CMPLServer
An open source approach for distributed and grid optimisation

Mike Steglich
Technical University of Applied Sciences Wildau
Hochschulring 1, D-15745 Wildau, Germany
E-mail: mike.steglich@th-wildau.de

KEYWORDS
Mathematical Modelling Language, Distributed and Grid Optimisation

ABSTRACT

This paper describes CMPLServer, an XML-RPC-based web service for distributed and grid optimisation for CMPL (Coin|Coliop Mathematical Programming Language) which is a mathematical programming language as well as a system for mathematical programming and optimisation of linear optimisation problems.

1 INTRODUCTION

Since the change in information and communication technologies over the last couple of years, Operations Research has been faced with new user needs. For example, the increasing use of mobile devices to make and analyse decisions requires Operations Research software that provides remote access to databases, modelling and optimisation software. Due to these demands, a lot of distributed optimisation approaches like the NEOS Server (Czyzyk et al. 1998), that be can used via a lot of different interface including the Kestrel interface (Dolan et al. 2008), commercial solutions like Gurobi Cloud, the CPLEX Enterprise Server and FICO XPRESS-Insight and also open source approaches like COIN-OR Optimization Services (Fourer et al. 2010) have appeared.

CMPL (<Coin|Coliop>Mathematical Programming Language) is also faced with these new challenges. Therefore, the CMPLServer was created as an open source approach for distributed and grid optimisation. The main targets are to ensure a high performance and a high reliability and to enable CMPL users to start and use CMPLServer as easily as possible.

CMPL is a mathematical programming language and a system for mathematical programming and optimisation of linear optimisation problems. CMPL executes CBC, GLPK, Gurobi, SCIP and CPLEX directly to solve the generated model instance. Since it is also possible to transform the mathematical problem into MPS, Free-MPS or OSiL files, alternative solvers can also be used. CMPL contains pyCMPL and jCMPL as application programming interfaces (API's) for Python and Java and is available for most of the relevant operating systems (Windows, OS X and Linux). CMPL is a COIN-OR project initiated by the Technical University of Applied Sciences Wildau and the Institute for Operations Research and Business Management at the Martin Luther University Halle-Wittenberg. (Steglich and Schleiff 2010) The CMPL distribution is available at http://coliop.org.

In this article, the CMPLServer which is an XML-RPC-based web service for distributed and grid optimisation is discussed. After an overview of the main functionalities in section 2, the XML-based file formats (`CmplInstance`, `CmplSolutions`, `CmplMessages`, `CmplInfo`) for the communication between a CMPLServer and its clients are described in section 3. In section 4 the single server mode including the synchronous and the asynchronous mode is explained. All these distributed optimisation procedures require a one-to-one connection between a CMPLServer and the client. Therefore, section 5 discusses how CMPLServers from several locations can be coupled to one "virtual CMPLServer", how a client can connect with it and how optimisation jobs are coordinated within the CMPLServer grid. In the following section 6, is presented how a high reliability can be ensured by different approaches. The last section, number 7, describes an analysis of the positive effects of shipping optimisation problems to a CMPLServer or into a grid of CMPLServers versus the corresponding network traffic.

2 CMPLSERVER IN A GLANCE

The CMPLServer is an XML-RPC-based web service for distributed and grid optimisation. XML-RPC provides XML based procedures for Remote Procedure Calls (RPC) which are transmitted between a client and a server via HTTP. (Laurent et al. 2001, p. 1) XML-RPC was chosen because it is less resource consuming than other protocols like SOAP or REST due to its simpler functionalities.

As shown in Figure 1 a CMPLServer can be used in a single server mode or in a grid mode. Both modes can be understood as distributed systems "in which hardware and software components located at networks computers communicate and coordinate their actions only by passing messages". (Coulouris et al. 2012, p. 2) Distributed optimisation is in this meaning interpretable as a distributed system that can be used for solving optimisation problems. (Kshemkalyani and Singhal 2008, p. 1; Fourer et al. 2010)

In the single server mode only one CMPLServer exists in the network and can be accessed synchronously or asynchronously by the clients. The client sends the model to the CMPLServer and then waits for the results. If the model is feasible and an optimal solution is found the solution(s) can be received. If the model contains syntax or other errors or if the model is not feasible CMPL and solver messages can be obtained.

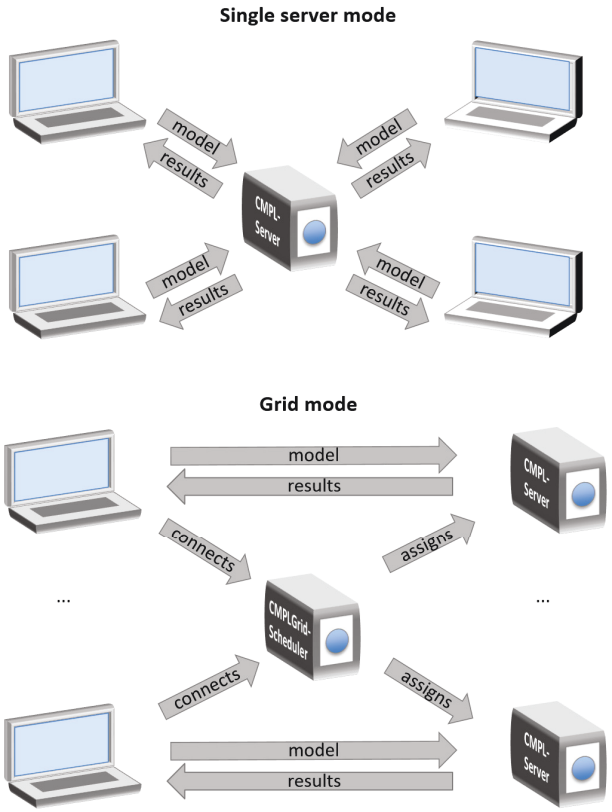

Figure 1: Single server mode and grid mode.

Whereby in the synchronous mode the client has to wait for the results and messages in one process after sending the problem, a model can also be solved asynchronously in several steps. After sending the model to the CMPL-Server via the method `send` the server status can be obtained with the method `knock`. When the CMPL-Server is finished, the solution, the CMPL as well as the solver states and messages can be received by the method `retrieve`. It is reasonable to use the single server mode if a large model is formulated on a thin client in order to solve it remotely on a CMPLServer that is installed on a high performance system.

The grid mode extends this approach by coupling CMPLServers from several locations and at least one coordinating CMPLGridScheduler to one "virtual CMPLServer" as a grid computing system that can be defined "as a system that coordinates distributed resources using standard, open, general-purpose protocols and interfaces to deliver non-trivial qualities of service." (Foster and Kesselman 2004, pos. 722) For the client there does not appear any difference whether there is a connection made to a single CMPLServer or to a CMPLGrid. The client's model is connected with the same functionalities as for a single CMPLServer to a CMPLGridScheduler which is responsible for the load balancing within the CMPLGrid and the assignment of the model to one of the connected CMPLServers. After this step the client is automatically connected to the chosen CMPLServer for one optimisation run and the model can be solved synchronously or asynchronously. A CMPLGrid should be used for handling a huge amount of large scale optimisation problems. An example can be a simulation in which each agent has to solve its own optimisation problem at several times. An additional example for such a CMPLGrid application is an optimisation web portal that provides a huge amount of optimisation problems.

