

ProActive *Parallel Suite*



An Open Source Middleware For Parallel, Distributed, Multicore Computing

Monte-Carlo Pricing Contest

The OASIS Research Team and ActiveEon Company



Version 2010-04-29 2010-04-29



Copyright © 1997-2008 INRIA

ProActive v 2010-04-29 Documentation

An Open Source Middleware For Parallel, Distributed, Multicore Computing : Monte-Carlo Pricing Contest

The OASIS Research Team and ActiveEon Company

Legal Notice

The Monte-Carlo Pricing Contest is distributed under the GPL2 license.

Copyright INRIA 1997-2008.

Contributors and Contact Information

Team Leader:

Denis Caromel
INRIA 2004, Route des Lucioles
BP 93
06902
Sophia Antipolis Cedex
France
phone: +33 492 387 631
fax: +33 492 387 971
Denis.Caromel@inria.fr

ActiveEon Team

Christian Delbé
Arnaud Contes
Vladimir Bodnartchouk
Emil Salageanu

OASIS Team

Guillaume Laurent	Baptiste De Stefano
Robert Lovas	Nicolas Dodelin
Jonathan Martin	Yu Feng
Elton Mathias	Imen Filiali
Maxime Menant	Johann Fradj
Guilherme Perretti Pezzi	Abhijeet Gaikwad
Franca Perrina	Regis Gascon
Kamran Qadir	Jean-Michael Guillamume
Bastien Sauvan	Abhishek-Rajeev Gupta
Germain Sigety	Elaine Isnard
Etienne Vallette-De-Osia	Vasile Jureschi
Laurent Vanni	Francoise Baude
Yulai Yuan	Antonio Cansado
Sylvain Cussat-Blanc	Marcela Rivera
Boutheina Bennour	Ludovic Henric
Vincent Cave	Fabrice Huet
Guillaume Chazarain	Virginie Contes
Clement Mathieu	Mario Leyton
Eric Madelaine	Paul Naoumenko
Brian Amedro	Viet Dong Doan
Florin Bratu	Fabien Viale
Tomasz Dobek	Cédric Dalmaso
Khan Muhammad	Jean-Luc Scheefer
Julian Krzeminski	
Zhihui Dai	

Past And External Important Contributors

Lionel Mestre	Laurent Baduel
Matthieu Morel	Alexandre di Costanzo
Guillaume Chazarain	Romain Quilici
	Nadia Ranaldo
	Julien Vayssiere

Public questions, comments, or discussions can be posted on the ProActive public mailing list

proactive@ow2.org

The mailing list archive is placed at

<http://www.objectweb.org/wws/arc/proactive>

Bugs can be posted on the ProActive Jira bug-tracking system

<https://galpage-exp.inria.fr:8181/jira>

Table of Contents

Part I. Monte-Carlo Pricing Contest	2
Chapter 1. GridPlugtests 2008: Super Quant Monte Carlo Pricing Challenge	2
1.1. Overview	2
1.2. SuperQuant System Overview	2
1.3. Data Manipulation	2
1.4. Core Pricing System	4
1.5. Building the system	8
1.6. Sample test cases	8
1.7. Notes	9

Part I. Monte-Carlo Pricing Contest

Table of Contents

Chapter 1. GridPlugtests 2008: Super Quant Monte Carlo Pricing Challenge	2
1.1. Overview	2
1.2. SuperQuant System Overview	2
1.3. Data Manipulation	2
1.4. Core Pricing System	4
1.5. Building the system	8
1.6. Sample test cases	8
1.7. Notes	9

Chapter 1. GridPlugtests 2008: Super Quant Monte Carlo Pricing Challenge

1.1. Overview

The goal of the **Super Quant Monte Carlo challenge** is to design and develop a Grid-enabled Option Pricing Platform, for pricing high dimensional European options, that can cope up with time critical intense computational demand for complex pricing requests.

The participants will have to develop their system and deploy in the provided Grid environment using ProActive Grid Middleware. To test the robustness and responsiveness, the application will need to price a industry size batch of options within a limited time interval. The system will be judged by the experts in the finance as well as grid computing domains and the winners will take home the title of "**Super Quant 2008**".

The participation of this challenge, including the preliminary qualification tests, is free of charge, however, the teams that wish to participate in the Face to Face challenge in Sophia Antipolis (20 October - 23 October 2008) have to register and pass the preliminary (possibly remote) qualification tests.

