The PyAALib Framework and The Profiler Interface: How to Develop and Debug a Massively Parallel Application based on Dynamic Plug-ins?

Romeu A. Pieritz* and Claudio Ferrero
SciSoft Group - ESRF – European Synchrotron Radiation Facility
BP 220 - 38043 – Grenoble Cedex 9 - France

This project concerns the development of a framework based on pure object oriented concepts to ensure asynchronous communication between threads (thread safety) and external processes, thus allowing the development of massive parallel applications mixing different programming languages and platforms for heterogeneous developers. A massive parallel application can be entirely designed and developed based on the dynamic multi-thread plug-in architecture implemented in the framework. Any code coming from different software sources can be adapted to the framework as a dynamic multi-threaded plug-in. It can be deployed in Java environments using the "Standalone Generic Application Skeleton" package without the need of installing Python or Jython. The Profiler GUI is a standard option available in the "ALApplication" module to plot a "timeline" of concurrent actions during the code execution. It features an interactive interface displaying asynchronous actions during the runtime of massively parallel applications. The interface shows dynamical hyperlinks to monitor the independent output of each component used during the execution of the application. In addition, it allows inspecting and assessing individually the performance of each plug-in to optimize and control the code as well as to verify the expected results. It can be used in association with other parameters like tag-debug and verbose options.

*Address For Contact: pieritz@esrf.eu

Introduction

The PyAALib project is a Python implementation of the Asynchronous Action Framework (AALib – http://aalib.sourceforge.net) based in "Pure Object Oriented Concepts" [1993] to develop Massive Parallel Applications [2000]. The AALib framework is a well tested open source framework to be used to develop asynchronous multi-thread applications. It implements the basic plug-in architecture to be used with the special PyAALib action objects and standard python classes. This mechanism allows the rapid development of dynamic and modular software to facilitate the continuous improvement. Modern software engineering techniques based on unit test are used to guarantee the quality of each part of the code during the development cycle. The python library version of the AALib is full compatible with all standard frameworks using C, C++, Python and Java. A Massive Parallel Application can be entirely designed and developed based on dynamic multi-thread plug-ins. Also, any legacy code can be adapted and used as a dynamic plug-in. This architecture allows the development of parallel online data analysis code for smart experiments; the parallel processing of online data using different models, comparing results; and it can be used to process a enormous amounts of data.

Main Concepts and Modules

The main concept in the framework is the "Action" (main class ALAction). It is a complete "atomic" independent "thread control system" and "exception manager". It contains all the control structure to drive the associated code, added or connected, by a "callback" system. The "Basic Callback Pattern" [1995] connects any code from any library and framework in synchronous and asynchronous mode. The callback design pattern is implemented in the kernel of the main classes of the framework.

The ALAction inspects any exception or controlled interruption of the thread (by the exception inspector) and a "workflow" pattern can be connected to the main loop of the action. All complex actions are constructed from that: ALActionController, ALActionManager, ALActionProcess, ALActionSubProcess, ALActionManagerProcess, ALActionSet, ALActionMethodTimer, ALActionCluster, etc… These different types of actions can be used individually or combined to construct more complex execution behaviors.

The ALAction is the origin of the main dynamic concept called "ALPlugin". It is a full multi-thread "dynamic code container" to be loaded, compiled and executed on runtime by a "Plug-in Factory" engine base on the "object factory pattern" [1995]. This plug-in factory is a runtime database of objects constructed after scan the disk to find, import, compile and instantiate the plug-in code.

The data binding standard based on XML scheme [2002] is natively implemented in the framework and a more complex plug-in can be constructed directly from XSD files and XML data. It generates and compiles in runtime the python code defined by the XSD file and instantiate an object from the associated xml data input. It is used into the plug-in framework to generate the input or output classes to manage any kind of data from or to xml files.

A powerfull "ALManagerProcess" class abstraction is implemented to control external processes independently of the programming language. The code interprets directly the task process list from the host machine and conserves a direct control of the state of the child process. It avoids the "hang" effect when an external process is not responding, keeping the control over the external process even when a "pipe" crashes or overflows (pipe data exchange channel between the processes) .

