DOCUMENTOS DE TRABAJO

BILTOKI

D.T. 2012.01

MPI parallel programming of mixed integer optimization problem using CPLEX with COIN-OR

Unai Aldasoro, M.Araceli Garín, María Merino y Gloria Pérez

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Facultad de Ciencias Económicas.
Avda. Lehendakari Aguirre, 83
48015 BILBAO.

Documento de Trabajo BILTOKI DT2012.01
Editado por el Departamento de Economía Aplicada III (Econometría y Estadística)
de la Universidad del País Vasco.

ISSN: 1134-8984

MPI parallel programming of mixed integer optimization problems using CPLEX within COIN-OR

Unai Aldasoro[1], **María Araceli Garín**[2], **María Merino**[3] **and Gloria Pérez**[3]
[1]Dpto. Matemática Aplicada,
Universidad del País Vasco, UPV/EHU
Bilbao (Bizkaia), Spain
e-mail: unai.aldasoro@ehu.es
[2]Dpto. de Economía Aplicada III,
Universidad del País Vasco, UPV/EHU
Bilbao (Bizkaia), Spain
e-mail: mariaaraceli.garin@ehu.es
[3]Dpto. de Matemática Aplicada, Estadística e Investigación Operativa ,
Universidad del País Vasco, UPV/EHU
Leioa (Bizkaia), Spain
e-mail: maria.merino@ehu.es, gloria.perez@ehu.es

Abstract

The aim of this technical report is to present some detailed explanations in order to help to understand and use the Message Passing Interface (MPI) parallel programming for solving several mixed integer optimization problems. We have developed a C++ experimental code that uses the IBM ILOG CPLEX optimizer within the COmputational INfrastructure for Operations Research (*COIN-OR*) and MPI parallel computing for solving the optimization models under UNIX-like systems. The computational experience illustrates how can we solve 44 optimization problems which are asymmetric with respect to the number of integer and continuous variables and the number of constraints. We also report a comparative with the speedup and efficiency of several strategies implemented for some available number of threads.
Keywords: Optimization, Message Passing Interface, Parallel Computing, COIN-OR Open Solver Interface, CPLEX optimizer.

[*]This research has been partially supported by the project ECO2008-00777 ECON from the Spanish Ministry of Education and Science, and Grupo de Investigación IT-347-10 from the Basque Government.

1 Introduction

Parallel computing is a Computer Science area that studies the necessary hardware and software aspects to perform simultaneous execution of tasks. Currently this discipline is the prominent paradigm on high performance computing and also on computer architecture due to the industry's shift to multicore processors.

At hardware level, the parallel era starts at the late 1950's in the form of shared memory multiprocessor supercomputers. The development of this type of computers continued until the early 1980s when a new paradigm of massively parallel multiprocessors (MPPs) arrived. MPPs showed a significantly better performance and became dominant of high performance computing. From late 1980's on computing clusters appear by linking a large numbers of off-the-shelf computers connected by an off-the-shelf network. Nowadays parallel computing is mainly based on clusters and multicore processors. For deeper information about parallel computing hardware consider [3] and [6]. The top 500 website gathers every six months the information concerning the top 500 supercomputing sites at world level .

This cooperation among processors can be of different nature depending on the way processors exchange information. The basic parallel architectures correspond to *shared memory* and *message passing* (or distributed memory). As described on [8] "one processor of a shared memory machine can communicate with another by writing the information into a global shared memory location and having the second processor read directly from that location" using a bus. So the communication is carried out by *shared variables*. This makes inter-processor communication very easy and fast but produces problems in terms of simultaneous access of a unique memory location. On the other hand on message-passing paradigm each processor has its own local memory connected by a network with the rest of processors, the communication is based on message-passing.

On mathematical optimization, we often find large-scale problems or we need to solve many of them, which can not be done efficiently by a single processor, even if technology is constantly improving. Parallel computing aims to be a powerful tool to manage this type of problems, operating on the principle that large problems can often be divided into smaller ones. The idea is simple: to use p processors in cooperation in order to (ideally) be able to solve a problem, to solve a problem p times faster or to solve a p times bigger problem using the same amount of time.

In this technical report we show how to solve many problems by a message-passing paradigm code, because the solving processor of each model is completely independent from the rest of them, so the transfer of information is not significant. The computational experience will be carried out in

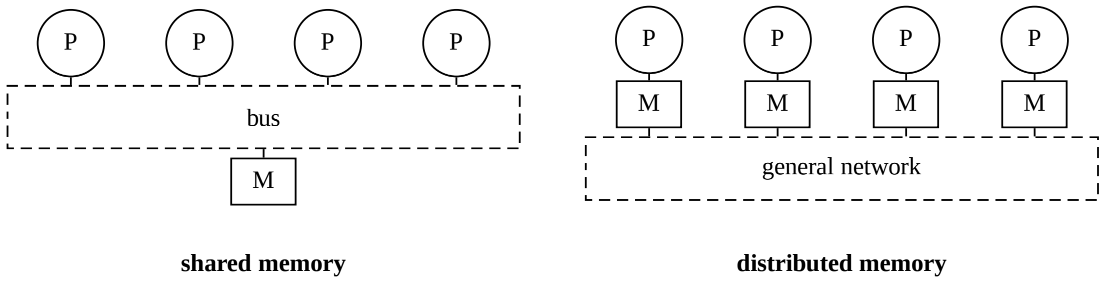


Figure 1: Shared memory and distributed memory architectures (P: processor, M: memory)

the cluster ARINA (see [1]), that provides an extensive environment for message-passing architecture.

The presented code have been programmed with the MPI library. As described on [10] "the Message-Passing Interface or MPI is a library of functions and macros that can be used in C, FORTRAN and C++ programs. As its name implies MPI is intended for use in programs that exploit the existence of multiple processors by messange-passing". Developed in 1993 is one of the first standards for parallel programming and it is the first that is based on message-passing.

This work aims to present a general environplment for parallel solving of several optimization problems using directly the IBM ILOG CPLEX optimizer (see [7]) within the COmputational INfrastructure for Operations Research (see [2] and [9]). The main objective of this work is to describe the MPI parallelization for solving optimization problems. In the computational experience, we show the evolution of the computing time depending on the number of processors used and the selected solver, where the combination of a general MPI parallelization with the internal ILOG CPLEX parallelization has been considered.

The remainder of the technical report is organized as follows. In Section 2 appears the context of the problem that we are interested in solving. Section 3 shows the main MPI sintaxis for parallel computing and Section 4 shows the paralleling strategy. The main ideas for MPI execution under UNIX-like systems are presented in Section 5. Section 6 reports the computational experience using COIN-OR and CPLEX. In the Appendix 1 we present the C++ code in detail for the illustrative example. An example of Makefile for compiling C++ code with MPI environment is given in the Appendix 2.

2 Context

Let us consider the following mixed 0-1 optimization problem

$$\begin{aligned} z = \min \quad & ax + by \\ s.t. \quad & Ax + By = h \\ & x \in \{0,1\}^{nx}, \\ & y \in \mathbb{R}^{+ny}, \end{aligned} \tag{1}$$

where x and y are the nx and ny dimensional vectors of the 0-1 and continuous variables, a and b are the vectors of the objective function coefficients, A and B are the constraint matrices, respectively, and h the right-hand-side.

As we have mentioned previously, we can consider two situations: to solve a large-scale optimization problem, that can be decomposed in q smaller subproblems or to solve a huge number q of independent mixed-integer problems. In any case, how can we solve $z^1, z^2, \ldots, z^q$ in non-sequencial order?

We will use the MPI parallel programming for solving q optimization problems, that will be read in Mathematical Programming System (MPS), which is a standard format for linear and mixed integer optimization. We will compare the COIN-OR optimization solver and the CPLEX optimizer under COIN-OR, with sequential and parallel programming. It is important to notice that the ILOG

CPLEX optimizer also includes internally a parallel environment for solving an optimization model. The number of parallel threads (on shared memory) that CPLEX may use for any invocation of a parallel algorithm is controlled by the threads parameter `CPX_PARAM_THREADS`.