To start with you can get the problem statement from the contest website. The participants can refer to the source code of a sample option pricing system provided on the website. In this document we describe the architecture of this pricing system and explain necessary details.

Note : No prior knowledge of finance is necessary to participate in the challenge as well as to understand the source code.

1.2. SuperQuant System Overview

The pricing system that we have provided is decomposed into three functional components:

- **Data Manipulating component:** which provides classes for manipulating the input/output data as well as the classes required for pricing the option. The input/output data for the challenge will be provided as xml descriptors. This xml files will be generated using the schema in the "compile/config" directory. We will describe the elements in this schema below, however for better understanding you are advised to refer to the schema itself.
- **Option Pricing Component:** which provides a sample source code for european option pricing algorithm and other algorithm related utilities.
- **Testing component:** in which we demonstrate how we can manipulate the input/output data for the challenge and compute the prices for some testcases. **Note :** Although participants are free to develop their own system, they must comply with the input/output data patterns described in this section.

1.3. Data Manipulation

For data manipulation the system uses the classes present in the plugtest.data package. This package is generated by castor (for castor documentation please refer to : www.castor.org) using the schema available in the "compile/config" directory of this archive. Castor is an open source data binding framework for transforming data from XML to Java.

The schema defines several elements which are used by the pricing component. Now we will briefly discuss the basic elements defined in the schema:

1. **Asset pool** is a collection of assets. An asset is any possession that has a value in an exchange. In the plugtests contest's context, we refer to a stock of any company as an asset.

```
<AssetPool>
  <Asset assetID="1" name="GOOG" date="2009-04-02T16:47:01" startPrice="100" volatility="0.2"
  dividend="0.0"/>
  <Asset assetID="2" name="YHOO" date="2009-04-02T16:47:01" startPrice="100" volatility="0.2"
  dividend="0.1"/>
</AssetPool>
```

```
<Asset assetID="3" name="MSFT" date="2009-04-02T16:47:01" startPrice="100" volatility="0.2"
dividend="0.1"/>
<Asset assetID="4" name="GOOG" date="2009-04-02T16:47:01" startPrice="100" volatility="0.2"
dividend="0.1"/>
<Asset assetID="5" name="YHOO" date="2009-04-02T16:47:01" startPrice="100" volatility="0.2"
dividend="0.1"/>
</AssetPool>
```

As can be seen an asset pool consist of a list of elements called asset. Each asset is characterised by certain attributes such as; it's start price, dividend, or volatility. For details on the definitions of attributes please refer to the schema.

- Correlation Matrix** represents the cross correlation values among the assets in the asset pool. In the option pricing algorithm the correlation values are used while simulating the future paths of any asset.

In the world of finance, correlation captures a notion of how two securities move in relation to each other. Correlation is computed into the correlation coefficient, which ranges between -1 and +1. Perfect positive correlation (a correlation coefficient of +1) implies that as one security moves, either up or down, the other security will move in lockstep, in the same direction. Alternatively, perfect negative correlation means that if one security moves in either direction the security that is perfectly negatively correlated will move by an equal amount in the opposite direction. If the correlation is 0, the movements of the securities is said to have no correlation, it is completely random.

If one security moves up or down there is as good a chance that the other will move either up or down, the way in which they move is totally random. In real life however you likely will not find perfectly correlated securities, rather you will find securities with some degree of correlation.

For example, the performance of two stocks within the same industry is strongly positively correlated although it may not be exactly +1.

For the plugtest, the correlation matrix will be provided as an input file. The example of such matrix can be shown as follows,

```
<CorrelationMatrix xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<Correlation AssetID="1" Values="1.0 0.0 0.0 0.0 0.0"/>
<Correlation AssetID="2" Values="0.0 1.0 0.0 0.0 0.0"/>
<Correlation AssetID="3" Values="0.0 0.0 1.0 0.0 0.0"/>
<Correlation AssetID="4" Values="0.0 0.0 0.0 1.0 0.0"/>
<Correlation AssetID="5" Values="0.0 0.0 0.0 0.0 1.0"/>
</CorrelationMatrix>
```

As can be seen in the figure, the correlation matrix contains list of assets and each asset is followed by a list correlation coefficients. The 0th value in the array is the correlation coefficient with relative to the 0th asset in the asset pool.