A basic application class (ALApplication) defines a "skeleton" to RAID ("Rapid Application Development and Deployment"). It manages all logs, resources and actions creating and connecting all code with the heterogeneous operational system in a single line of code. All signals and events are controlled and monitored by the main code inspector (main loop of actions). The main architecture to generate and control the multi-thread actions is encapsulated in the framework and it simplifies the code development. The plug-in factory pattern is extensively used on the basic application skeleton to implement the code generation and object management during the runtime. It dynamically creates and manages the mechanism to load objects and data.

A "command line interface" based on a xml file is associated to the ALApplication to map the commands, help and manual text to any plug-in or code of the application. The file allows a central repository of all human readable information associated with the application. It is used to generate an interface for all parameters controlling the different elements of the code.

An extension of the Basic Application skeleton implements a complete internet server/client architecture based on the XML-RPC standard ("Remote Procedure Call based on XML file exchange") [2001]. It is created and configured by the multi-thread plug-in factory implemented by the library. A special mechanism is used to exchange the runtime objects natively and transparently to the user. It is a high level interface to manage the complexity of the object translation to XML definition and its management.

The full framework is continuing tested and validated by a "Test Framework". The test framework is an abstraction to dynamically execute, control and report all results of the individual unit tests and test sets. It is used to validate all code implemented in the framework for all supported operational systems and hardware architectures. The test framework implements debug and profile tools available for all classes and code using the AALib framework. It can be operated dynamically on runtime from any part of the code using the "static method concept" [1993].

The "Tag debug" scheme was introduced in the framework to allow the developer to debug his own plug-in when running in parallel all other plug-ins in the application. The new parallel scheme outputs only the information required by the user using the verbose or debug interfaces. Also, the internal AALib debug information is output only under user control. It represents "no pollution" in the output screen and the log file contains only the debug information required by the user.

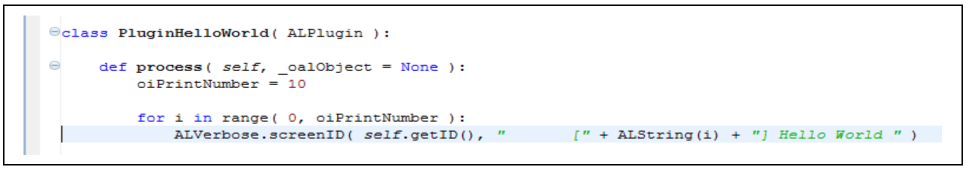
The PyAAlib-JyAAlib framework is full compatible with Python and Jython (http://www.jython.org – Java code interpreter). This compatibility with Jython permits mix Java objects inside plug-ins and defines a simplified folder structure for code deployment based on a standard PyAALib-JyAAlib "jar" file. The target system does not need any longer a previous distribution of Python or Jython. It requires only a modern Java Virtual machine because all Jython interpreter code is distributed with the PyAALib-JyAALib jar file.

The Massive Parallel "Hello World" Example

The "hello world" example demonstrates the simplicity of programming a massive parallel code using the PyAALib-JyAALib framework. The example prints on the console 1024 times a batch of 10 "hello world" (10240 prints in total). The code is composed by three main modules: i) one plug-in to print "hello world" 10 times (figure 1) ; ii) one plug-in to define and control a "cluster" of 1024 "hello world" plug-ins (figure 2) and iii) one ALApplication to define and manage the software (figure 3).

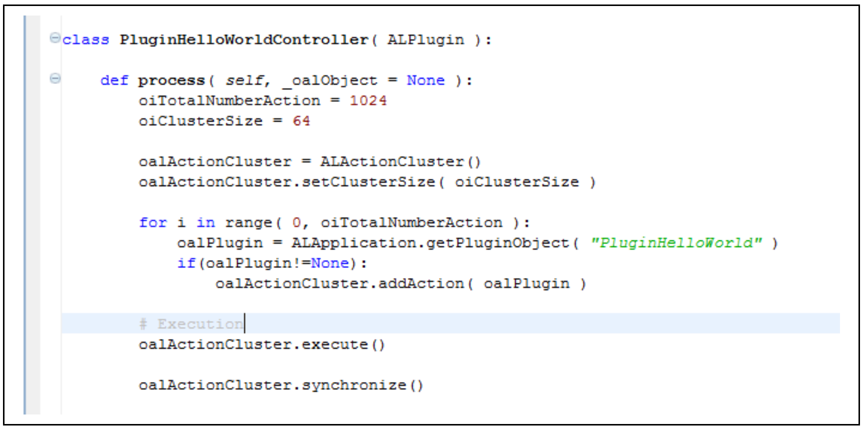
The execution starts creating the application (figure 3) to manage the resources (log, registers, etc) and verifies the command line (independent action) about the control flux parameters (if they are DEBUG, or profile, etc.). Also, the application executes the internal plug-in factory engine (independent action) to look for plug-ins in the standard folder (in this case, the folder "plug-ins" and its sub-folders). The plug-ins are compiled and available for all application modules.