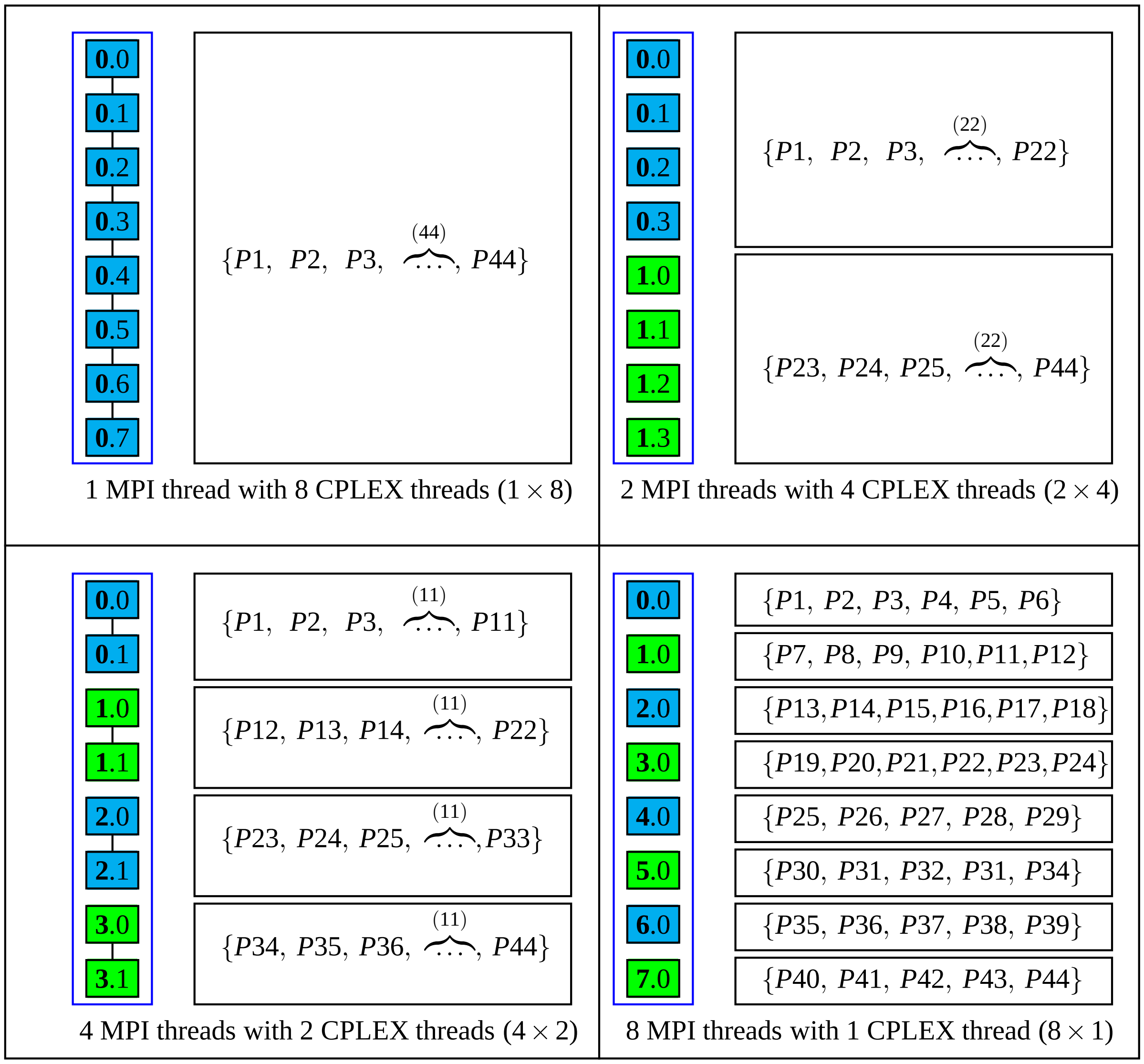

0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
$\{P1,\ P2,\ P3,\ \overbrace{\dots}^{(44)},\ P44\}$
1 MPI thread with 8 CPLEX threads (1×8)
0.0
0.1
0.2
0.3
1.0
1.1
1.2
1.3
$\{P1,\ P2,\ P3,\ \overbrace{\dots}^{(22)},\ P22\}$
$\{P23,\ P24,\ P25,\ \overbrace{\dots}^{(22)},\ P44\}$
2 MPI threads with 4 CPLEX threads (2×4)
0.0
0.1
1.0
1.1
2.0
2.1
3.0
3.1
$\{P1,\ P2,\ P3,\ \overbrace{\dots}^{(11)},\ P11\}$
$\{P12,\ P13,\ P14,\ \overbrace{\dots}^{(11)},\ P22\}$
$\{P23,\ P24,\ P25,\ \overbrace{\dots}^{(11)}, P33\}$
$\{P34,\ P35,\ P36,\ \overbrace{\dots}^{(11)},\ P44\}$
4 MPI threads with 2 CPLEX threads (4×2)
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
$\{P1,\ P2,\ P3,\ P4,\ P5,\ P6\}$
$\{P7,\ P8,\ P9,\ P10, P11, P12\}$
$\{P13, P14, P15, P16, P17, P18\}$
$\{P19, P20, P21, P22, P23, P24\}$
$\{P25,\ P26,\ P27,\ P28,\ P29\}$
$\{P30,\ P31,\ P32,\ P31,\ P34\}$
$\{P35,\ P36,\ P37,\ P38,\ P39\}$
$\{P40,\ P41,\ P42,\ P43,\ P44\}$
8 MPI threads with 1 CPLEX thread (8×1)

Figure 2: Strategies: MPI threads with CPLEX threads (8 threads available for solving 44 problems)

If we consider cores with 8 threads, we can solve the q problems in different ways depending on the number of threads for distributed memory (MPI threads) and the number of threads for shared memory (CPLEX threads). For example, we will consider $q = 44$ problems, so- called $P1$, $P2$, $\dots$, $P44$, we could use 8 MPI threads and sequential CPLEX; or 4 MPI threads and 2 CPLEX threads; or 2 MPI threads and 4 CPLEX threads; or 1 MPI thread and 8 CPLEX threads, among others. We illustrate the different executions in the Figure 2.

It is important to notice that not all thread have the same role. We denote **primary threads** to those who execute a copy of the executable, that is, the threads working on a distributed memory paradigm and linked by MPI functions. One of the primary threads will be denoted **coordinator thread**, this specific thread will be the one who gather the distributed information. On the other hand **auxiliary threads** will take part on the process only when the CPLEX optimizer requires their participation for a shared memory solving. Each auxiliary thread is linked to a specific primary thread. The corresponding notation will be applied at the different diagrams of this technical report and it is summarized on the following lines.

Considering $n+1$ MPI threads and $m+1$ CPLEX threads:

Coordinator thread	$\mathbf{0}.0$	
Primary threads	$\mathbf{i}.0$	$i = 0, \dots, n$
Auxiliary threads	$\mathbf{i}.j$	$i = 0, \dots, n,\ j = 1, \dots, m$

3 MPI sintaxis

Let us start the description of the MPI environment by underlining the three main phases that appear at machine level on a message-passing paradigm, see [10]. These phases are not directly executed by the user, they are automatically performed by the machine.

Phase a: The user issues a directive to the operating system which has the effect of placing a copy of the executable program on each processor.

Phase b: Each processor begins execution of its copy of the executable.

Phase c: Different processors can execute different statements by branching within the program. Typically branching will be based on processor ranks.

The main idea behind these phases is that every primary thread receives an exact copy of the executable. The code is executed in parallel but we would like different primary threads to execute different statements. This is achieved by a rank based branching, in other words, defining the rank of the processor that will perform a specific statement or by branching data vectors by rank. This allows to work on a Single Program Multiple Data paradigm.

The implementation of the corresponding executable supports different experimental codes. On the following lines we present the general structure of a MPI implementation in C++ (see [10]). This layout can be divided in five groups by considering the nature of the functions they use.

```
...
#include "mpi.h"
...
main(int argc, char **argv) {
...
  // Group 1: Declaring MPI variables.
  ...
  // Group 2: Beginning of the MPI environment (No MPI functions called before this).
  ...
  // Group 3: Functions controling the number of processes.
  ...
  // Group 4: Communication functions.
  ...
  // Group 5: End of the MPI environment (No MPI functions called after this).
  ...
...
}
...
```

The following subsections will describe the objective of each group and will use as practical example sentences of the C++ code added as Appendix 1.

3.1 Group 1: Declaring MPI variables

In this part the specific variables related to the MPI environment are declared. This section may not appear in every MPI code since a big part of MPI functions have as input or output common C++ variables.

Among the most frequent MPI variables we find **MPI_Group** and **MPI_Comm**. The first one creates *groups* of processor whereas the second creates *communicators*, that is, a group of processors plus a context of communication, in other words, a collection of processors that can send messages to each other.

By default, a *communicator* called MPI_COMM_WORLD is created at the beginning of each program and it consists of all the active processors at the execution. This turns very useful for global communications but frequently it is simpler to consider also subgroups of processors. That is the case of the considered code.

