# Mercury, a reflective middleware for automatic parallelization of Bag-of-Tasks

João Nuno Silva
INESC-ID / IST
Rua Alves Redol, 9
Lisboa, Portugal
joao.n.silva@inesc-id.pt

Luís Veiga
INESC-ID / IST
Rua Alves Redol, 9
Lisboa, Portugal
luis.veiga@inesc-id.pt

Paulo Ferreira
INESC-ID / IST
Rua Alves Redol, 9
Lisboa, Portugal
paulo.ferreira@inesc-id.pt

## ABSTRACT

Automatic aprallel application adaptation

Bag opf tasks,

multiple object creation method invocations. Parallelization of the execution of such methods.

Execution of such methods in different platforms : multiprocessor/ multicore or remote objects in remote machines (cluster, cloud, or distributed computing systems.

Selection of best target environments.

## 1. INTRODUCTION

There has been an increase of use of scripting languages (such as python) on the management of scientific computing jobs. Pyhton has been used to interact with grid enabled simulations, as with Ganga [3], or access and manipulation of scientific data sets as in PyRAF [4]. Furthermore the performance obtained when using Python is on par with some other commonly available programming languages or environments [1, 6], making a suitable language for scientific processing.

The use of python is not limited to the invocation of serial simulations. With a suitable middleware or libraries it is possible to take advantage of multiprocessors or clusters of computers. For instance Star-P [5] offers a set of APIs to data and code distributions, while MPI [2] have the usual functions for task creation and synchronization, and data transfer.

For large scale projects, besides the use of parallel kernel (such as BLAS parallel implementations) the use of MPI is the best way for the data and work distribution. Not only the programmer is able to take advantage of the remote resources, but also is hidden from network and computing platforms heterogeneity.

For the execution pure of bags-of-tasks or embarrassingly parallel jobs, the use of such libraries may be overkill. With an increased complexity on the parallel implementation, the gains can not be enough to justify it.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

```
1  for i in range(1000):
2    inputData = getTaskInput()
3    objectList[i] = processinObject(inputData)
4    objectList[i].processData()
5
6  for i in  range1000):
7    outputresult = objectList[i].getResult()
8    process(outputResult)
```

**Figure 1: Tipical bag-of-tasks problem serial version**

In order to parallelize the application shown in Figure 1 (allowing all `processData()` methods to execute concurrently) using MPI, some modifications had to be made: i) the identification of the master and slave tasks, ii) the sending of the input data, and iii )the receiving the results. To accomplish that, the overall organization of the code would have to be changed. These modifications would require the programmer to know the MPI API and would introduce error prone code.

To reduce the modification effort necessary to transform a serial application in structure to the example presented in Figure 1 into a parallel Bag-of-tasks we developed Mercury. Mercury is a middleware that allow the parallelization of independent object methods, allowing their concurrent execution on different local threads or different remote computers.

Our proposed solutions transparently transforms an applications with a processing cycles where in each iteration a different set of data is processed. With no user intervention the lengthy tasks executed on each iteration of the cycle is executed on different thread. This thread can be executed locally, in the case of a multiprocessor (or multi-core) computer, or on remote computers.

The programmer must write its data processing application in the form of the example in Figure 1, and state on XML configuration file the class and method that can be executed concurrently.

Our middleware will be responsible for spawning the necessary threads, and synchronize the invocation of the abjects methods. In the previous example, only after the conclusion of the `processdata`, the corresponding `getResult()` method can be executed.

Mercury also allows different execution environments for the parallelized methods. A set of adaptation classes al-

low the execution of the methods on different threads on the same multiprocessor/multicore computer or the remote execution of faster computers. The indication of what parallel execution environments are available is also made in a descriptive way. The selection of the best location to execute the parallel code is made during runtime and taking into account both the code requirements and the computing resources availability.

The distribution of work among several computers or processors can now be done using libraries such as MPI, but require the programmer to know their API. With run-time code this need disappears, as the work distribution code is inserted in the correct places during application execution. Furthermore it becomes easier to add or change the possible execution environments.

The proposed solution uses metaclasses, allowing the modification of the code to be done on run-time, without any need to transform and recompile the source code. The developed metaclass intercepts all class creation and modifies the implementation of those that are to be parallel, without any user intervention: the user must only state what classes have methods that can be executed concurrently with the rest of the code.

In the next section we present some technologies and systems that address similar issues as our work (parallel execution, reflection and parallel methods definitions). In sections 3 and 4 we describe the architecture and implementation (respectively) of our system. Finally we show performance and functional evaluation as well as the conclusions and future work.