The second step is to find and create the controller of the cluster of plug-ins. The plug-in factory mechanism instantiates an object based on the name of the class ("getPluginObject()", in figure 2), in that case it is the "PluginHelloWorldController" (note: it is not necessary to import or knowledge about the location on the disk of the plug-in code). The standard plug-in execution point is connected to the main application loop using the callback method "connectExecute()".



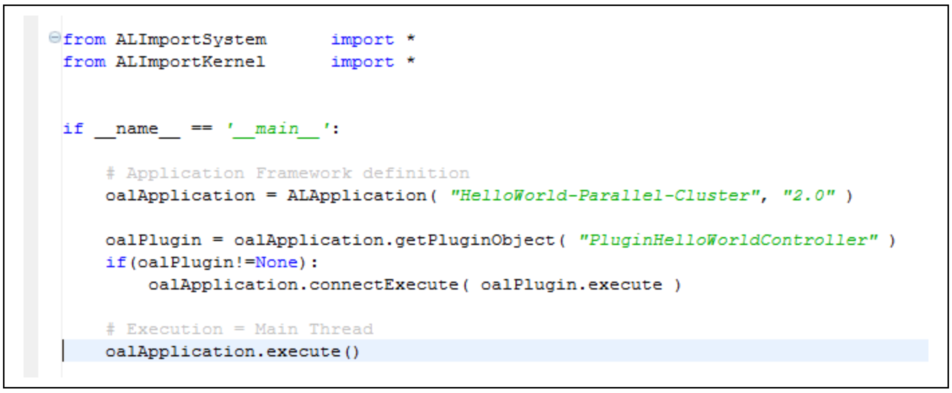
```
class PluginHelloWorld( ALPlugin ):

    def process( self, _oalObject = None ):
        oiPrintNumber = 10

        for i in range( 0, oiPrintNumber ):
            ALVerbose.screenID( self.getID(), "        [" + ALString(i) + "] Hello World " )
```

Fig. 1: The code of the plug-in to print the "hello world" 10 times.



```
class PluginHelloWorldController( ALPlugin ):

    def process( self, _oalObject = None ):
        oiTotalNumberAction = 1024
        oiClusterSize = 64

        oalActionCluster = ALActionCluster()
        oalActionCluster.setClusterSize( oiClusterSize )

        for i in range( 0, oiTotalNumberAction ):
            oalPlugin = ALApplication.getPluginObject( "PluginHelloWorld" )
            if(oalPlugin!=None):
                oalActionCluster.addAction( oalPlugin )

        # Execution
        oalActionCluster.execute()

        oalActionCluster.synchronize()
```

Fig. 2: The code of the plug-in "Controller" to generate a cluster of 1024 executions of the plug-in "hello world"



```
from ALImportSystem      import *
from ALImportKernel      import *


if __name__ == '__main__':

    # Application Framework definition
    oalApplication = ALApplication( "HelloWorld-Parallel-Cluster", "2.0" )

    oalPlugin = oalApplication.getPluginObject( "PluginHelloWorldController" )
    if(oalPlugin!=None):
        oalApplication.connectExecute( oalPlugin.execute )

    # Execution = Main Thread
    oalApplication.execute()
```

Fig. 3: The main code using the ALApplication module.

The third step showed in figure 3 is the execution of the main loop of the application "oalApplication.execute()". This method contains all control and communication methods (threads or sub-processes) started by the application. The application ends when all actions are finished or stopped.