As it is shown at the Appendix 1the following MPI variables are declared at *mainmpi.cpp*:

```
MPI_Group  orig_group,new_group;
MPI_Comm   new_comm;
```

For other MPI variables consider [10] and [12]

3.2 Group 2: Beginning of a MPI environment

```
MPI_Init(&argc,&argv);
```

MPI_Init corresponds to the first MPI function that must be executed on a program. It allows the system to use the MPI library and therefore its features. It must be called just once and no MPI functions can be called before this.

3.3 Group 3: Functions controlling the number of processors

In order to have a functional point-to-point communication environment it is essential to identify each processor with a *rank* number, so that we can easily specify the sender and the receiver. Additionally, the *rank* allows the programmer to work on a *Single Program Multiple Data* (SPMD) paradigm, that implies that with a single program different processor execute different tasks. Schematically:

```
if  (rank == 0) DO X
else if (rank == 1) DO Y
```

To do so, the MPI library provides the **MPI_Comm_rank** function:

```
MPI_Comm_rank(MPI_COMM_WORLD,&original_rank);
```

The first argument defines the *communicator* and the corresponding *rank* of the processor is stored on the second argument. Note that a processor has *rank*, in general different, for each *communicator* that is involved on.

Frequently, the tasks to be executed are divided among the available processors. The function **MPI_Comm_size** allows us to determine the number of processors in a communicator (stored on the second argument):

```
MPI_Comm_size(MPI_COMM_WORLD,&original_size);
```

How can we create a new communicator considering only active processors?

Let us consider the case where the number of processors is bigger than the number of submodels to solve, that is *nmodel* < *original_size*. That will mean that some of the processor will not actually solve any subproblem. In order to simplify the processor of gathering the final solutions, we have created a new group considering only the active threads. The necessary steps will be:

Extract handle of global group from MPI_COMM_WORLD using **MPI_Comm_group**. The handle is stored on the second argument.

```
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
```

Form a new group as a subset of global group using **MPI_Group_incl**.

```
if (nmodel < original_size) {
MPI_Group_incl(orig_group, nmodel, ranks1, &new_group);
} else {
MPI_Group_incl(orig_group, original_size, ranks2, &new_group);
}
```

ranks1 is an array of ***nmodel*** elements containing the *ranks* of the processors to be part of the new group. The handle of the new group is stored on ***&new_group***. On the other hand, ***ranks2***

contains the ranks of the *MPI_COMM_WORLD*, so notice that if *nmodel* $\nless$ *original_size* the new group will correspond to the existing *MPI_COMM_WORLD* group.

Let us create a new communicator for the new group using **MPI_Comm_create**. This function creates the new communicator ***&new_comm*** with the processor ***new_group*** extracted from communicator *MPI_COMM_WORLD*.

```
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
```

3.4 Group 4: Communication functions

Message-passing communication is the core of a MPI environment. The basic functions to develop such an exchange correspond to MPI_Send and MPI_Recv. The first function sends a message to a designated processor whereas the second receives a message from a processor. As mentioned on [10] in order for the message to be succesfully communicated the system must append some information to the data that the application program wishes to transmit. This information is called the **envelope of a MPI message**. It will contain the following basic information:

- The rank of the receiver
- The rank of the sender
- A tag
- A communicator

Every message-passing function defines as arguments the previous aspects. As example we will consider the specific functions used at the Appendix 1 code: **MPI_Gatherv** and **MPI_Allgatherv**.

Analysing the parallel computing strategy we will see that each submodel is assigned to a processor. After the solving task is finished we will gather the value of the objective function at the submodel ranked 0. The corresponding function will be defined as follows:

```
MPI_Gatherv(&zq_loc[0], assignment[pid], MPI_INT,
&zq[0],assignment, inicial, MPI_INT, 0, new_comm);
```

Where **MPI_Gatherv** is used to gather information of type vector in a specific processor (for scalar values use MPI_Gather). The first three arguments define the vector to be sent, that is, a vector stored from *&zq_loc[0]* of length *assignment[pid]* and of type *MPI_INT* (in other words, a vector of integer values). The processor ranked *0* at the *new_comm* communicator will receive a vector of type *MPI_INT* from each processor of the same communicator. The size of the received vectors is defined at the vector *assignment* and their position at the new vector *&zq[0]* is established by the vector *inicial*.

Additionally we want all the processors to have access to the presolve status. To do so we will use the **MPI_Allgatherv**. This function is an extension of **MPI_Gatherv**, since the same structure is repeated for every processor, in other words, the gather vector will be stored in every processor. Note that the only difference at argument level is that this function does not ask to define the receiver rank.

```
MPI_Allgatherv(&istruef969_loc[0], assignment[pid], MPI_INT,
&istruef969[0],assignment, inicial, MPI_INT, new_comm);
```

Notice that the Appendix 1 code also includes the **MPI_Allreduce** function. It has a double nature since it combines values from all processors and distributes the result back to all processors. In this particular case all processors will store at the *ncols* variable the value obtained by taking the maximum value (MPI_MAX) among the local *ncols_max_loc* variables.

```
MPI_Allreduce(&ncols_max_loc,&ncols, 1, MPI_INT, MPI_MAX,new_comm);
```

The MPI library provides a rich variety of functions for message-passing tasks. For further information see [12].

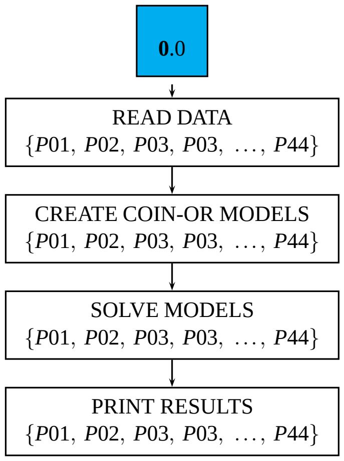
3.5 Group 5: End of a MPI environment

```
MPI_Finalize();
```

MPI_Finalize corresponds to the last MPI function that must be executed on a program. It cleans up unfinished tasks and closes the interaction with the MPI library. It must be called just once and no MPI functions can be called after this.

4 Parallelization Strategy

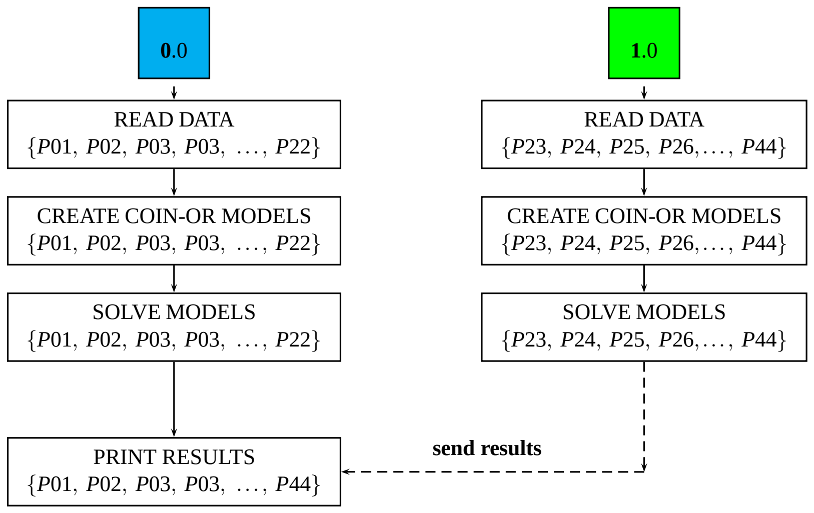
Before analysing the parallelization strategy, let us consider the serial version of the solving process. Figure 3 summarizes on a descriptive diagram the four tasks to be performed by the only thread. These tasks correspond to a data input, model creation, solving process and printing the numerical results. Each of them is applied to every single problem before starting the next step.



0.0
READ DATA
{P01, P02, P03, P03, …, P44}
CREATE COIN-OR MODELS
{P01, P02, P03, P03, …, P44}
SOLVE MODELS
{P01, P02, P03, P03, …, P44}
PRINT RESULTS
{P01, P02, P03, P03, …, P44}

Figure 3: Basic steps of the serial execution

We can extend the descriptive diagram to a two thread parallel version as shown in Figure 4. In this case the first three tasks are performed by both threads sharing the total amount of problems, this will significantly reduce the total amount of time. Once the solving process is finished on both threads, the coordinator thread gathers the numerical results by massing passing functions and prints them.