## 2. RELATED WORK

mobile midlleware
p2p programming systems
aspect oriented,
reflection, meta-programming

## 3. ARCHITECTURE

The architecture of Mercury is closely mapped to its mains functionality: load the various class adaptors, load information about parallel classes, transform parallel classes, and adapt transformed classes to the various execution environments.

The organization and linking between the several components is shown in Figure 3.

The **Application Transformer** module replaces the entry point of the program being transformed, loads the transformation code and initiates the original application transformation. This module starts loading and creating the **Adaptor Loader**, **Metaclass Loader** modules, then initializes the necessary data structures and starts the **Application Loader**.

The adaptor Loader reads an configuration file (the **Adaptors List** on Figure 3) stating the available **Adaptors**, loads each **Adaptor** code, and creates and registers the corresponding class. For each available computational resource (threads and other parallel execution environments with suitable middleware) there is one **Adaptor**. Each of these **Adaptors** are responsible for the creation and termination of the concurrent tasks on one infrastructure.

The **Metaclass Loader** reads the **ClassTransformer Metaclass** code, creates it and registers it for use when

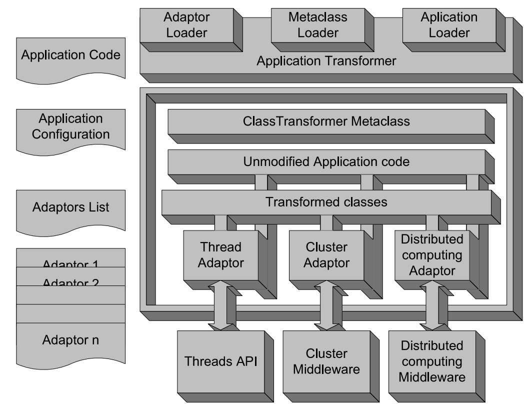


**Figure 2: architecture**

**Aplication Loader** starts loading and processing the **Application Code**, taking into account the name of the parallel classes and methods stated on the **Application Configuration**. The way the application is transformed and how this is implemented will be presented in sections 3.1 and 4

During the application execution the classes and objects organization is the one shown in Figure 3.

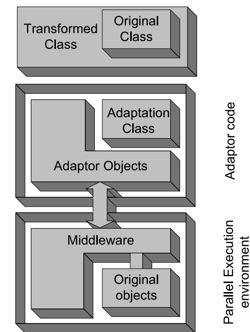


**Figure 3: Transformed classes organization**

After loading the application besides the original classes (and their objects), some more auxiliary classes are created and instantiated.

The **transformed class** is a wrapper for the **original class**, to which it has one reference. When an instance of the **original class** was supposed to be created, it is responsibility of the **transformed class** to decide what kind of Adaptation object to create. No instances of this **transformed class** exist during execution: only instances of the **original class** (locally or on remote computers) and instances of the **adaptation classes**.

The **adaptation objects** server the purpose of handling all particularities of the underlying **Middleware** parallel execution mechanisms: threads, or processes on remote computers. These objects act as proxies, being responsible for redirecting all calls to the original objects (instances of the **original classes**), and handling all synchronization issues. During execution unmodified objects interact with **adaptor abjects** transparently.

### 3.1 execution

The code loading process and initialization of a trans-

formed application is shown in Figure 4.

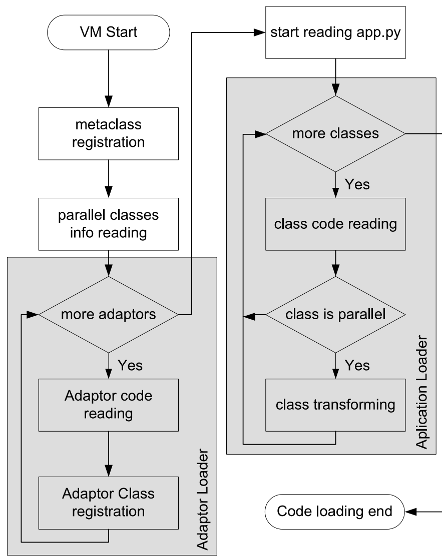


**Figure 4: Aplication start fluxogram**

The first steps are straightforward. First the supplied metaclass is loaded and registered for latter use: when the information about the parallel classes is read. The supplied **Application Configuration** files must contain for each classes that have parallel methods its name, and the name of those methods. This information is stored and will be used when loading the application class, and when invoking their different methods as will be shown in the next section.

The **Adaptor Loader** Module is responsible for loading the **Adaptor List** file and create the **Adaptor classes** refereed on that file. This configuration file contains the list of classes. From those names the **Adaptor Loader** gets the file with the class implementation and imports it. From this moment forward the corresponding class is available for use. The names of all **Adaptor classes** are stored in a list.