3 CMPL SPECIFIC XML FORMATS

The communication between a client and a server works through XML-RPC and four CMPL-specific XML formats for the communication between clients and servers. A `CmplInstance` file contains an optimisation problem formulated in CMPL, the corresponding sets and parameters in the `CmplData` file format as well as all CMPL and solver options that belong to the CMPL model. If the model is feasible and a solution is found, then a `CmplSolutions` file contains the solution(s) and the status of the invoked solver. If the model is not feasible then only the solver's status and the solver messages are given in the `CmplSolutions` file. The `CmplMessages` file is intended to provide the CMPL status and (if existing) the CMPL error or warning messages. A `CmplInfo` file is an XML file that contains (if requested) several statistics and the generated matrix of the CMPL model. For all of these files the XSD schemes are available at www.coliop.org/schemes.

This section is intended to describe these XML formats by using the following simple linear programme:

$$\text{maximise} \quad 10 \cdot x_1 + 18 \cdot x_2 + 22 \cdot x_3 \tag{1}$$

$$\text{subject to} \quad 5 \cdot x_1 + 10 \cdot x_2 + 15 \cdot x_3 \leq 175 \tag{2}$$

$$10 \cdot x_1 + 5 \cdot x_2 + 10 \cdot x_3 \leq 200 \tag{3}$$

$$x_n \geq 0; n = 1(1)3 \tag{4}$$

This model can be formulated in matrix-vector form as follows:

$$\text{maximise} \quad \mathbf{c}^{\mathbf{T}} \cdot \mathbf{x} \tag{5}$$

$$\text{subject to} \quad \mathbf{a} \cdot \mathbf{x} \leq \mathbf{b} \tag{6}$$

$$\mathbf{x} \geq \mathbf{0} \tag{7}$$

with

$$\mathbf{c} = \begin{pmatrix} 5 \\ 10 \\ 15 \end{pmatrix}, \mathbf{a} = \begin{pmatrix} 55 & 10 & 15 \\ 10 & 5 & 10 \end{pmatrix} \tag{8}$$

$$\mathbf{b} = \begin{pmatrix} 175 \\ 200 \end{pmatrix}, \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{9}$$

The first step to solve the model is to formulate a `CmplData` file and a CMPL model. A `CmplData` file is a plain text file that contains the definition of parameters and sets with their values in a specific syntax. As shown in the Listing 1 it is necessary to define two indexing sets `n` and `m` (lines 1 and 2) used for the definitions of the vectors `c` and `b` and the matrix `a` (lines 4-6).

```
01 %n set <1..2>
02 %m set <1..3>
03
04 %c[m] < 15 18 22 >
05 %b[n] < 175 200 >
06 %a[n,m] <  5 10 15   10 5 10 >
```

Listing 1: CmplData example - test.cdat

The parameters and sets can be read into a CMPL model by using the CMPL header argument `%data` as shown in the CMPL model (Listing 2 - line 1). The set `m` can then be used for the definition of the vector `x` of the nonnegative, continuous variables (line 4). The next lines are intended to create the objective function `profit` and the constraints `res` in matrix-vector form as in the terms (5)-(6) .

```
01 %data test.cdat
02
03 variables:
04   x[m]: real[0..];
05 objectives:
06   profit: c[]T * x[] -> max;
07 constraints:
08   res: a[,] * x[] <= b[];
```

Listing 2: CMPL example - test.cmpl

To solve this model on a CMPLServer located at `http://10.0.1.52:8008` and to to save the generated matrix and also some statistics the following command has to be executed:

```
cmpl test.cmpl ↵
  -cmplUrl http://10.0.1.52:8008  ↵
  -matrix "test.mat" -s "test.stat"
```

Figure 2 gives an overview of the main working steps of solving a CMPL model on a CMPLServer. In the first step CMPL writes automatically all model relevant information (CMPL and `CmplData` files, CMPL and solver options) in a `CmplInstance` file and sends it to the connected CMPLServer, where the included CMPL model and the corresponding `CmplData` files are parsed and translated into a Free-MPS file.

As shown in Listing 3 for the given example a `CmplInstance` file consists of three major sections. The `<general>` section contains the name of the problem and the `jobId` that is received automatically while connecting the CMPLServer (lines 3-6). The `<options>` section consists of the CMPL and the solver options that a user has specified on the command line (lines 7-11). The `<problemFiles>` section is indented to store the CMPL file(s) and all corresponding `CmplData` files. The CMPL example file (Listing 2) is included in the `CmplInstance` file in the lines 13-21 followed by the corresponding `CmplData` file (Listing 1) in the lines 22-19.

To avoid misinterpretations of some special characters while reading and parsing the `CmplInstance` on the CMPLServer the content of the CMPL model and the `CmplData` files are automatically unescaped.

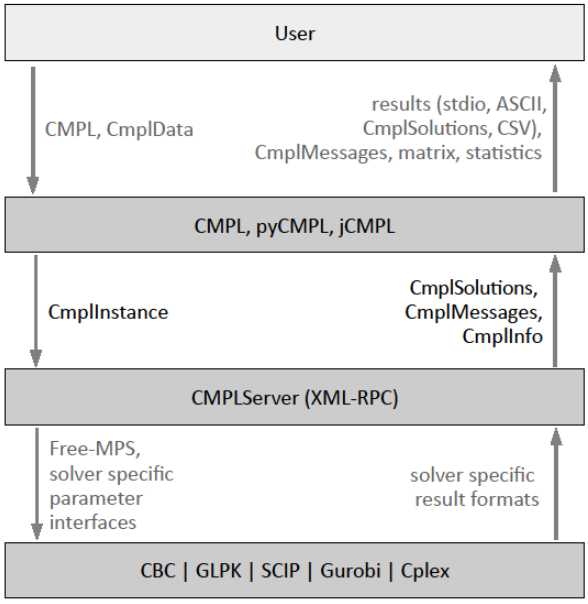

Figure 2: Single server mode and grid mode

The generated Free-MPS file and the solver specific parameters are handed over to the chosen solver that is executed directly. If the problem is feasible and an optimal solution is found this solution is read in the form of the solver specific result format. The CMPLServer then automatically creates three XML-based files (`CmplSolutions`, `CmplMessages`, `CmplInfo`) and sends them to the CMPL client. After that, the user can obtain (if an optimal solution is found) the standard solution report, can save the solution(s) in several formats and is also able to get the generated matrix and some statistics. If the CMPL model contains errors, then the user retrieves the CMPL messages automatically.

`CmplSolutions` is an XML-based format for representing the general status and the solution(s) if the problem is feasible and one or more solutions are found. As shown in Listing 4 a `CmplSolutions` file contains a `<general>` block for general information about the solved problem and a `<solutions>` block for the results of all solutions found including the variables and constraints. Each entry in the variables and constraints section contains information about the index, the name, the type, the activity, the bounds and the marginal (shadow prices or reduced costs).