0.0
1.0
READ DATA
{P01, P02, P03, P03, …, P22}
READ DATA
{P23, P24, P25, P26,…, P44}
CREATE COIN-OR MODELS
{P01, P02, P03, P03, …, P22}
CREATE COIN-OR MODELS
{P23, P24, P25, P26,…, P44}
SOLVE MODELS
{P01, P02, P03, P03, …, P22}
SOLVE MODELS
{P23, P24, P25, P26,…, P44}
PRINT RESULTS
{P01, P02, P03, P03, …, P44}
send results

Figure 4: Basic steps of the MPI parallel execution

The user can define the branching cited on the third block according to the desired parallel programming strategy. The following procedure shows the parallel computing idea behind the example code provided in the Appendix 1 and a parallel diagram is illustrated in Figure 5.

Step 0: Declaring optimization and MPI variables. [All primary processors]

The correspondence at variable level between the optimization model (1) and appended C++ code is described at Table 1. Declare MPI variables by using functions of Group 1 described at Subsection 3.1.

Table 1: Correspondence between the optimization model and the implemented C++ code

Model	**q**	**z**	**(x,y)**	**nx+ny**
C++	nmodel	zq	x0	ncols

Step 1: Definition of the global environment. [All primary processors]

⋆ Beginning of the MPI environment using functions from Group 2, Subsection 3.2

Definition of the total number of processors, the rank of each of them and creation of a new communicator by using functions of Group 3 described at Subsection 3.3. By default, processor ranked *0* is defined as coordinator.

Step 2: Presolve assigned models. [All primary and auxiliary processors]

Every processor reads the corresponding MPS files and creates the associated CPLEX within COIN-OR optimization models.

- If selected optimizer = CPLEX: Each processor starts a shared memory parallel computing environment with a chosen number of auxiliary processors. In Figure 5 we show an illustrative diagram that corresponds to use 2 MPI threads and 4 CPLEX threads. Assigned models are presolved.
- If selected optimizer = COIN-OR: Each processor presolves assigned models.

Step 3: Global MPI communication. [All primary processors]

The coordinator processor gathers the presolve information of all models and all primary processors gather presolve status by using functions from Group 4 described at Subsection 3.4

Step 4: Check the presolve status [Coordinator processor]

- If all models are feasible and bounded: Go to Step 5.
- Else: Go to Step 8.

Step 5: Solve assigned models. [All primary and auxiliary processors]

- If selected optimizer = CPLEX: Each processor starts a shared memory parallel programming environment with a chosen number of auxiliary processors (see Figure 5). Assigned models are solved.
- If selected optimizer = COIN-OR: Each processor solves assigned models.

Step 6: Global MPI communication. [All primary processors]

The coordinator processor gathers the presolve information of all models by using functions form Group 4 described at Subsection 3.4

Step 7: Print results [Coordinator processor]

The coordinator processor prints all results.

⋆ End of the MPI environment using functions from Group 5 described at Subsection 3.5

Step 8: Execution ends in all processors. [All primary processors]

5 Basic installations and executions under UNIX-like systems

5.1 Basic installations: COIN-OR, CPLEX, C++, MPI

Firstly, its necessary to install COIN-OR, which stands for **CO**mputational **IN**fraestructure for **O**perations **R**esearch, a collection of open source software for optimization, see [2].

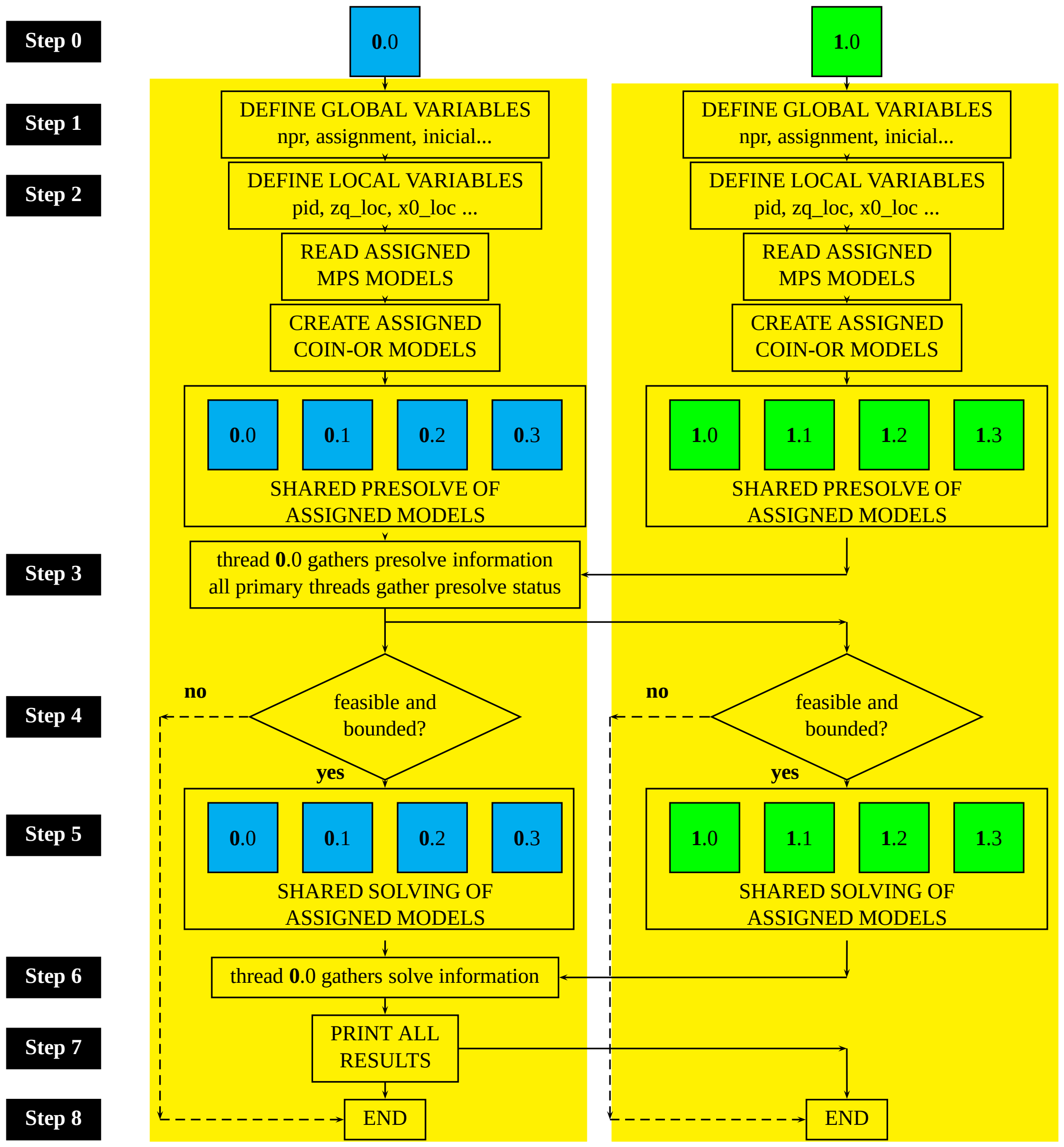

Step 0
Step 1
Step 2
Step 3
Step 4
Step 5
Step 6
Step 7
Step 8
0.0
1.0
DEFINE GLOBAL VARIABLES
npr, assignment, inicial...
DEFINE LOCAL VARIABLES
pid, zq_loc, x0_loc ...
READ ASSIGNED MPS MODELS
CREATE ASSIGNED COIN-OR MODELS
0.0
0.1
0.2
0.3
1.0
1.1
1.2
1.3
SHARED PRESOLVE OF ASSIGNED MODELS
thread 0.0 gathers presolve information
all primary threads gather presolve status
feasible and bounded?
no
yes
SHARED SOLVING OF ASSIGNED MODELS
thread 0.0 gathers solve information
PRINT ALL RESULTS
END

Figure 5: Parallelization diagram (2 MPI threads $\times$ 4 CPLEX threads)

The first step is to download the codes. For downloading *COIN-OR*, you must click on Download/Use in the left hand side of the home page http://www.coin-or.org. Then in the second Section titled Source Code, you must click on here to download the source code for the latest stable release. You can observe an index for the source of a number of COIN projects. You must click on CoinAll/ to obtain the list of last versions. In this case you will click and select CoinAll-1.3.1.zip. Alternatively you can go directly to the corresponding home page CoinAll-1.3.1.zip. After downloading the tarball you must extract the code into a new subdirectory, /software/CoinAll. For a step by step explanation on how to download and use COIN-OR consider [11].