- Testcases**, is a collection of testcases that the participants will need to solve. An example of the test cases descriptor is given as follows,

```
<TestCases>
<BasketParams riskFreeRate="0.05" timeIntervalsPerYear="300" epsilonT="0.01" epsilonS="0.01"
epsilonR="0.01"/>

<TestCase testCaseID="1" Option="Put" PayOff="Avg" Algorithm="BarrierDownIn" timeToMaturity=
"1" barrier="5.0" Basket="0"/>

<TestCase testCaseID="2" Option="Call" PayOff="Min" Algorithm="Vanilla" timeToMaturity="3"
barrier="0.0" Basket="0 1 2 3 4"/>

</TestCases>
```

As you can see, the testcases element has two main sections.

- The first is the basket parameters which is a list of arguments that are required to compute the price, greeks of any basket option. These parameters are constant for all the testcases.
- While, the testcase is an actual test case that defines the problem.

Each test case has an identifier. **The results for individual test cases MUST include the this identifier.** An automated program will use this identifier to judge the final results.

- Results** for the computation must be stored in XML format and should follow the results schema as defined in the schema.

```

    <Results>
  <Result>
    <testCaseID>1</testCaseID>
    <Price CallPrice="0.0" PutPrice="5.570964213092109"
      Variance="75.06186453117554" CI="0.05342496980624275"/>
    <Greeks Delta="0.0" Gamma="0.0" Theta="0.0" Rho="0.0" Speed="0.0"/>
  </Result>
  <Result>
    <testCaseID>2</testCaseID>
    <Price CallPrice="0.049415277448755006" PutPrice="0.0"
      Variance="0.5156580601932929" CI="0.004428080593084317"/>
    <Greeks Delta="0.0 0.0 0.0 0.0 0.0" Gamma="0.0 0.0 0.0 0.0 0.0"
      Theta="0.0" Rho="0.0" Speed="0.0 0.0 0.0 0.0 0.0"/>
  </Result>
  <Metrics startTime="1217603030800" stopTime="1217603141008" wallTime="110"/>
</Results>

```

As you can see in the example results, each computed results comprises of the three components.

- Price of the Option:** This include the premium/the price to be paid in order to buy the option, the variance and confidence interval of the computed price. The example for the price can be seen in the figure above.
- Greeks:** In mathematical finance, the Greeks are the quantities representing the market sensitivities of derivatives such as options. Each "Greek" measures a different aspect of the risk in an option position, and corresponds to a parameter on which the value of an instrument or portfolio of financial instruments is dependent. The name is used because the parameters are often denoted by Greek letters. (Wikipedia)

You can compute as many greeks as you can along with the price computed for each options, though we limited to compute following five type of greeks:

- Delta**, which represents the rate of change between the option's price and the underlying asset's price - in other words, price sensitivity. You should compute the delta for ALL the assets in the basket.
 - Gamma**, which represents the rate of change between an option portfolio's delta and the underlying asset's price - in other words, second-order time price sensitivity. You should compute the gamma values for ALL the assets in the basket.
 - Theta**, which represents the rate of change between an option portfolio and time, or time sensitivity.
 - Rho**, which represents the rate of change between an option portfolio's value and the interest rate, or sensitivity to the interest rate.
 - Speed**, which measures third order sensitivity to price. The speed is the third derivative of the value function with respect to the underlying price> You must compute the Speed for ALL the assets in the basket.
- Metric** It should record the computational time required for computing the entity that it is encompassed in.

1.4. Core Pricing System

The core pricing algorithm is an example of a monte carlo based option pricing algorithm. The sample source code in this package is an example of how one can use **ProActive Grid Middleware** for developing such an option pricing system.

- The pricing engine is an engine which churns out bags of tasks of core algorithm using the Monte carlo API available in the ProActive Grid Middleware. The monte carlo based european option pricing engine to compute the price and the greeks for the european basket options. The engine can be constructed as follows,

```
/**
 * Constructor for the monte carlo based european option pricing engine
 *
 * @param descrFile the deployment descriptor so that engine can launch the algorithm tasks.
 * @param vn_name the virtual node name of the worker
 * @param master_vn_name the name of virtual master node
 */
public MCEuropeanPricingEngine(final File descrFile, String vn_name, String master_vn_name)
```

and the monte carlo based algorithm is implemented as a bag-of-task application and constructed as follows,

```
/**
 * The constructor to create the instance of monte carlo based algorithm's task.
 * Such tasks are launched by the engine.
 *
 * @param basketParams the market constants, as provided in the testcases
 * @param testCase individual test case
 * @param testCaseAssetPool the basket for the testcase
 * @param correlationMatrix the correlation matrix for the assets in the basket
 * @param mciter the number of monte carlo simulation per task, can be controlled for accuracy and speed
 */
public MCEuropeanPricingAlgorithm(BasketParams basketParams, TestCase testCase,
    AssetPool testCaseAssetPool, Matrix correlationMatrix, long mciter)
```