`CmplMessages` is an XML-based format for representing the general status and (if existing) the errors or warnings of the transformation of a CMPL model in one of the supported output files. A `CmplMessages` file consists of two major sections. The `<general>` section describes the general status, the name of the model and a general message after the transformation. The `<messages>` section contains one or more messages about specific lines in the CMPL model.

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes"?>
<CmplInstance version="1.0">
  <general>
    <name>test.cmpl</name>
    <jobId>10.0.1.2-2014-01-05-17-05-23-496795</jobId>
  </general>
  <options>
    <opt>%arg -cmplUrl http://10.0.1.52:8008</opt>
    <opt>%arg -matrix "test.mat"</opt>
    <opt>%arg -s "test.stat"</opt>
  </options>
  <problemFiles>
    <file name="test.cmpl" type="cmplMain">
      %data test.cdat
      variables:
          x[m]: real[0..];
      objectives:
          profit: c[]T * x[] -&gt; max;
      constraints:
          res: A[,] * x[] &lt;= b[];
    </file>
    <file name="test.cdat" type="cmplData">
      %n set &lt;1..2&gt;
      %m set &lt;1..3&gt;

      %c[m] &lt; 15 18 22 &gt;
      %b[n] &lt; 175 200 &gt;
      %A[n,m] &lt; 5 10 15  10 5 10 &gt;
    </file>
  </problemFiles>
</CmplInstance>
```

Listing 3: CmplInstance example - test.cinst

```
<?xml version = "1.1" encoding="UTF-8" standalone="yes"?>
<CmplSolutions version="1.0">
  <general>
    <instanceName>test.cmpl</instanceName>
    <nrOfVariables>3</nrOfVariables>
    <nrOfConstraints>2</nrOfConstraints>
    <objectiveName>profit</objectiveName>
    <objectiveSense>max</objectiveSense>
    <nrOfSolutions>1</nrOfSolutions>
    <solverName>CBC</solverName>
    <variablesDisplayOptions>(all)</variablesDisplayOptions>
    <constraintsDisplayOptions>(all)</constraintsDisplayOptions>
  </general>
  <solution idx="0" status="optimal" value="405">
    <variables>
      <variable idx="0" name="x[1]" type="C" activity="15" lowerBound="0"
                upperBound="INF" marginal="0"/>
      <variable idx="1" name="x[2]" type="C" activity="10" lowerBound="0"
                upperBound="INF" marginal="0"/>
      <variable idx="2" name="x[3]" type="C" activity="0" lowerBound="0"
                upperBound="INF" marginal="-7"/>
    </variables>
    <linearConstraints>
      <constraint idx="0" name="res[1]" type="L" activity="175"
                lowerBound="-INF" upperBound="175" marginal="1.4"/>
      <constraint idx="1" name="res[2]" type="L" activity="200"
                lowerBound="-INF" upperBound="200" marginal="0.8"/>
    </linearConstraints>
  </solution>
</CmplSolutions>
```

Listing 4: CmplSolution example - test.csol

After executing the CMPL example model, CMPL will finish without errors. The general status is represented in the following `CmplMessages` file `test.cmsg` shown in Listing 5. If a wrong symbol name for the matrix `A[,]` (e.g. `a[,]`) is used in line 11, CMPL would be finished with errors represented in `CmplMessages` file `test.cmsg` shown in Listing 6.

In case that a user requests some statistics or wants to obtain the generated matrix a `CmplInfo` file is generated automatically and sent to the CMPL client. `CmplInfo` is a simple XML file that contains several statistics and the generated matrix of the CMPL model as shown in Listing 7.

4 SINGLE SERVER MODE

In the single server mode only one CMPLServer exists in the network and can be connected by several CMPL clients.

The first step to establish the single server mode is to start the CMPLServer by typing the following command.

```
cmplServer -start  [<port>] ↵
     [-showLog]
```

Optionally, a port can be specified as a second argument and the log file can be shown by using the command line argument `showLog`. The behaviour of a CMPLServer can be influenced by editing the file `cmplServer.opt` that is located in the CMPLServer installation folder. The example below shows the default values in this file.

```
cmplServerPort = 8008
maxProblems = 4
maxInactivityTime = 43200
serviceIntervall = 30
solvers = cbc glpk
```

The default port of the CMPLServer can be specified with the parameter `port`. The parameter `maxProblems` defines how many problems can be carried out simultaneously. If more problems than `maxProblems` are connected with the CMPLServer, the supernumerary problems are assigned to the problem waiting queue and automatically started if a running problem is finished or cancelled. If a problem is inactive longer than defined by the parameter `maxInactivityTime` it is cancelled and deleted automatically by the CMPLServer. This procedure as well as the problem waiting queue handling are performed by a service thread that works perpetual after a couple of seconds defined by the parameter `serviceIntervall`. With the parameter `solvers` it can be specified which solvers are provided by the CMPLServer.

A running CMPLServer can be accessed by the CMPL binary or via CMPL's Python and Java APIs that contain CMPLServer clients. One can execute a CMPL model remotely on a CMPLServer by using the command line argument `-cmplUrl`.

```
cmpl <problem>.cmpl -cmplUrl  ↵
     http://<ip-or-domain>:<port>
```

In this case CMPL uses the CMPLServer synchronously. That means CMPL waits for the results and messages in one process right after sending the problem.

In pyCMPL and jCMPL programmes a CMPLServer can be connected via the method `Cmpl.connect()` and executed synchronously with the method `Cmpl.solve()` or asynchronously by using the methods `Cmpl.send()`, `Cmpl.knock()` and `Cmpl.retrieve()`. These main functionalities are illustrated in Figure 3.

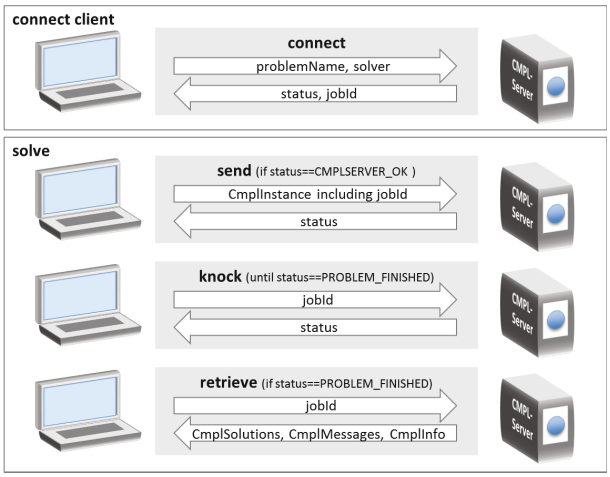

Figure 3: Single server mode procedures

In the first step, the client connects the CMPLServer, hands over the problem name and the solver with which the problem shall be solved. Then the client receives the status of the CMPLServer and also the `jobId` if the status is `CMPLSERVER_OK`. The status equals `CMPLSERVER_ERROR` if the demanded solver is not supported or an error on the CMPLServer occurs.

The synchronous method `Cmpl.solve()` is a bundle of the asynchronous methods `Cmpl.send()`, `Cmpl.knock()` and `Cmpl.retrieve()`.

`Cmpl.send()` sends a `CmplInstance` XML string that contains all relevant information about the CMPL model including the `jobId`. If the number of running problems including the model sent is greater than `maxProblems` the model is moved to the problem waiting queue and the CMPLServer returns the status `CMPLSERVER_BUSY`. If the status is `CMPLSERVER_OK`, then the CMPLServer starts the solving process automatically. After that the status is set to `PROBLEM_RUNNING`.