Secondly, you must install the IBM ILOG CPLEX software for optimization with academic license, see [7]. For downloading CPLEX, you must click on

www-01.ibm.com/software/websphere/products/optimization/academic-initiative/index.html

You have to register "join now" in the right of screen and go to step 2. You will see the screen, IBM Academic Initiative (the ILOG Optimization Key Request) page, or click on

https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?lang=en_US&source=ai-ILOGKEY, give your institutional (for example, UPV/EHU) e-mail address. When you are registered click on academic-initiative you can do "Get Full version software". Select " download from the software catalogue", give your institutional e-mail address and password, click by email. In **Find by search text** put CPLEX and click on **Search** you have a list of programs, *IBM ILOG CPLEX Optimization Studio Academic Research Edition V12.2*, for several platforms. Select one of them and click on "I agree" at the end of the screen and **Download now**. Select the directory to install /software/C-PLEX. Also, save the license key file access.ilm in the location of your choice, for downloading the license key, you can follow the instructions of ILOGQuickStart.pdf, to obtain this file, please click on ILOGQuickStart.pdf. For a step by step explanation on how to download and use CPLEX within COIN-OR consider [11]. Finally, you can install the mpic++ compiler and the mpi.h library for MPI parallel computing from the webpage of GNU http://gcc.gnu.org/ (see [5]).

5.2 Basic executions

Once you have downloaded and installed all the software described in the previous subsection, you can compile, link and execute your own C++ code with the previous ones.

For the compilation in the machine `guinness.lgp.ehu.es` of the so-called `mainmpi.cpp` file provided in the Appendix 1 of this technical report, you will need a script that contains all the sentences to compile the C++ file under MPI, linking with the COIN-OR libraries and the optimization software CPLEX. We will use the following script so-called `compile`, which has been written by the technicians of the SGI/IZO-SGIker Computing Service at the UPV/EHU, see [1]. For an alternative unix machine, you can consult the Appendix 2.

```
#!/bin/bash
[ $# != 1 ] && echo -e "\n Provide one file name\n" && exit
#compiler=g++
#compiler=icpc
compiler=mpiicpc
extra_emt64=""
if [ $compiler == "g++" ]; then
  gnu_CXXFLAGS="-O3 -fomit-frame-pointer -pipe -fexceptions -DIL_STD -DNDEBUG"
  gnu_warnings="-pedantic-errors -Wimplicit -Wparentheses -Wreturn-type -Wcast-
      qual -Wall -Wpointer-arith -Wwrite-strings -Wconversion -Wno-unknown-
      pragmas"
else
  gnu_CXXFLAGS="-O3 -fomit-frame-pointer -pipe -fexceptions -DIL_STD -DNDEBUG"
  [ $(arch) == "x86_64" ] && extra_emt64="-Wparentheses"
  gnu_warnings="$extra_emt64 -Wreturn-type -Wcast-qual -Wall -Wpointer-arith -
      Wwrite-strings -Wconversion -Wno-unknown-pragmas"
fi
coin_dir="/software/CoinAll"
coin_libdir="$coin_dir/lib"
coin_liblink="-L$coin_libdir"
#all libraries of coin
coin_libs=$(for i in $(ls $coin_libdir/lib*.la);do basename $i .la;done |sed 's
    /^lib/-l/g' |tr -s "\n" " ")
coin_incs="-I$coin_dir/include/coin"
cplex_incs="-I/software/CPLEX/cplex/include/ilcplex/"
mkl_libs="-L/opt/intel/composerxe/mkl/lib/intel64/ -lmkl_intel_lp64 -
    lmkl_sequential -lmkl_core -lpthread"
```

```
cplex_libs="-L/software/CPLEX/cplex/lib/x86-64_sles10_4.1/static_pic/ -lcplex -
    lilocplex"
[ $(arch) == "ia64" ] && cplex_libs=""
[ -f $*_$(arch) ] && echo -e "\n ============= Deleting old $*_$(arch)\n" && rm
    $*_$(arch)
$compiler $gnu_CXXFLAGS $gnu_warnings $coin_incs $cplex_incs $coin_liblink
    $coin_libs $cplex_libs $mkl_libs $*.cpp -o $*_$(arch)
echo ""
if [ -f $*_$(arch) ] ;then
  echo ============= Done!!, $*_$(arch) binary has been created
else
  echo ============= ERROR!!,  $*_$(arch) binary could not be created
fi
echo ""
```

compile.dat

If you import this script from non UNIX-like systems, you should edit it

`$ vi compile`

and write

`Esc :  set fileformat=unix`

`Esc :  wq`

copy compile.dat in compile

`$ cp compile.dat compile`

Then, you can give execution permission with the following sentence:

`$ chmod u+x compile`

Now, you can execute the main C++ file with the script described previously:

`$ ./compile mainmpi`

The binary will be created with the `mainmpi_x86_64` name and it can be executed from the prompt with the command `mpirun` or `mpiexec` for mpi execution with 8 threads:

`$ mpirun -n 8 ./mainmpi_x86_64`

Or alternatively, the binary can be sent to the queue of the system using a `script_file` with the specific sentences to determine, among others,

- the maximum real time. For reserving 12 hours: `#PBS -l walltime=06:00:00`
- the memory allowed. For reserving 1 gb: `#PBS -l mem=40gb`
- the number of threads. For reserving 8 threads: `#PBS -l nodes=1:ppn=8:xeon8`

You can send the executable file to the queue with the sentence:

`$ qsub_arina script_file`

For more information about sending files to ARINA, you can see

http://www.ehu.es/sgi/category/mandar-trabajos.

6 Computational experience

The proposed main program has been implemented in a C++ experimental code. It uses the open source optimization engine *COIN-OR* for solving the mixed 0-1 optimization problems, in particular, we have used the functions: Clp (LP solver), Cbc (MIP solver), Osi, OsiClp, OsiCbc, OsiCpx and CoinUtils. Additionally, it uses one of the state-of-the-art commercial optimization engines, in particular CPLEX, see [7], within the open source engine *COIN-OR*.

The computational experiments were conducted at the ARINA computational cluster provided by the SGI/IZO-SGIker at the UPV/EHU. ARINA provides 1400 cores divided as follows: 1112 xeon cores, 248 Itanium2 cores and 40 opteron cores. All calculation nodes are connected by an Infiniband network with high bandwidth and low latency. For the present experiments the xeon x86_64 architecture (Xeon Nehalem-EP E5520 @ 2.27GHz) type nodes have been used, consisting on 8 cores with 24 Gb of RAM with an QDR infiniband interconnection. Whereas for the calculation data storage, a 22 Tb high performance file system based on Lustre was used.

The list of optimization problems that we are going to solve is based on the scenario-cluster submodels obtained from the multistage stochastic mixed 0-1 problem P4 taken from the computational experience reported in [4]. The full model is a large-scale problem with $|\Omega| = 217$ scenarios (nonsymmetric scenario tree) and $|\mathscr{T}| = 4$ stages, where the nonanticipativity constraints has been relaxed for the first and second stage, so $q = 44$ subproblems have been obtained. The main dimensions of the original large-scale optimization problem and the subproblems average and standard deviation dimensions are shown in Tables 2 and 3, respectively and Table 4 shows the value of the optimal objective function for the 44 mixed integer problems. The 44 MPS files can be downloaded as well as the `mainmpi.cpp` code.

Table 2: Large-scale problem dimensions

# rows	nx	ny	# non-zeros	density (%)	q	$\|\Omega\|$	$\mathscr{G}$	$\mathscr{T}$
9248	2176	4792	515768	0.64	44	217	272	4

Table 3: Testbed 44 subproblem average (standard deviation) dimensions

	# rows	nx	ny	# non-zeros	density (%)
average	270	63	151	14263	7.93
(st. deviation)	(53.46)	(12.58)	(25.16)	(2981.38)	(1.57)

Tables 5 and 6 give the main execution times for COIN-OR optimizer and for CPLEX solver within COIN-OR (allowing one single thread for CPLEX), respectively. The headings are as follows: *Available threads*, the number of threads for the serial and parallel programming; *Maximum # prob/thread*, the bottleneck or maximum number of problems to be solved by thread, that is, $\left\lceil \frac{q}{npr} \right\rceil$; *Min*, *Average*, *Max*, the minimum, average and maximum CPU time by thread (in seconds), respectively; *Average*, *St. dev.*, the average and standard deviation (in seconds) for the wall clock time after 15 executions of the same code. The CPLEX within COIN-OR solver is between 3 and 5 five times

Table 4: Testbed 44 MIP solutions

P1	-2953.46	P12	-3215.94	P23	-8945.76	P34	-2915.80
P2	-5067.97	P13	-6662.71	P24	-7271.52	P35	-4291.44
P3	-3957.70	P14	-4078.15	P25	-7136.25	P36	-9412.94
P4	-5310.86	P15	-5315.24	P26	-7046.85	P37	-7899.80
P5	-9219.36	P16	-8504.21	P27	-5561.61	P38	-4851.96
P6	-6539.62	P17	-6592.45	P28	-5327.66	P39	-7735.64
P7	-8196.80	P18	-5186.78	P29	-3991.81	P40	-7764.66
P8	-6582.53	P19	-6291.11	P30	-8051.70	P41	-9828.10
P9	-7811.68	P20	-7378.17	P31	-4887.34	P42	-2786.57
P10	-8126.43	P21	-8450.21	P32	-5434.41	P43	-4311.35
P11	-8420.17	P22	-8558.38	P33	-9432.48	P44	-8135.21

faster than the plain use of COIN-OR (without any presolving or cut generation). If we compare CPU time versus real time, we can conclude that communication time is not very relevant.