Note that, we here provide only the source code for computing the price/premium of the basket option. However, the participants will need to write the code for computing greeks intelligently. We have provided the mathematical models for computing greeks in the problem statement which is available on the GridPlugtests website.

- The monte carlo based basket option is a collection of the parameters required for the algorithms. This package provides an abstract implementation of a basket option. Each basket has an abstract pricing engine which computes its price and the other greeks. The european basket option class extends this abstract class and assigns the monte carlo based pricing engine to itself. It can be constructed as follows.

```
/**
 * The constructor takes the market constants in the basketParams, the correlation matrix for the assets in the basket in the corMatrix, the test case parameters and the asset pool in the basket
 */
public MCEuropeanBasketOption(BasketParams basketParams, Matrix corMatrix, TestCase testCase,
    AssetPool subAssetPool)
```

The pricing engine can be set to the basket using this method,

```
/**
 * Sets the pricing engine to the option
 *
 * @param mcEuroEngine monte carlo based pricing engine to price european options
 */
public void setPricingEngine(final MCEuropeanPricingEngine mcEuroEngine)
```

- In the third subpackage, there are several utilities provided that can be re-used in your system
- to load the testcases from the input file.

```
/**
 * Reads the testcases from the input file.
```

```

*
* @param testCaseFile the file containing the testcases.
* @return testCases returns the testcases
*/

```

```
public static TestCases loadTestCases(File testCaseFile)
```

- to load the assetpool from the input file

```

/**
 * Reads the asset pool from the input file. Asset pool is a collection of
 * assets that the testcases are based on.
 *
 * @param assetPoolFile the input file containing the asset pool
 * @return the asset pool
 */

```

```
public static AssetPool loadAssetPool(File assetPoolFile)
```

- to load the correlation matrix from the input file

```

/**
 * Reads the correlation matrix from the input file. The correlation matrix
 * is a matrix containing correlation values among the assets in the asset
 * pool.
 *
 * @param corrFile the file containing the correlation matrix
 * @return returns the correlation matrix
 */

```

```
public static CorrelationMatrix loadCorrMatrix(File corrFile)
```

- to parse the correlation matrix into the Jama.Matrix format

```

/**
 * Parses the correlation matrix from the CorrelationMatrix format into
 * Matrix format. This preprocessing is convenient for reducing the
 * computational time required for computing the subset of the correlation
 * matrix.
 *
 * @param corrMatrix Correlation Matrix that is defined in the data format
 * @return
 */

```

```
public static Matrix parseCorrelationDescriptor(CorrelationMatrix corrMatrix)
```

- to extract a subset of assets from the global asset pool

```

/**
 * @param assetPool asset pool contains a set of assets
 * @param basket the basket indices of assets which will be pulled from the
 * assetpool to form the subset of the assetpool for the current
 * test case
 * @return the extracted sub set of asset pool
 */

```

```
public static AssetPool extractAssetPool(AssetPool assetPool, int[] basket)
```

- to extract the subset of correlation matrix from the global correlation matrix

```

/**
 * @param corrMatrix a super matrix of correlation values among all the assets
 * @param basket the basket indices of assets of which the correlation values

```

```

* will be extracted from the (super) correlation matrix
* @return a sub-correlation matrix for the current test case
*/

public static Matrix extractCorrelationMatrix(Matrix corrMatrix, int[] basket)

```

- Testing component In the plugtest.test package, we provide a simple example of:
 - how to extract the data from the input files
 - how to process the data individual test cases 3. how to compute the price
 - how to store the output data into the output file.