```
<?xml version="1.0" encoding="UTF-8"?>
<CmplMessages version="1.1">
  <general>
    <generalStatus>normal</generalStatus>
    <instanceName>test.cmpl</instanceName>
    <message>cmpl finished normal</message>
  </general>
</CmplMessages>
```

Listing 5: CmplMessages example without errors and warnings: test.cmsg

```
<?xml version="1.0" encoding="UTF-8"?>
<CmplMessages version="1.1">
  <general>
    <generalStatus>error</generalStatus>
    <instanceName>test.cmpl</instanceName>
    <message>cmpl finished with errors</message>
  </general>
  <messages numberOfMessages="1">
    <message type="error" file="test.cmpl" line="11" description="syntax error,
        unexpected SYMBOL_UNDEF"/>
  </messages>
</CmplMessages>
```

Listing 6: CmplMessages example with an error

```
<?xml version="1.0" encoding="UTF-8"?>
<CmplInfo version="1.0">
  <general>
    <instancename>test.cmpl</instancename>
  <general>
  <statistics file="test.stat">


File: /Users/mike/CmplServer/10.0.1.2-2014-01-05-17-05-23-496795/test.cmpl
3 Columns (variables), 3 Rows (constraints + objective function)
6 (100%) of 6 coefficients of the constraints are non-zero.


  </statistics>
  <matrix file="test.mat">
Variable name                      x[1]           x[2]           x[3]
Variable type                         C              C              C

profit              max              15             18             22
   Subject to                                                                RHS
res[1]              L                 5             10             15         175
res[2]              L                10              5             10         200

Lower Bound                           0              0              0
Upper Bound

  </matrix>
</CmplInfo>
```

Listing 7: CmplInfo example - test.cinfo

In the next step, the client asks the CMPLServer whether solving the problem is finished or not via `Cmpl.knock()` whereby the `jobId` identifies the problem and the CMPLServer returns the current status. The client has to knock until the status is `PROBLEM_FINISHED` (or `CMPLSERVER_ERROR`). If the status is `CMPLSERVER_BUSY`, the problem is put into the problem waiting queue until an empty solving slot is available or the maximum queuing time is reached. The procedure then stops automatically.

If the status is equal to `PROBLEM_FINISHED` the solution, the CMPL and the solver messages and if requested some statistics can be received by using `Cmpl.retrieve()`. The client sends its `jobId` and then retrieves the `CmplSolution`, `CmplMesages` and `CmplInfo` XML strings. If `Cmpl.knock()` returns `CMPLSERVER_ERROR`, the process is stopped.

The CMPLServer can be stopped by typing the command:

```
cmplServer -stop  [<port>]
```

5 GRID MODE

A CMPLGrid consists at least of one CMPLGridScheduler and usually a couple of CMPLServers that are connected to at least one scheduler. A CMPLGridScheduler is the gateway to the CMPLGrid for the clients and has to coordinate the traffic in the grid. That means it is responsible for the load balancing within the CMPLGrid and the assignment of the models to the connected CMPLServers. After receiving a model from a CMPLGridScheduler a CMPLServer communicates directly with the client to receive the model, to solve it and to send (if the problem is feasible) the solution(s), the CMPL and solver messages and if requested some information to the client. After these steps the client is disconnected automatically and the CMPLServer is waiting for the next problem from a CMPLGridScheduler.

The first step to start a CMPLGrid is to execute one or more CMPLGridSchedulers by typing the command:

```
cmplServer -startScheduler ↵
     [<port>] [-showLog]
```

As for the CMPLServers the parameter of a CMPLGridScheduler can be edited in the file `cmplServer.opt`. The relevant parameters for a CMPLGridScheduler with their default values are shown below.

```
cmplServerPort = 8008
maxServerTries = 3
schedulerServiceIntervall = 0.1
```

The parameter `port` specifies the default port of the CMPLGridScheduler. If one wants to run a CMPLServer on the same computer as the CMPLGridScheduler then the server needs to be started with a different port via command line argument. Since the CMPLGridScheduler has to call functions provided by connected CMPLServers with a high availability and failover, the CMPLGridScheduler repeats failed CMPLServer calls whereby the number of tries are specified by the parameter `maxServerTries`. There is also a service thread that works permanently after a couple of seconds defined by the parameter `serviceIntervall`.

After running one or more CMPLGridSchedulers, the involved CMPLServers can be started by typing the following command as also shown in Figure 4.

```
cmplServer -startInGrid [<port>] ↵
     [-showLog]
```

In addition to the described parameters in `cmplServer.opt` the following parameters are necessary for running a CMPLServer in a CMPLGrid.

```
...
maxServerTries = 3
performanceIndex = 1
cmplGridScheduler =
http://10.0.1.52:8008  4
```

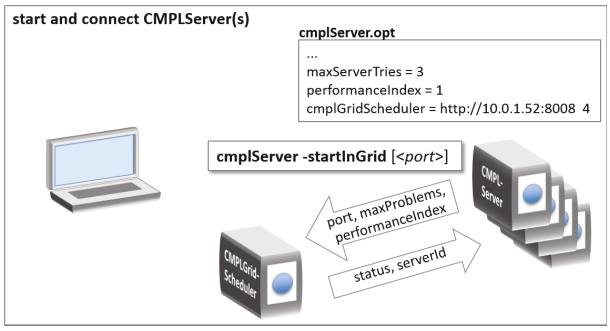

Figure 4: Start CMPLServer in grid mode

A CMPLServer in a CMPLGrid also has to call functions provided by a CMPLGridScheduler. Due to a maximum availability and failover the maximum number of tries of failed CMPLGridScheduler calls are specified with the parameter `maxServerTries`. Assuming heterogeneous hardware for the CMPLServers in a CMPLGrid it is necessary to identify several performance levels of the invoked CMPLServers for a reasonable load balancing. This can be done by the parameter `performanceIndex` that influences the load balancing function directly. The involved operators of the CMPLServers and the CMPLGridScheduler(s) should specify standardised performance classes used within the entire CMPLGrid with the simple rule: the higher the performance class, the higher the `performanceIndex`. The parameter `cmplGridScheduler` is intended to specify the CMPLGridScheduler with which the CMPLServer is to be connected. The first argument is the URL of the scheduler. The second parameter defines the maximum number of parallel problems that the CMPLServer provides to this CMPLGridScheduler. If a CMPLServer should be connected to more than one scheduler one entry per CMPLGridScheduler is required.

While connecting the CMPLGridScheduler the CMPLServer sends its port, the maximum number of provided problems and its performance index. It receives the status of the CMPLGridScheduler and a `serverId`. Possible states for connecting a CMPLServer are `CMPLGRID_SCHEDULER_OK` or `CMPLGRID_SCHEDULER_ERROR`.

Now a client can connect the CMPLGrid in the same way as a client connects a single CMPLServer either by using the CMPL binary