The last step before the application execution is it loading from disk and transformation. Inside the **Application Loader** whenever a class is loaded from disk it is verified if its name was read from the **Application Configuration** file. If the name was present in the configuration file, a class transforming should occur, on the other hand ordinary classes are created normally.

The python mechanisms used in the interception of the classes loading and the classes transformation are presented in the next section.

# 4. IMPLEMENTATION

In Section 3.1 we made an overview on how the application transformation is made. The implementation details and how exactly the presented steps are carried out is now presented: i)interception of the class load, ii) class transforming and iii) adaptor implementation.

## 4.1 Class loading interception

The class transformations are performed by a custom metaclass. While python allow the use of a global metaclass some libraries do not allow its classes to be created by a metaclass. So, it was necessary to after loading each class code (but before actually creating it) to check if it was necessary to invoke a metaclass. Before starting reading the application code our initialization code installs a custom import function. Whenever a python file was included, it is our import function that is executed, whose pseudo-code is shown in Figure 5.

```
1 def my__import__(fileName):
2   mod = __old__import__(fileName)
3   for name, object in mod:
4     if isClass(object):
5       if (name in paralelClasses):
6         mod[name] = classTransformer.
              newClass(className)
7   return mod
```

**Figure 5: Custom file import**

On line 2, the original import function load all the file code, which is stored in the `mod` variable. Then, on line 3, for every object loaded (that includes constants, function, and classes) we check if it is on class. If the class was referred in the **Application Configuration** file (its name is stored in the `paralelClasses`) we replace it for a new class created by our metaclass named `classTransformer`. After the loading of the python file, object containing the classes

## 4.2 Class transforming

As stated earlier the transformation of the parallel classes is performed by a metaclass. In python, classes and metaclasses are first-class objects, and as such have the ordinary class methods: `__new__` where the instances are actually created, and `__ini__` used to initiate the state of its instances.

As we want to intervene on the actual creation of the classes the `__new__`method should be defined. The actual implementation of the **classTransformer** metaclass is shown in Figure 6.

```
1 class classTransformer(type):
2   def __new__(cls, name, bases, dct):
3     oldclass = newClass(name+"old", cls)
4     proxyclass = newClass(name,
          transformedClass)
5     proxyclass.originalClass = oldclass
6     return proxyclass
```

**Figure 6: `classTransformer` metaclass pseudo-code**

On line 3 we build a copy of the original class but with a different name. From this point forward, the original class can be accessed globally by its new name, or locally to this `__new__` method through the `oldclass` variable.

On the following lines name a copy of pre-existent class (**transformedClass**) is made and a link to the original class is set ans a class attribute.

The `__new__` method concludes returning a reference to a copy of the **transformedClass** class. From this point forward, on the original code, whenever the programmer created an instance on the original class, that object creation will be handled by a copy of the **transformedClass** class.

The transformed class **transformedClass** can be seen as an object factory. When trying to creating an instance of this class the return objects will belong to on of the **adaptor**

**classes**. Also in this case the method `__new__` (shown in Figure 7) is executed.

```
1 class transformedClass (object):
2   def __new__ (cls, *args):
3     adaptorClass = select Adaptor Class()
4     proxyObj = adaptorClass(cls.
          originalClass, paralelClasses[
          originalClassName], *args)
5     return proxyObj
```

**Figure 7: Transformed Class**

On line 3 the best adaptor is selected. This has to be done to optimize the allocation of available resources (processors, memory, ...) taking into account the object computational requirements. The way the best **adaptor** is chosen is out of scope of this paper, but possible solutions will be presented on the Conclusion.

On the next line, an instance of the selected **Adaptor class** is created. The constructor receives as parameters the **original class**, the list of **parallel methods** (those from the **original class** that can be executed concurrently) and the original arguments that are to passed to the **original class** constructor.

This method return a proxy object, that is an instance of an **adaptor class**, that forwards all method call to the original objects.

During normal program operation, different instances of the same class can live concurrently and execute on different computers.

## 4.3 Adaptor implementation

The Adaptor classes are responsible a series of management activities: i) evaluation of the adequacy of the execution environment to the classes parallelized, ii) creation of **original objects** on the target platform iii) proxying of the methods invocation, and iv) synchronization of invoked methods.

The fundamental **Adaptor class** is the one that takes advantage of local multiprocessors/multicores to allow the efficient concurrent execution of several **original classes**. Besides allowing the concurrent execution of some methods, this **adaptor class** must also block other method invocations until parallel methods terminate. Other adaptor classes, are built on/with this classes to accomplish the same objectives, but must create the **original objects** on different computing infrastructures.