The source code of the main test case challenge is as follows,

```

// load the data from testcases descriptor file
final TestCases testCases = LoadData.loadTestCases(testCasesDescriptorFile);

// load the asset pool from the assetpool descriptor file
final AssetPool assetPool = LoadData.loadAssetPool(assetPoolDescriptorFile);

// load the correlation matrix from the correlation matrix descriptor file
final CorrelationMatrix corrMatrix =
LoadData.loadCorrMatrix(correlationMatrixDescriptorFile);

// get the fixed parameters from the test cases
final BasketParams basketParams = testCases.getBasketParams();

// get the correlation descriptor in a jama.Matrix format
final Matrix matCorrelation = ProcessData.parseCorrelationDescriptor(corrMatrix);

/**
 * Create a monte carlo based european option pricing engine
 * it will price an euroean/barrier option using monte carlo based methods.
 * The workers and master's virtual names can be given as arguments otherwise are assigned
 * default values.
 * Since engine deploys the application in the grid setting, we have to provide
 * the GCM deployment descriptor file to it.
 */
MCEuropeanPricingEngine mcEngine = new MCEuropeanPricingEngine(deploymentDescriptorFile,
vn_name,
    master_vn_name);

long startTime = System.currentTimeMillis();

MCEuropeanBasketOption mcEuroBsk = new MCEuropeanBasketOption(basketParams);
// Now, lets set the monte carlo based pricing engine to the basket
mcEuroBsk.setPricingEngine(mcEngine);

// initiate the Monte Carlo API and hence the ProActive runtime
mcEuroBsk.initMC();

// process each test case
for (int i = 0; i < testCases.getTestCaseCount(); i++) {

    /**
     * We first need to construct the basket option out of the global asset pool.
     * As the testcase includes the indices of the assets from this global pool,
     * we must extract the assets from it before processing it.
     */
    testCaseAssetPool = ProcessData.extractAssetPool(assetPool,

```

```

testCases.getTestCase(i).getBasket());

    /**
     * We also need to extract submatrix from the global correlation matrix.
     * The same basket of indices is used to get this submatrix.
     */
    testCaseCorrMatrix = ProcessData.extractCorrelationMatrix(matCorrelation, testCases
        .getTestCase(i).getBasket());
    /**
     * With all the paramters, market constants such as interest rate, time per interal and
     monte carlo
     * simulations in basketParams, the correlation matrix for the sub set of assets,
     * other parameters associated with testcase such as the execution type, option type,
     pay off types and time
     * to maturity and, of course, the assets in the basket,
     * we construct the baskey option using the MCEuropeanBasketOption constructor.
     */
    mcEuroBsk.set(testCaseCorrMatrix, testCases.getTestCase(i), testCaseAssetPool);

    // Fire the engine to price the option using monte carlo based algorithm
    mcEuroBsk.calculateMC();

    // Get the result for this testcase and update the final results
    results.setResult(mcEuroBsk.getResult());
-}

long stopTime = System.currentTimeMillis();
metrics.setStartTime(startTime);
metrics.setStopTime(stopTime);
metrics.setWallTime((stopTime -- startTime) -/ 1000);

// update the metrics for final results.
results.setMetrics(metrics); //

// store the final results in the out put descriptor file
DisplayData.printResults(results, new FileOutputStream(FINAL_RESULTS_DESCRIPTOR));

```

1.5. Building the system

We provide a set of scripts to build this package. The ant script is available in the "compile" directory. The script will build the source code as well as the java documentations for the code. You can build it using following simple command:

Linux:

```
$cd ARCHIVE_DIR/compile
```

```
./build
```

Windows:

```
#cd ARCHIVE_DIR/compile
```

```
#build.bat
```

This will create the necessary jar for running the test cases as described in the next section.

1.6. Sample test cases

The sample input files can be found in "bin/testcases" directory:

1. **assetPool0.xml** : contains the collection of assets
2. **correlation0.xml** : contains the correlation matrix for the assets
3. **testcases0.xml** : contains the sample test cases

You can execute the following script to run this example.

Linux:

```
$cd ARCHIVE_DIR/bin  
$./gridplugtests2008.sh
```

Windows:

```
# cd ARCHIVE_DIR\bin  
# gridplugtests2008.bat
```

The sample output file, **GridPlugtestsV2008Results-TEAMNAME.xml**, will be generated in the testcase directory

1.7. Notes

1. Note the names of the input files will be provided at the day of the contest.
2. The name of the results file must be **GridPlugtestsV2008Results-TEAMNAME.xml**. If there is any change in the format, we will announce at the beginning of the contest. The participants will need to change the **TEAMNAME** to their team's name (unless their team's name is TEAMNAME ;))
3. If there are any changes in the schema or input/output format, those will be announced on the mailing list.