```
cmpl <problem>.cmpl -cmplUrl ↵
     http://<ip-or-domain>:<port>
```

or through the method `Cmpl.connect()` in pyCmpl and jCMPL programmes.

The client automatically sends the name of the problem and the name of the solver with which the problem should be solved to the CMPLGridScheduler.

If the solver is not available in the CMPLGrid the CMPLGridScheduler returns `CMPLSERVER_ERROR`. The status `CMPLGRID_SCHEDULER_BUSY` occurs when the grid is busy and the problem is assigned to the problem waiting queue. Otherwise, the CMPLGrid-

Scheduler returns the status `CMPLGRID_SCHEDULER_OK`, the `serverUrl` of the CMPLServer on which the problem will be solved and the `jobId` of the problem. This CMPLServer is determined on the basis of the load balancing function that is shown in Figure Figure 5. Per server that is providing the requested solver the current capacity factor is calculated by the relationship between the number of the current empty problems and the maximum number of provided problems.

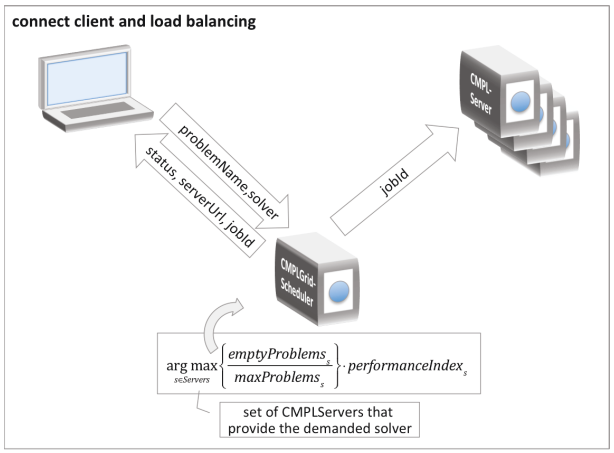

Figure 5: Load balancing

The number of empty problems is monitored by the CMPLGridScheduler with a lower bound of zero and an upper bound equal to the maximum number of provided problems. This parameter is decreased if the CMPLServer is taking over a problem and it is increased when the CMPLServer has finished the problem or the problem is cancelled. The idea is to send problems tendentiously to those CMPLServer with the highest empty capacity. To include the different performance levels of the invoked CMPLServers in the load balancing decision, the current capacity factor is multiplied by the performance index. The result is the load balancing factor and the CMPLServer with the highest load balancing factor is assigned to the client to solve the problem. This CMPLServer then gets the `jobId` of the CMPL problem by the CMPLGridServer in order to take over all relevant processes to solve this problem. Afterwards, the client is automatically connected to this CMPLServer.

The problem waiting queue handling is organised by the CMPLGridScheduler service thread that assigns the waiting problems automatically to CMPLServers by using the load balancing functionalities as described above. The waiting clients either ask automatically in the synchronous mode or manually in the asynchronous mode both through `Cmpl.knock()` until the received status is not equal to `CMPLGRID_SCHEDULER_BUSY`.

The next steps to solve the problem synchronously or asynchronously on the CMPLServer are similar to the procedures in the single server mode as shown Figure 6.

The methods `Cmpl.send()`, `Cmpl.knock()` and `Cmpl.retrieve()` are used to send the problem to the CMPLServer, to knock for the current status, to retrieve the solution and the CMPL messages as well as the solver messages and (if requested) some statistics. The main differences to the single server mode are that the CMPLServer calls the CMPLServerGrid to add an empty problem slot after finishing solving the problem and that the client is disconnected automatically from the CMPLServer after retrieving the solution, messages and statistics.

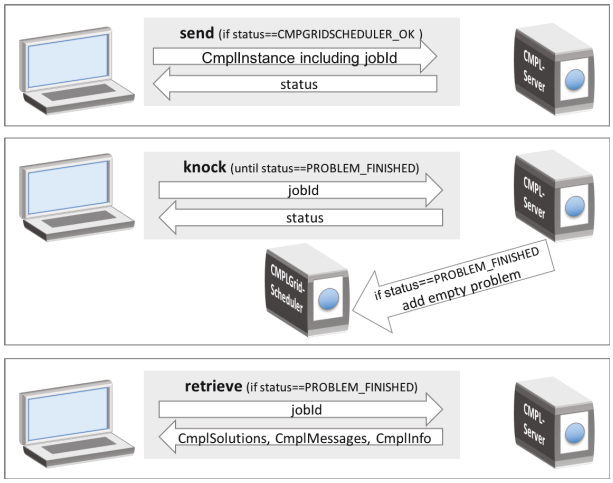

Figure 6: CmplGrid procedures

The CmplGridScheduler and the CmplServers can be stopped by typing the command:

```
cmplServer -stop  [<port>]
```

6 RELIABILITY AND FAILOVER

A distributed optimisation or a grid optimisation system is usually implemented in a heterogeneous environment. The network nodes can be installed on different hardware as well as on different operating systems. This fact could cause some disturbances within the optimisation network that should be either avoided or reduced in their negative impact on the optimisation processes.

Therefore, maximum reliability and failover are additional important targets of the CMPLServer and the CMPLGrid implementations. They are ensured by:

1. the problem queue handling on the CMPLGridScheduler and the CMPLServer,
2. multiple executions of failed server calls and
3. re-connections of problems to the CMPLGridScheduler if an assigned CMPLServer fails.

6.1 Problem queue handling

If a problem is connected to a CMPLServer or a CMPLGridScheduler and the number of running problems including the model sent is greater than `maxProblems`, it neither makes sense to cancel the problem nor to interrupt the solving process. Especially in case of an iterating solving process with a couple of depending problems it is the better way to refer the supernumerary problems automatically to the problem waiting queue.

In the single server mode, the problem queue handling is organised by the CMPLServer whilst in the grid mode the CMPLGridScheduler(s) are responsible for it. In both modes a problem stored in the problem waiting

queue has to wait until an empty solving slot is available or the maximum queuing time is reached.

In the single server mode, the number of problems that can be executed simultaneously on the particular CMPLServer are defined by the parameter `maxproblems` in the option file `cmplServer.opt`. With this parameter it both should be avoided to overwhelm the server and the super-proportional effort of managing a huge amount of parallel problems. The first empty solving slot that appears when a running problem is finished or cancelled, is taking over a waiting problem by using the FIFO approach.

The number of simultaneously running problems in a CMPLGrid is defined by the sum over all connected CMPLServers of the maximum number of problems provided by the servers. This parameter has to be defined per CMPLServer in `cmplServer.opt` as second argument in the entry `cmplGridScheduler = <url> <maxProblems>`. The CMPLGridScheduler counts the number of running problems per CMPLServer in relation to its maximum number of provided problems. If it is not possible to find a connected CMPLServer with an empty solving slot, then the problem is put to the problem waiting queue. In contrast to the single server mode the problem which has been waiting longest is not executed automatically by the first appearing free CMPLServer. The next next running CMPL problem is chosen by the described load balancing function over the set of CMPLServers that stated an empty solving slot during two iterations of the CMPLGridScheduler service thread.

The client's maximum queuing time in seconds can be specified with the CMPL command line argument `-maxQueuingTime <sec>`. This argument can also be set as CMPL header entry or in pyCMPL and jCMPL with the method `Cmpl.setOption([option])`. The default value is 300 seconds.

6.2 Multiple executions of failed server calls

To avoid that a single execution of a server method, which fails due to network problems like socket errors or others, cancels the entire process, all failed server calls can be executed again several times.

As a necessary parameter the maximum number of executions of failed server calls can be specified for the clients with the CMPL command line argument `-maxServerTries <tries>` or in the CMPL model as CMPL header entry or in pyCMPL and jCMPL by using `Cmpl.setOption([option])`. The default value is 10.

The number of maximum executions of failed server calls in the communication between the CMPLGridScheduler and CMPLServers is defined in `cmplServer.opt` with the entry `maxServerTries = <tries>`.

An exemplary and simplified implementation of this behaviour is shown in the pseudo code listing below (Listing 9).

In a first step, the variable `serverTries` is assigned zero. The call of the server method (line 4) is imbedded in an infinite loop (lines 2-13) and in a try-except-block for the exception handling (lines 3-11). If no exception occurs, then the loop is finished by the break command in line 12. Otherwise `serverTries` is incremented by 1. If the maximum number is not exceeded (line 7) the server method is called again (line 4). If `serverTries` is greater than `maxServerTries` then the class variable `Cmpl.status` is set to `CMPLSERVER_ERROR` and a `CmplException` is raised that has to be handled in the code in which the listing below is imbedded (lines 7-9).

6.3 Re-connections of failed problems to the CMPLGridScheduler

Multiple server calls are mainly intended to prevent network problems. But it could be also possible that other problems caused by CMPLServers connected to a CMPLGridScheduler (e.g. a failed execution of a solver, file handling problems at a CMPLServer or the unpredictable shutdown of a CMPLServer) occur. The idea to handle such problems is to reconnect the particular CMPL problem to the CMPLGridScheduler if the CMPLServer fails and to assign the problem to another CMPLServer automatically.

The pseudo code in Listing 9 describes a simplified implementation of `Cmpl.solve()` only for the grid mode to illustrate this approach.

As in the listing of the multiple server calls the variable `serverTries` is assigned zero (line 1). The entire method is also imbedded in an infinite loop (lines 2-37) and the exception handling is organised as try-except-block (lines 3-36).

Before `Cmpl.solve()` is called the client has to execute `Cmpl.connect()` successfully. Therefore the class variable `Cmpl.status` has to be unequal to `CMPLSERVER_ERROR` and an additional execution of `Cmpl.connect()` is not necessary in the first run of `Cmpl.solve()` (lines 4-6).

It is possible that the entire CMPLGrid is busy (`CMPLGRID_SCHEDULER_BUSY`) (line 8) and the problem is moved to the CMPLGridScheduler problem waiting queue. In this case the problem has to wait for the next empty solving slot via `Cmpl.knock()` (line 10) until the CMPLGridScheduler returns the status `CMPLGRID_SCHEDULER_OK` (line 9) or the waiting time exceeds the maximum queuing time and a `CmplException` is raised (lines 11-13).

After this loop the problem is automatically connected to a CMPLServer within the CMPLGrid. The class variable `Cmpl.connectedToServer` is assigned `True` (line 16) and the problem is sent to this server through `Cmpl.send()` (line 18). The problem then has to wait until the problem status is `PROBLEM_FINISHED` (lines 20-22). As soon as the problem is finished, the solution(s), the CMPL and the solver messages as well as (if requested) some statistics can be retrieved via `Cmpl.retrieve()` (line 24). If no

```
01 serverTries=0
02 while True do
03     try
04         callServerMethod()
05     except
06         serverTries+=1
07         if serverTries>maxServerTries then
08             status=CMPLSERVER_ERROR
09             raise CmplException("calling CmplServer function failed")
10         end if
11     end try
12     break
13 end while
```

Listing 9: Pseudo code for multiple executions of failed server calls