Table 5: Computing times under COIN-OR

	Available	**Maximum**	**CPU time**			**Real time**	
	threads	**# prob/thread**	**Min.**	**Average**	**Max.**	**Average**	**St. dev.**
Serial	**1**	44	17.651	17.651	17.651	17.780	0.013
MPI	**1**	44	18.013	18.013	18.013	18.239	0.093
	2	22	8.823	8.866	8.779	8.972	0.010
	4	11	5.018	5.165	5.271	5.373	0.029
	8	6	3.118	3.647	3.960	4.508	0.253
	16	3	1.735	2.077	2.272	2.632	0.270
	32	2	1.239	1.857	1.996	2.054	0.005
	64	1	0.839	1.403	1.517	1.590	0.260

Table 6: Computing times under COIN-OR&CPLEX - single thread for CPLEX

	Available	**Maximum**	**CPU time**			**Real time**	
	threads	**# prob/thread**	**Min.**	**Average**	**Max.**	**Average**	**St. dev.**
Serial	**1**	44	6.107	6.107	6.107	6.163	0.019
MPI	**1**	44	6.344	6.344	6.344	6.401	0.005
	2	22	3.161	3.165	3.168	3.201	0.002
	4	11	1.647	1.667	1.685	1.706	0.005
	8	6	0.992	1.025	1.074	1.120	0.030
	16	3	0.521	0.546	0.564	0.604	0.058
	32	2	0.349	0.390	0.405	0.426	0.0175
	64	1	0.227	0.259	0.276	0.297	0.0857

Table 7 gives the main execution times for CPLEX solver within COIN-OR considering a number of available threads and several combinations between the number of threads for distributed memory (MPI threads) and shared memory (CPLEX threads). The rest of headings are as defined in the previous tables. For 8 available threads the average real time for computing the serial execution is 7.699 seconds, which corresponds to the worst strategy. The Figure 6 shows the real time for COIN-OR in horizontal read line and the real time for CPLEX in vertical blue bars, it is classified according to the available number of threads, as detailed in Tables 5 and 7. We can observe that for a fixed number of available threads the real time for solving the q problems (a big number of them with small dimensions) is decreasing with the bigger number of MPI threads. So, the fastest case corresponds to the biggest number of MPI threads and a single thread for CPLEX, which highlights the interest of MPI parallel computing.

Table 7: Computing times under COIN-OR&CPLEX - multiple threads for CPLEX

Available threads	**MPI $\times$ CPLEX threads**	**CPU time**			**Real time**	
		Min.	**Average**	**Max.**	**Average**	**St. dev.**
1	1×1	6.107	6.107	6.107	6.163	0.019
2	1×2	4.328	4.328	4.328	5.683	0.046
2	2×1	3.161	3.165	3.168	3.201	0.002
4	1×4	2.382	2.382	2.382	6.076	0.055
4	2×2	2.141	2.184	2.227	2.986	0.022
4	4×1	1.647	1.667	1.685	1.706	0.005
8	1×8	2.590	2.590	2.590	7.699	0.222
8	2×4	1.184	1.229	1.258	3.229	0.245
8	4×2	1.095	1.124	1.176	1.632	0.032
8	8×1	0.992	1.025	1.074	1.120	0.030

MPI$=1$ corresponds to serial execution

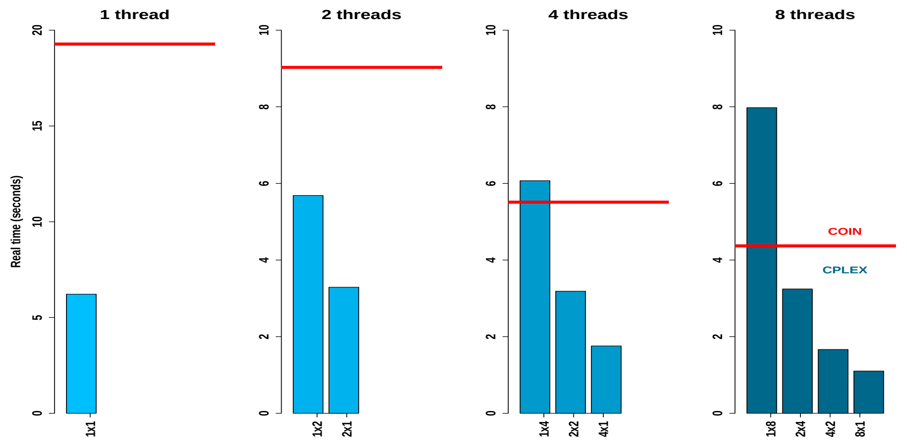


Figure 6: CPLEX within COIN-OR real time, classification for (MPI $\times$ CPLEX) threads

Finally, the Table 8 summarizes the best real times for COIN-OR and CPLEX, with all the available threads considered. The headings are as follows: *Real Time*, the wall clock time in seconds (average time over 15 realizations); *Speedup*, serial real time over parallel real time; and *Efficiency*, parallel real time over number of threads, in percentage. The speedup and efficiency is high and similar for 2 and 4 available threads, but is better for CPLEX within COIN-OR optimization solver for 8 or more threads. These results are illustrated in Figure 7.

Table 8: Efficiency COIN-OR versus COIN-OR&CPLEX

	Available threads	**COIN-OR**			**COIN-OR&CPLEX**		
		Real time	**Speedup**	**Efficiency**	**Real time**	**Speedup**	**Efficiency**
Serial	**1**	17.780	1.000	100.0	6.163	1.000	100.0
MPI	**2**	8.972	1.982	99.1	3.201	1.925	96.3
	4	5.373	3.309	82.7	1.706	3.613	90.3
	8	4.508	3.944	49.3	1.120	5.503	68.8
	16	2.632	6.755	42.2	0.604	10.204	63.8
	32	2.054	8.656	27.1	0.426	14.670	45.8
	64	1.590	11.182	17.5	0.297	20.751	32.4

