under the same license as the originating asset.
3) Usually requires a commercial license from the copyright holder.
4) Although much more permissive than an OSI license, some BSD based licenses, such as Apache V2, still have some copyleft materials.

http://programmers.stackexchange.com/questions/105344/is-there-a-chart-for-helping-me-decide-between-open-source-licenses

Lots more information at http://choosealicense.com/licenses/

## Summary

- Most languages provide mechanisms for writing generic code
  - These mechanisms can be used to implement parallelism in a way that is independent of the application
  - Where is my split between mechanism and policy, and what does this interface look like (i.e. is it powerful enough)?
- With clean interfaces, you can write “good” code first, and then optimise for performance
- A variety of design decisions can impact whether a framework is generally useful or not
  - And additional non-technical issues too