```
01 serverTries=0
02 while True do
03   try
04     if status==CMPLSERVER_ERROR then
05       CmplGridScheduler.connect()
06     end if
07
08     if status==CMPLGRID_SCHEDULER_BUSY then
09       while status<>CMPLGRID_SCHEDULER_OK do
10         CmplGridScheduler.knock()
11         if waitingTime()>=maxQueuingTime then
12          raise Exception("max. queuing time is exceeded.")
13         end if
14       end while
15     end if
16     connectedToServer=True
17
18     CmplServer.send()
19
20     while status<>PROBLEM_FINISHED do
21       CmplServer.knock()
22     end while
23
24     CmplServer.retrieve()
25     break
26
27   except
28     serverTries+=1
29     if status==CMPL_ERROR and connectedToServer==True then
30       CmplGridScheduler.cmplServerFailed()
31     end if
32     if serverTries>maxServerTries or status==CMPLGRID_SCHEDULER_BUSY then
33       ExceptionHandling()
34       exit
35     end if
36   end try
37 end while
```

Listing 9: Re-connections of failed problems to the CMPLGridScheduler

`CmplException` or another exception appeared during these procedures the infinite loop is left by the break command in line 25.

If during these procedures a `CmplException` or other exceptions occur this has to be handled in the except block in the lines 27-36. The first step is to increase the number of failed server call tries (line 28). If the problem is connected to a CMPLServer and a `CmplException` is raised on the server (status equals `CMPL_ERROR`) then the client reports via `CMPLGridScheduler.cmplServerFailed()` that this CMPLServer failed (line 30). This CMPLServer is then excluded from the CMPLGridScheduler load balancing until CMPLGridScheduler's service thread recognises that this CMPLServer is able to take over problems again.

If the number of failed server calls exceeds the maximum number of tries or the status is `CMPL-`

`GRID_SCHEDULER_BUSY` because the maximum queuing time is exceeded (line 32), the entire procedure stops by doing the necessary exception handling and by exiting the programme (lines 33-34).

Otherwise the problem has to pass the loop again. That means that the problem is reconnected to the CMPLGrid via `CMPLGridScheduler.connect()` (lines 4-6) and the solving process starts again.

7 PERFORMANCE TESTS

One of the main targets of this project was to ensure a high performance. This section describes a performance test through that the performance and the efficiency of the CMPLGrid is analysed.

There are two conflicting effects on the total computing time of a set of simultaneous problems.

The first positive effect is the decreased total computing time due to the use of an increased number of computing nodes in a grid of CMPLServers. The more CMPLServers in a CMPLGrid exist the more computing nodes can be used and the number of problems that have to be solved simultaneously on each node are decreased. It can be assumed that in an ideal environment the total computing time declines linearly. This effect is shown with an example in Table 1.

number of optimisation nodes	maximum number of parallel problems per node	normalised theoretical minimum computation time
1	10	100
2	5	50
3	4	40
4	3	30
5	2	20

Table 1: Theoretical computation time

Having scenarios in the range of one to up to five optimisation nodes the maximum number of problems is halved in the scenario with two optimisation nodes and then decreased by one problem per node by adding an additional node. The normalised theoretical minimum computation time corresponds with the number of the optimisation nodes.

But this positive effect has to be paid by the increased time spent on managing the problems within the grid and by the network traffic times itself.

To analyse these two conflicting effects, a performance test is realised in which the average total computation time and the average normalised computation time of a pyCMPL script that executes ten identical CMPL models simultaneously in separate threads are measured for seven scenarios and five different CMPL models. The first two scenarios are the standalone scenario in which the problems are solved locally and the single CMPLServer scenario in which one CMPLServer takes over all optimisation tasks. The following scenarios are CMPLGrids with one CMPLGridScheduler coordinating one to up to five CMPLServers. The parameters of the CMPL models are shown in Table 2. In all scenarios (excluding the standalone scenario) an additional network node works as the client for the CMPLServer or the CMPLGrid. CBC was chosen as solver for all scenarios.

All performance tests are made on a (self-made) Raspberry Pi cluster computer that is shown in Figure Figure 7. The Raspberry Pi is a low cost, credit-card sized computer with a Broadcom BCM2835 system on a chip. (http://www.raspberrypi.org/) A Raspberry Pi is definitely not a good environment for optimisations due to its restricted hardware but it is perfect for software tests. It is possible to analyse all effects of shipping optimisation problems into a grid of CMPLServers and the corresponding network traffic with a standardised hardware that costs roughly USD 500.

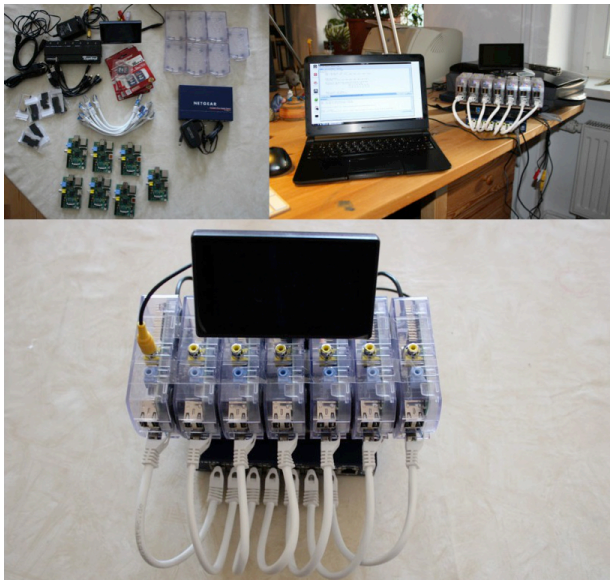

Figure 7: Raspberry Pi cluster computer

As shown in Table 2 and in Figure 8 the CMPL models vary in their size and in the average total computation time. All models need more computation time in the Scenario with one CMPLServer compared to the standalone scenario. This effect is caused by the coordinating effort on the CMPLServer and the additional time which all models need for the network traffic between the client and the CMPLServer. By using a CMPLGrid with one CMPLServer the average total computation time is less than the time needed in the single server scenario. The reason for this effect is that the CMPLGridScheduler coordinates the models sent by the client whereby the CMPLServer only solves the problems whereby in the single server scenario the CMPLServer is responsible for both coordinating and solving the models. For all other scenarios it can be shown that the average total computation time declines by adding more optimisation nodes into the CMPLGrid whereby it seems that the CMPLGrid scales linearly. This behaviour can be proved with the average normalised computation times shown in Figure 9.

The standalone scenario is the basis with a target of 100% normalised computation time.

short name	description	number of variables	number of constraints	non zeros
mcdm	Goal programming model with Euclidean distance measure	74 (30 integer)	107	01 (3.80\%)
tsp18	Traveling salesman problem with 18 cities	341 (all integer)	308	1464 (1.39%)
tsp20	Traveling salesman problem with 20 cities	419 (all integer)	382	1826 (1.14%)
tsp22	Traveling salesman problem with 22 cities	505 (all integer)	464	2228 (0.95%)
warehouse	Warehouse location problem	3600 (all integer)	3659	10680 (0.08%)

Table 2: Overview of the tested problems