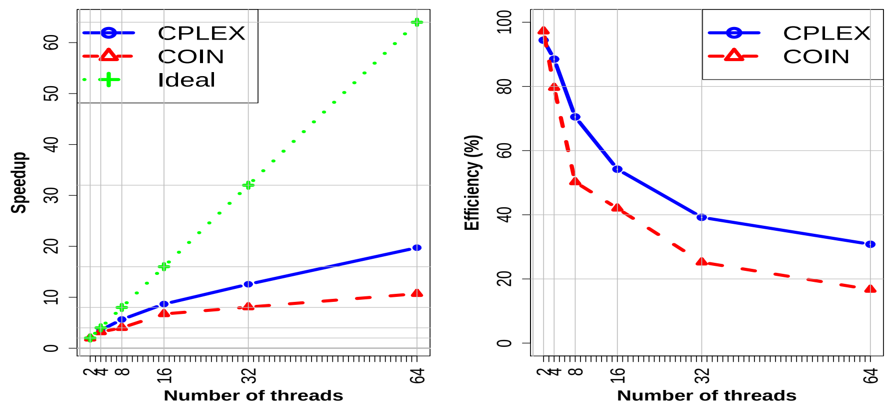


Figure 7: Speedup and efficiency vs number of threads

Appendix 1

The main program `mainmpi.cpp` is detailed below (C++ keywords are shown in blue, comments in green and MPI elements in red).

```
// This is the last version of the code mainmpi.cpp. All rights reserved.
// Copyright (C) 2012 by U. Aldasoro, M.A. Garin, M. Merino and G. Perez,
// from UPV/EHU.
// No part of this code may be reproduced, modified or transmitted, in any
// form or by any means without the prior written permission of the authors.

// Cluster models are in mps format, as external files

#include "itoa.h"
#include <mpi.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
using namespace std;
#include "ClpSimplex.hpp"
#include "CoinHelperFunctions.hpp"
#include "CoinBuild.hpp"
#include "CoinModel.hpp"
#include <iomanip>
#include <cassert>
#include "OsiClpSolverInterface.hpp"
#include "CbcModel.hpp"

#define rmaxmin (1)    // (1): Maximum; (-1): Minimum
#define nmodel 44    // Number of cluster models
#define useCplex 1     // Select solver. 0: COIN-OR; 1: Cplex
#define threadsCplex 2  // Number of parallel threads used by each MPI thread

#ifdef useCplex
  #include "OsiCpxSolverInterface.hpp"
#endif

int main(int argc, char **argv)
{
  // STEP 0 - DECLARING OPTIMIZATION AND MPI VARIABLES
  int imod,i,j,loc,pid,npr,lp,original_rank,original_size,error,ncols;
  int assignment[nmodel],inicial[nmodel],nonzero[nmodel],numres[nmodel],
    numvar[nmodel],numvarint[nmodel],assignmentX0[nmodel],iniX0[nmodel],
    istruef969[nmodel];
  double zq[nmodel];

  MPI_Group  orig_group,new_group;      //MPI groups
  MPI_Comm   new_comm;              //MPI communicator


  // STEP 1 - DEFINITION OF THE GLOBAL ENVIRONMENT
  MPI_Init(&argc,&argv); // Beginning of the MPI environment
    MPI_Comm_size(MPI_COMM_WORLD, &original_size); // Total number of threads
    MPI_Comm_rank(MPI_COMM_WORLD, &original_rank); // Who am I?

    // 1.1 - Creating a new communicator considering active threads

    // 1.1.1- Basic variables
    int ranks1[nmodel];
    int *ranks2;  ranks2=new int[original_size];
    for(i=0;i<nmodel;i++) {ranks1[i]=i;}
    for(i=0;i<original_size;i++) {ranks2[i]=i;}

    // 1.1.2- Extract handle of global group from MPI_COMM_WORLD using
        MPI_Comm_group
    MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

    // 1.1.3- Form new group as a subset of global group using MPI_Group_incl
    if (nmodel < original_size){MPI_Group_incl(orig_group, nmodel, ranks1, &
        new_group);
    } else {MPI_Group_incl(orig_group, original_size, ranks2, &new_group);}

    // 1.1.4- Create new communicator for new group using MPI_Comm_create
    MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);

    // 1.1.5- Determine new rank in new communicator using MPI_Comm_rank
    MPI_Group_rank (new_group, &pid);
    MPI_Group_size (new_group, &npr);

    // 1.1.6 - Creating the output file
    ofstream solution("CPLEX_COIN_MPI_solution.dat",ios::out); //Output file

    // 1.2 - Only active threads follow the solving process

    if (original_rank < npr) {

      // 1.2.1 - Assigning the work load
      for(i=0;i<npr;i++) assignment[i]=int(nmodel/npr);
      for(i=0;i<(nmodel-npr*int(nmodel/npr));i++) assignment[i]=assignment[i]+1;

      inicial[0]=0;
      for(i=1;i<npr;i++) inicial[i]=inicial[i-1]+assignment[i-1];

      // 1.2.2 - Declaring local variables
      int ncols_max_loc=0;
      double *zq_loc;  zq_loc=new double[assignment[pid]];
      int *nonzero_loc;  nonzero_loc=new int[assignment[pid]];
      int *numres_loc;  numres_loc=new int[assignment[pid]];
      int *numvar_loc;  numvar_loc=new int[assignment[pid]];
      int *numvarint_loc;  numvarint_loc=new int[assignment[pid]];
      int *istruef969_loc;  istruef969_loc=new int[assignment[pid]];

      // 1.2.3 - Creating the output file
      if (pid == 0) {
        if (useCplex == 0) {solution<<"*** SOLVER: COIN-OR ***"<<"\n\n";}
        else {solution<<"*** SOLVER: CPLEX *** \n\n  +++ NUMBER OF " <<
          "PARALLEL THREADS ON CPLEX = "<<threadsCplex<<" +++ \n\n";}

        solution<<"Beginning of CPLEX_COIN_MPI_solution.dat (output of mainmpi.
            cpp) \n  Number of threads = "<<original_size<<"\n Number of
            submodels = "<<nmodel<<"\n Number of active threads = "<<npr<<"\n";
        for(i=0;i<npr;i++) {
          solution<<" Number of submodels solved by thread "<<i<<" = "<<
```

```
            assignment[i]<<", starting by submodel = "<< inicial[i]+1<<"\n";}
    }

    // 1.2.4 - Creating the COIN-OR or CPLEX/COIN-OR models
    OsiClpSolverInterface *sol0;
    sol0=new OsiClpSolverInterface[assignment[pid]];

    OsiCpxSolverInterface *sol1;
    sol1=new OsiCpxSolverInterface[assignment[pid]];
    for(loc=0;loc<assignment[pid];loc++){
      CPXENVptr env = sol1[loc].getEnvironmentPtr();
      CPXsetintparam(env, CPX_PARAM_THREADS, threadsCplex);
    }

    for(loc=0;loc<assignment[pid];loc++) {

      imod=inicial[pid]+loc;       const char* final;
      final="";              char buffer[33];
      final=itoa(imod+1, buffer,10); //convert in char
      char model[80];          strcpy (model,"Cluster");
      strcat (model,final);      puts (model);

      if (useCplex == 0) {
        sol0[loc].setObjSense(rmaxmin);
        sol0[loc].readMps(model); //Read cluster submodel
        nonzero_loc[loc] = sol0[loc].getNumElements();
        numres_loc[loc] = sol0[loc].getNumRows();
        numvar_loc[loc] = sol0[loc].getNumCols();
        for(j=0;j<sol0[loc].getNumCols();j++)
          if(sol0[loc].isInteger(j)) numvarint_loc[loc]++;
        if(sol0[loc].getNumCols() > ncols_max_loc)
          {ncols_max_loc = sol0[loc].getNumCols();}
      } else {
        sol1[loc].setObjSense(rmaxmin);
        sol1[loc].readMps(model); //Read cluster submodel
        nonzero_loc[loc] = sol1[loc].getNumElements();
        numres_loc[loc] = sol1[loc].getNumRows();
        numvar_loc[loc] = sol1[loc].getNumCols();
        for(j=0;j<sol1[loc].getNumCols();j++)
          if(sol1[loc].isInteger(j)) numvarint_loc[loc]++;
        if(sol1[loc].getNumCols() > ncols_max_loc)
          {ncols_max_loc = sol1[loc].getNumCols();}
      }
    }

    if (npr > 1) {
      MPI_Allreduce(&ncols_max_loc,&ncols , 1, MPI_INT, MPI_MAX,new_comm);
    } else {
      ncols=ncols_max_loc;}

    double **x0;x0=new double*[nmodel];
          for(i=0;i<nmodel;i++) x0[i]=new double [ncols];

    double *x0_loc;   x0_loc=new double[assignment[pid]*ncols];
```

```
      double *x0vector;  x0vector=new double[nmodel*ncols];

      for(imod=0;imod<assignment[pid];imod++) {
        for(j=0;j<nmodel;j++) {
            x0_loc[imod*ncols+j]=0;
        }
      }

      for(i=0;i<npr;i++) {
        iniX0[i]=inicial[i]*ncols; assignmentX0[i]=assignment[i]*ncols;}

      // STEP 2 - PRESOLVE ASSIGNED MODELS
      error=0;

      for(loc=0;loc<assignment[pid];loc++)
      {
        if (useCplex == 0) {
          imod=inicial[pid]+loc;
          CbcModel pm0(sol0[loc]);
          istruef969_loc[loc]=0;      sol0[loc].initialSolve();

          if(sol0[loc].isProvenPrimalInfeasible()) {zq_loc[loc]=rmaxmin*1.e25;
          istruef969_loc[loc]=1;}

          if(istruef969_loc[loc]==0){
            if(sol0[loc].isProvenDualInfeasible() ){
              zq_loc[loc]=-1.0*rmaxmin*1.e25; istruef969_loc[loc]=2;}

            if(!sol0[loc].isProvenOptimal()) {
              zq_loc[loc]=-1.0*rmaxmin*1.e25; istruef969_loc[loc]=2;}
          }
        } else {
          imod=inicial[pid]+loc;
          istruef969_loc[loc]=0;      sol1[loc].initialSolve();

          if(sol1[loc].isProvenPrimalInfeasible()) {zq_loc[loc]=rmaxmin*1.e25;
          istruef969_loc[loc]=1;}

          if(istruef969_loc[loc]==0){
            if(sol1[loc].isProvenDualInfeasible() ){
              zq_loc[loc]=-1.0*rmaxmin*1.e25; istruef969_loc[loc]=2;}

            if(!sol1[loc].isProvenOptimal()) {
              zq_loc[loc]=-1.0*rmaxmin*1.e25; istruef969_loc[loc]=2;}
          }
        }
      }

      // STEP 3 - GLOBAL MPI COMMUNICATION (GATHERING PRESOLVE INFORMATION)
      if (npr > 1) {
        MPI_Allgatherv(&istruef969_loc[0], assignment[pid], MPI_INT,
          &istruef969[0],assignment, inicial, MPI_INT, new_comm);
        MPI_Gatherv(&zq_loc[0], assignment[pid], MPI_INT, &zq[0],
          assignment, inicial, MPI_INT, 0, new_comm);
      } else {
        for(imod=0;imod<nmodel;imod++) {
          istruef969[imod]=istruef969_loc[imod];     zq[imod]=zq_loc[imod];
        }
      }

      for(imod=0;imod<nmodel;imod++) {
        if(istruef969[imod] == 1){
          solution<<"\n Cluster submodel "<<imod+1
            <<" is primal infeasible. ObjValue = "<<zq[imod]; error=1;}
        if(istruef969[imod] == 2){
          solution<<"\n Cluster submodel "<<imod+1
            <<" is unbounded. ObjValue = "<<zq[imod]; error=1;}
      }

      // STEP 4 – CHECK PRESOLVE STATUS
      if(error == 0){

        // STEP 5– SOLVE ASSIGNED MODELS
        if (useCplex == 0) {
          for(loc=0;loc<assignment[pid];loc++)
          {
            imod=inicial[pid]+loc;     CbcModel pm0(sol1[loc]);
            pm0.branchAndBound();

            if(!pm0.isProvenOptimal()) {
              zq_loc[loc]=-1.0*rmaxmin*1.e25;     istruef969_loc[loc]=3;}

            if(fabs(pm0.getBestPossibleObjValue())>1.e24){
              zq_loc[loc]=-1.0*rmaxmin*1.e25;     istruef969_loc[loc]=4;}

            if(istruef969_loc[loc] == 0) {
              zq_loc[loc]=pm0.getObjValue();
              for(j=0;j<pm0.getNumCols();j++) {
                x0_loc[loc*ncols+j]=pm0.getColSolution()[j];}}
          }
        } else {
          for(loc=0;loc<assignment[pid];loc++)
          {
            imod=inicial[pid]+loc;     sol1[loc].branchAndBound();

            if(!sol1[loc].isProvenOptimal()) {
              zq_loc[loc]=-1.0*rmaxmin*1.e25;  istruef969_loc[loc]=3;}

            if(fabs(sol1[loc].getObjValue())>1.e24){
              zq_loc[loc]=-1.0*rmaxmin*1.e25;  istruef969_loc[loc]=4;}

            if(istruef969_loc[loc] == 0) {
              zq_loc[loc]=sol1[loc].getObjValue();
              for(j=0;j<sol1[loc].getNumCols();j++)
                x0_loc[loc*ncols+j]=sol1[loc].getColSolution()[j];}
          }
        }
```

```
    // STEP 6- GLOBAL MPI COMMUNICATION (GATHERING SOLVE INFORMATION)
    if (npr > 1) {
      MPI_Gatherv(&zq_loc[0],assignment[pid],MPI_DOUBLE,&zq[0],
        assignment,inicial,MPI_DOUBLE,0,new_comm);
      MPI_Gatherv(&nonzero_loc[0],assignment[pid],MPI_INT,&nonzero[0],
        assignment,inicial,MPI_INT,0,new_comm);
      MPI_Gatherv(&numvar_loc[0],assignment[pid],MPI_INT,&numvar[0],
        assignment,inicial,MPI_INT,0,new_comm);
      MPI_Gatherv(&numvarint_loc[0],assignment[pid],MPI_INT,&numvarint[0],
        assignment,inicial,MPI_INT,0,new_comm);
      MPI_Gatherv(&numres_loc[0],assignment[pid],MPI_INT,&numres[0],
        assignment,inicial,MPI_INT,0,new_comm);
      MPI_Gatherv(x0_loc,assignmentX0[pid],MPI_DOUBLE,x0vector,
        assignmentX0,iniX0,MPI_DOUBLE,0,new_comm);
      MPI_Gatherv(&istruef969_loc[0],assignment[pid],MPI_INT,&istruef969[0],
        assignment,inicial,MPI_INT,0,new_comm);
    } else {
      for(imod=0;imod<nmodel;imod++) {
        zq[imod]=zq_loc[imod];          nonzero[imod]=nonzero_loc[imod];
        numvar[imod]=numvar_loc[imod];       numres[imod]=numres_loc[imod];
        istruef969[imod]=istruef969_loc[imod];
        for(j=0;j<numvar[imod];j++){
          x0vector[imod*ncols+j]=x0_loc[imod*ncols+j];
        }
      }
    }

    for(imod=0;imod<nmodel;imod++) {
      if((istruef969[imod] == 3) | (istruef969[imod] == 4)){
        solution<<"\n Cluster submodel "<<imod+1 <<" is unbounded ";}
    }
  }

  // STEP 7 - PRINT RESULTS (if pid == 0)
  if (pid == 0) {

    solution<<"\n\n [RESULTS 1:] PROPERTIES OF SOLVED MODELS ";
    for(imod=0;imod<nmodel;imod++){
      solution<<"\n \n Cluster submodel "<<imod+1<<" of "<<nmodel;
      solution<< "\n Number of variables: "; solution<< numvar[imod];
      solution<< "\n Number of constraints: "; solution<< numres[imod];
      solution<< "\n Number of nonzero elements: "<<nonzero[imod];
      for(j=0;j<numvar[imod];j++){
        x0[imod][j]=x0vector[imod*ncols+j];
        //solution<< "\n x[ "<<imod<<","<<j<<"]"<<x0[imod][j]<<" \n"; // To
            print x0
      }
    }

    solution<<"\n\n\n [RESULTS 2:] OBJECTIVE FUNCTION VALUES \n";

    for(imod=0;imod<nmodel;imod++) {
      solution<<"\n Optimal objective value Cluster submodel "<<imod+1
        <<" of "<<nmodel <<" is "<<zq[imod];}
      }

    }

  // STEP 8 - EXECUTION ENDS IN ALL PROCESSORS
  solution.close();
  MPI_Finalize(); //End of the MPI environment
  return 0;
}
```