The main cluster of 1024 plug-ins "hello world" is created and controlled by the plug-in "PluginHelloWorldController" presented in figure 2. The ALActionCluster class creates the plug-ins "hello world" ("for loop" in the figure 2) and controls the number of simultaneous actions executed by the system (64 simultaneous actions). It keeps the same number of simultaneous actions in execution: if one action ends it starts a new one. Finally, during the runtime all 64 simultaneous plug-ins "hello world" prints 10 times in the screen using the code presented in figure 1. The controller uses the method "oalActionCluster.synchronize()" (in figure 2) to blocks the execution until the end off all 1024 actions "hello world". The main application ends when all different active actions are finished.

Performance

The figure 4 presents the execution performance of the massive parallel "hello world" code. The plot indicates the total runtime by the number of required simultaneous threads controlled by the ALActionCluster class. It starts from the standard sequential processing thread by thread (12 hellos per second – total runtime 820 seconds) until 1024 simultaneous threads (783 hellos per second – total runtime 13 seconds). The graph shows clearly the limitation of the resources available in the hardware when an enormous amount of simultaneous threads was required (Intel dual core processor – Microsoft WindowsXP). In that case, the maximum optimization was obtained around 70 simultaneous threads: it is the maximum charge supported by the chip core in use.

The parallel execution reduces 65 times the total runtime of the application when compared to the pure sequential version. This example illustrates the simplicity of developing and controlling a massive parallel application based on dynamics plug-ins.

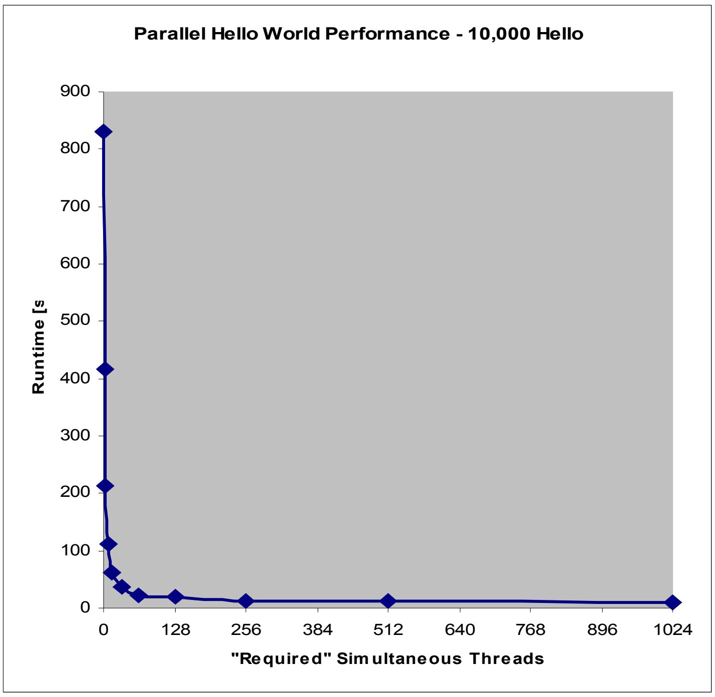


Fig. 4: Performance of the parallel code "hello world" – 65 times more faster when compared to the sequential version