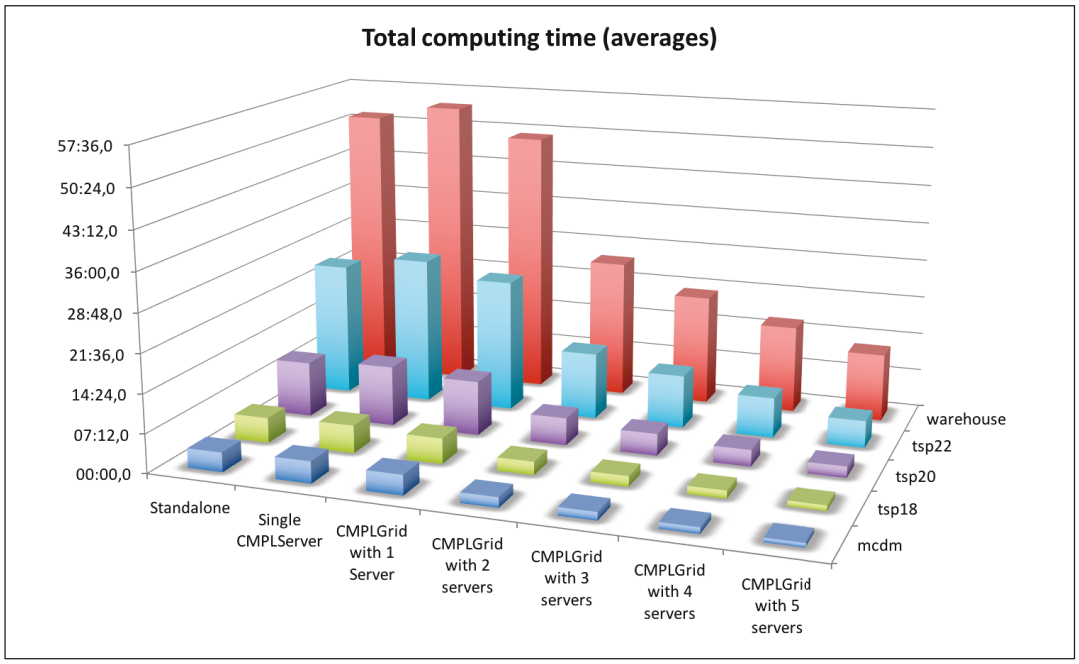

Figure 8: Total computing time (averages)

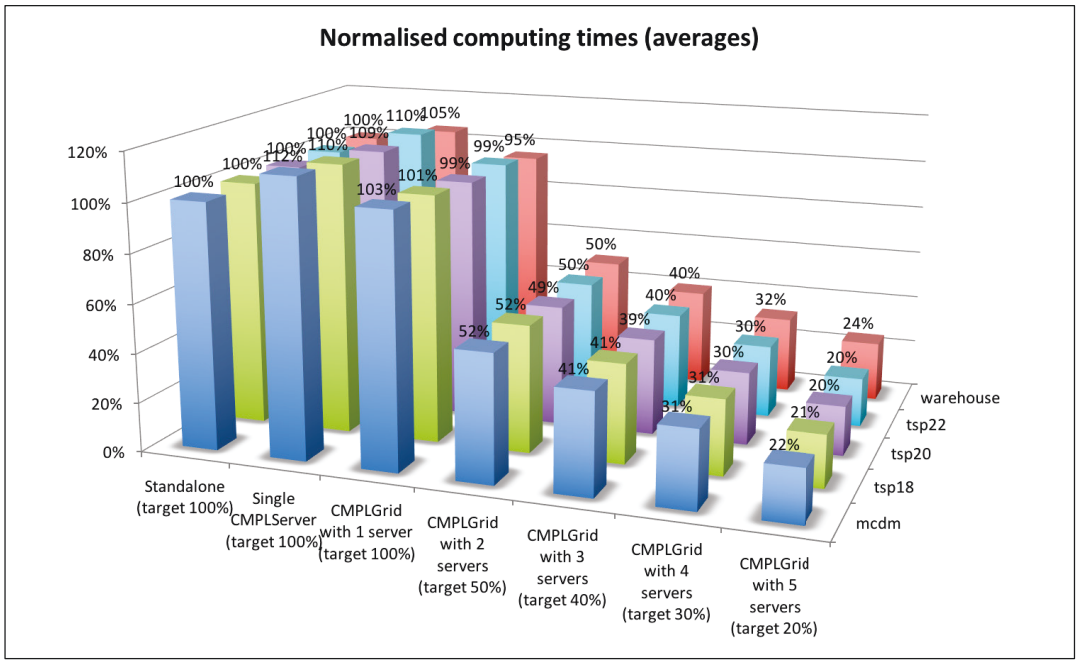

Figure 9: Normalised computing time (averages)

All CMPL models in the single CMPLServer scenario need more than 100% normalised computation time due to the additional network traffic and the coordination effort. The average normalised computation times of the CMPLGrid scenario with only one CMPLServer are approximately 100%. As mentioned before, this effect mainly depends on the share of the coordination and solving tasks between the scheduler and the server whilst in the single server scenario the CMPLServer is responsible for both. It seems somehow irritating that some of the CMPL models need less than 100%. But this effect is caused by the limited hardware of a Raspberry Pi which is overwhelmed in the standalone scenario by coordinating ten simultaneous models in separate threads on a single-core CPU and by swapping the memory. In all of the CMPLGrid scenarios all CMPL models nearly meet the targets of the theoretical minimum of the normalised computation time. That means that a CMPLGrid scales linearly for all tested CMPL models that vary in their size and structure.

It can be summarised that the target of a huge performance and a high efficiency seems to be reached.

8 SUMMARY

Since the information and communication technologies has been changed by the internet technologies, CMPL was faced (as other optimisation software packages too) with the necessity of distributed optimisation. Therefore, the CMPLServer which is an XML-RPC-based web service for distributed and grid optimisation was created. The aim of this article was to explain CMPLServer.

After an overview of the main functionalities, the XML-based file formats (`CmplInstance`, `CmplSolutions`, `CmplMessages`, `CmplInfo`) for the communication between a CMPLServer and its clients were described. A `CmplInstance` file contains an optimisation problem formulated in CMPL, the corresponding sets and parameters in the `CmplData` file format as well as all CMPL and solver options that belong to the CMPL model. If the model is feasible and a solution is found, then a `CmplSolutions` file contains the solution(s) and the status of the invoked solver. The `CmplMessages` file is intended to provide the CMPL status and (if existing) the CMPL error or warning messages. A `CmplInfo` file is an XML file that contains (if requested) several statistics and the generated matrix of the CMPL model.

A CMPLServer can be used in a single server mode or in a grid mode which were described in the following sections.

In the single server mode only one CMPLServer exists in the network and can be accessed synchronously or asynchronously by the clients. The client sends the model to the CMPLServer and then waits for the results. If the model is feasible and an optimal solution is found the solution(s) can be received. If the model contains syntax or other errors or if the model is not feasible the CMPL and solver messages can be obtained. Whereby in the synchronous mode the client has to wait after sending the problem for the results and messages in one process. A model can also be solved asynchronously in several steps. It seems reasonable to use the single server mode if a large model is formulated on a thin client in order to solve it remotely on a CMPLServer that is installed on a high performance system.

The grid mode extends this single server mode by coupling CMPLServers from several locations and at least one coordinating CMPLGridScheduler. For the client there does not appear any difference whether there is a connection made to a single CMPLServer or to a CMPLGrid. The client's model is connected with the same functionalities as for a single CMPLServer to a CMPLGridScheduler which is responsible for the load balancing within the CMPLGrid and the assignment of the model to one of the connected CMPLServers. After this step the client is automatically connected to the chosen CMPLServer for one optimisation run and the model can be solved synchronously or asynchronously. A CMPLGrid should be used for handling a huge amount of large scale optimisation problems. An example can be a simulation in which each agent has to solve its own optimisation problem at several times. An additional example for such a CMPLGrid application is an optimisation web portal that provides a huge amount of optimisation problems.

A distributed optimisation or a grid optimisation system is usually implemented in a heterogeneous environment that could cause some disturbances within the optimisation network. That should be either avoided or reduced in their negative impact on the optimisation processes. Therefore, maximum reliability and failover are additional important targets of the CMPLServer and the CMPLGrid implementations. In a next section, it was explained that these targets can be ensured by the problem queue handling on the CMPLGridScheduler and the CMPLServer, multiple executions of failed server calls and re-connections of problems to the CMPLGridScheduler if an assigned CMPLServer fails.

In the last section, a performance test to measure the performance and the efficiency of the CMPLGrid was described. It was shown that CMPLGrid scales linearly without an impact of the size or structure of the CMPL model.

REFERENCES

Coulouris, G.F., J. Dollimore, T. Kindberg, G. Blai. 2012. Distributed Systems: Concepts and Design. 5th ed. Addison-Wesley.

Czyzyk, J., M. P. Mesnier, J. J. Moré. 1998. The neos server. IEEE Journal on Computational Science and Engineering 5 68–75.

Dolan, Elizabeth D., Robert Fourer, Jean-Pierre Goux, Jason Sarich Todd S. Munson. 2008. Kestrel: An interface from optimization modeling systems to the neos server. INFORMS Journal on Computing 20 525–538.

Foster, I., C. Kesselman. 2004. The Grid2: 2nd Edition: Blueprint for a New Computing Infrastructure. Kindle edition ed. Morgan Kaufmann Publishers Inc.

Fourer, Robert, Jun Ma, R. Kipp Martin. 2010. Optimization services: A framework for distributed optimization. Operations Research 58 1624–1636.

Kshemkalyani, Ajay D., Mukesh Singhal. 2008. Distributed Computing: Principles, Algorithms, and Systems. 1st ed. Cambridge University Press.

Laurent, S. St., J. Johnston, E. Dumbill. 2001. Programming Web Services with XML-RPC. 1st ed. O'Reilly.

Steglich, M. and Schleiff, Th. 2010. "CMPL: Coliop Mathematical Programming Language." In: Wildauer Schriftenreihe - Entscheidungsunterstützung und Operations Research, Beitrag 1, Technische Hochschule Wildau [FH].

AUTHOR BIOGRAPHIE

Mike Steglich is a Professor of Business Administration, Quantitative Methods and Management Accounting in the Faculty of Business, Computing, Law of the Technical University of Applied Sciences Wildau in Germany. He is also one of the authors and distributors of the open source project CMPL (<Coliop|Coin> Mathematical Programming Language).