In figures 8 and 9 we present the **Thread Adaptor** pseudocode.

```
1 def __init__(self, originalClass,
      _paralelMethods, *args):
2   self._proxiedObject=originalClass(*args)
3   self._lock = threading.Lock()
4   self._paralelMethods = _paralelMethods
```

**Figure 8: Thread Adaptor Class**

The initia;lizatio of the Thread Adaptor is shown on Figure 8. The **adaptor** will have a reference to an instance on the **original class**.This object is created on line 2 and will be responsible for actually perform all computational tasks, not the **Adaptor**.

The lock created on line 3 will be used to guarantee that no methods executed by the same object will execute concurrently. The list with the parallel methods (`_parallelmethods` is necessary to know which methods should be executed in a separate thread from the rest of the application.

In order to synchronize method invocations and launch threads when necessary, it is necessary to intercept all method calls. The necessary code, present in the **adaptor** object, is shown on Figura 9.

```
1 def __getattr__(self, attr):
2   if type(attr) is MethodType:
3     self._name.append(attr)
4     if attr in self._paralelMethods:
5       return self.__invokeParalel__
6     else:
7       return self.__invokeSerial__
8   else:
9     self._lock.acquire();
10    ret = getattr(self._proxiedObject,
          attr)
11    self._lock.release();
12    return ret
13
14 def __invokeSerial__(self, *vargs):
15   self._lock.acquire();
16   methodName = self._name.pop()
17   method = getattr(self._proxiedObject,
          methodName)
18   ret = method(*vargs)
19   self._lock.release()
20   return ret
21
22 def __invokeParalel__(self, *vargs):
23   self._lock.acquire();
24   self._thread =
25     threading.Thread(self.
           __paralelCode__, vargs)
26   self._thread.start()
27
28 def __paralelCode__(self, *args):
29   methodName = self._name.pop()
30   method = getattr(self._proxiedObject,
          methodName)
31   method(*vargs)
32   self._lock.release()
```

**Figure 9: Threads Adaptation Class**

Before the execution of any method or access to an object attribute, the method `__getattr__` is called. This method returns either the value of the attribute or a a reference to the method object, as in python methods are first class objects. If the access is to an attribute, the access is forward to the original object (in linee 9-11) and guarded by a lock.

If it is a method call, two cases are possible: parallelizable methods or not. In either case the method returned does not belong to the **original object** but to the **Adaptor** (lines 5 and 7). Before returning references to these methods (`__invokeSerial__` or `__invokeParalle__`) the name of the called method is pushed to a stack (line 3).

The `__invokeSerial__` method gets the name of the method to being called (line 15), obtains from the **original object** the actual method (line 16), and invokes it (line 17).

The `__invokeParalle__` acts in a similar way but on a

different thread. The code that gets a reference to the called method and its execution (method `__parallelCode__` on lines 26-30) runs on a different thread. This thread is started on lines 22-24 inside the `__invokeParalle__` method.

The synchronization that guarantees that the execution of a transformed objects is the same as the one of it unmodified version is performed by the various acquires and releases of the lock in various places: i) when accessing the attributes of the original object (lines 9-11), ii) during the serial execution of the methods (lines 15-19), and iii) when the parrallel methods execute the lock is aquired before starting the thread (line 23) and release at its end (line 32).

# 5. EXECUTION

# 6. EVALUATION

7 segundos para 1000 threads/objectos parallelos

# 7. CONCLUSIONS

# 8. REFERENCES


[1] O. Bröker, O. Chinellato, and R. Geus. Using python for large scale linear algebra applications. *Using Python for large scale linear algebra applications*, 21(6):969 – 979, 2005.

[2] L. Dalcin. Mpi for python - python bindings for mpi. http://code.google.com/p/mpi4py/.

[3] U. Egede, K.Harrison, R. Jones, A. Maier, J. Moscicki, G. Patrick, A. Soroko, and C. Tan. Ganga user interface for job definition and management. In *Proc. Fourth International Workshop on Frontier Science: New Frontiers in Subnuclear Physics*, Italy, September 2005. Laboratori Nazionali di Frascati.

[4] S. T. S. Institute. Pyraf home page. http://www.stsci.edu/resources/software_hardware/pyraf.

[5] I. Interactive Supercomputing. Star-p overview. http://www.interactivesupercomputing.com/products/.

[6] J. K. Nilsen. Montepython: Implementing quantum monte carlo using python. *Computer Physics Communications*, 177(10):799 – 814, 2007.