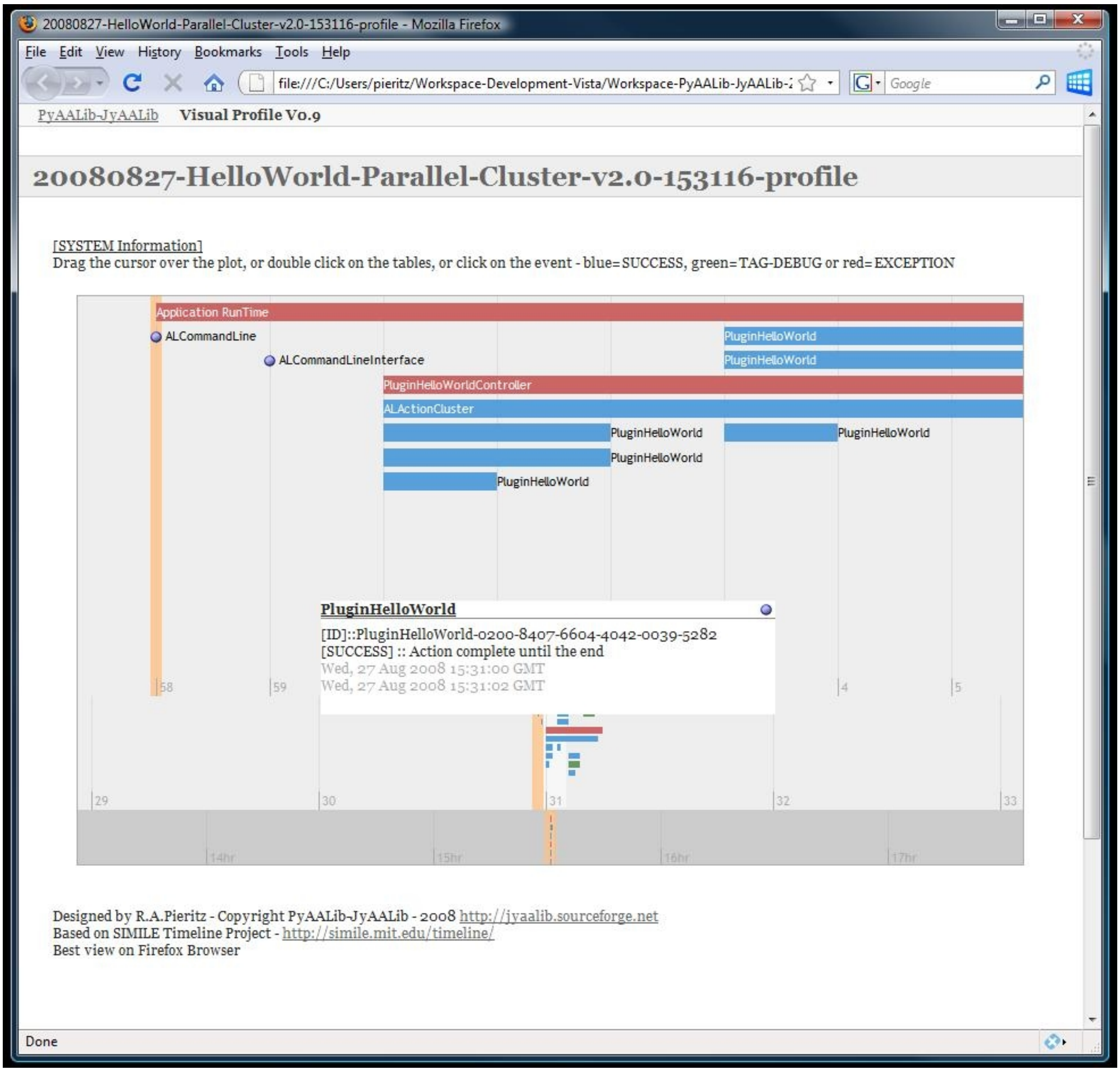
Profiler Tool and GUI

The Profiler GUI is a standard option available in the ALApplication module to plot a "timeline" of the concurrent simultaneous actions during the runtime of the application. It allows an iterative interface to map the asynchronous actions during the runtime of the massively parallel applications. The interface is dynamic, showing hyperlinks to present the independent output for each dynamic component used during the runtime of the application. The figure 5 shows the main interactive interface of the GUI plotting the execution of the "hello world" code (presented before). For simplicity it plots only a cluster of 3 simultaneous actions.

The user must add to the command line the option "--profile". It is available for all application based on the AALib ALApplication class. It can be used with other commands such as DEBUG and TAG-DEBUG options.

The HTML profile web page is created at the end of the execution of the application (or at any moment using the interface code). The profiler tool does not consume resources during the runtime. The profile GUI is composed by a "data base folder", a set of HTML pages using Javascript to plot the timeline. It is compatible with many "WebKit" based browsers (http://www.webkit.org).

The main feature available is the plot of the execution timeline for each independent element based on the main class ALAction (figure 6).


20080827-HelloWorld-Parallel-Cluster-v2.0-153116-profile - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Users/pieritz/Workspace-Development-Vista/Workspace-PyAALib-JyAALib-
Google
Application RunTime
ALCommandLine
ALCommandLineInterface
PluginHelloWorld
PluginHelloWorld
PluginHelloWorldController
ALActionCluster
PluginHelloWorld
PluginHelloWorld
PluginHelloWorld
PluginHelloWorld
58
59
4
5
29
30
31
32
33
14hr
15hr
16hr
17hr
Done

PyAALib-JyAALib Visual Profile V0.9

20080827-HelloWorld-Parallel-Cluster-v2.0-153116-profile

[SYSTEM Information]
Drag the cursor over the plot, or double click on the tables, or click on the event - blue=SUCCESS, green=TAG-DEBUG or red=EXCEPTION

PluginHelloWorld
[ID]::PluginHelloWorld-0200-8407-6604-4042-0039-5282
[SUCCESS] :: Action complete until the end
Wed, 27 Aug 2008 15:31:00 GMT
Wed, 27 Aug 2008 15:31:02 GMT

Designed by R.A.Pieritz - Copyright PyAALib-JyAALib - 2008 http://jyaalib.sourceforge.net
Based on SIMILE Timeline Project - http://simile.mit.edu/timeline/
Best view on Firefox Browser

Fig. 5: The PyAALib-JyAALib Profiler tool GUI

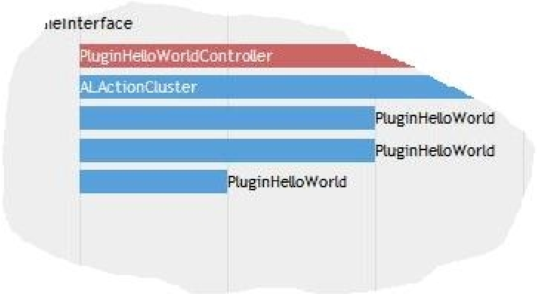
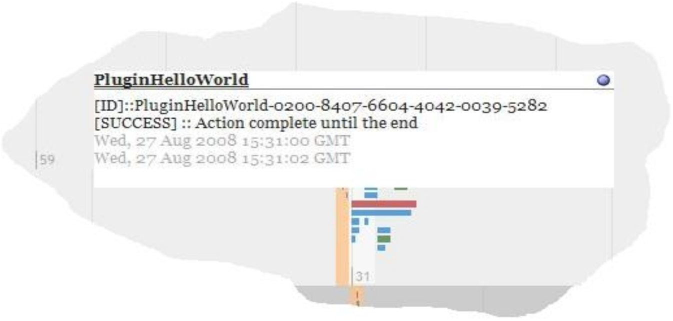


Fig. 6: The runtime timeline of dynamic objects and the associated color scheme.

The profiler tool plots the start time and end time of the dynamic object, showing three main color status:

- BLUE represents a SUCCESS: the object was executed until the end;
- GREEN represents the use of the TAG-DEBUG feature and a SUCCESS;
- RED identify the object when it contains at least one EXCEPTION;

The timeline plot is iterative and dynamic: the mouse drives the timeline in all directions. A single click on the plot moves the GUI to the time position. A single click on each object pops up a dialog box showing the main status and time information - figure 7. The URL link inside the dialog opens a log file showing all output from the object code.



PluginHelloWorld

[ID]::PluginHelloWorld-0200-8407-6604-4042-0039-5282
[SUCCESS] :: Action complete until the end
Wed, 27 Aug 2008 15:31:00 GMT
Wed, 27 Aug 2008 15:31:02 GMT

Fig. 7: The pop up dialog box showing the main information of each dynamic object and the Hyperlink to the associated log information.

Conclusions

The PyAALib-JyAALib is a smart and compact multi-thread plug-in based framework to design and implement massively parallel applications. It is designed to be used in a server or client software architecture and it is scalable for a single or multi-processor machines. It is open source and free software available for download (http://jyaalib.sourceforge.net). It runs in all the main operational systems (Windows, Linux and MacOS) and it is full compatible with Python and Jython-Java languages.

References

[1993] Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, Kelli A. Houston, “Object-Oriented Analysis and Design with Applications”, Ed. Addison-Wesley Object Technology Series.

[1995] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Ed. Addison-Wesley Professional Computing Series.

[2000] Hans Petter Langtangen, “Python Scripting for Computational science”, Ed. Springer.

[2001] Simon St. Laurent, Joe Johnston, Edd Dumbill, “Programming Web Services with XML-RPC”, Ed. O’Reilly.

[2002] Lucinda Dykes, Ed Tittel, Chelsea Valentine, “XML Schemas”, Ed. Transcend Technique.