mainmpi.cpp

Appendix 2

To compile and link the code, you can also use a Makefile similar to the described below in this Appendix, where are used additionally the libraries -lpthread. As summary to generate and run the executable you must to install the packages,

lam-runtime	libgomp1
liblam4	libmpich1.0gf
libopenmpi-dev	libopenmpi1.3
mpi-default-bin	mpi-default-dev
mpich-bin	openmpi-bin
openmpi-checkpoint	openmpi-common
openmpi-doc	

and their dependences. All of them have been installed in the machine `b012526.bs.ehu.es` of the Laboratory of Quantitative Economics from the University of the Basque Country (UPV/EHU, Bilbao, Spain). To run the executable of name `mainmpi` in this machine with eight cores, you must type from the prompt,

```
$ mpirun -v -np 8 -host b012526 mainmpi
```

```
# Copyright (C) 2006 International Business Machines and others.
# All Rights Reserved.
# This file is distributed under the Common Public License.
# $Id: Makefile.in 726 2006-04-17 04:16:00Z andreasw $
##########################################################################
#    You can modify this example makefile to fit for your own program.   #
#    Usually, you only need to change the five CHANGEME entries below.   #
##########################################################################
# To compile other examples, either changed the following line, or
# add the argument DRIVER=problem_name to make
DRIVER = mainmpi

# CHANGEME: This should be the name of your executable
EXE = $(DRIVER)

# CHANGEME: Here is the name of all object files corresponding to the source
#           code that you wrote in order to define the problem statement
OBJS= mainmpi.o
```

```
SYSTEM = x86-64_sles10_4.1
# Directory with COIN header files
COININCDIR = /opt/coin-1.3.1/include/coin
# Directory with COIN libraries
COINLIBDIR = /opt/coin-1.3.1/lib
COINLIBDIROSI = /opt/coin-1.3.1/Osi/lib
# Directory with CPLEX
CPLEXDIR= /opt/ILOG/CPLEX_Studio_AcademicResearch122/cplex
# Directory with CPLEX header files
CPLEXINCDIR = $(CPLEXDIR)/include/ilcplex
# Directory with CPLEX libraries
CPLEXLIBDIR =$(CPLEXDIR)/lib/$(SYSTEM)/static_pic/libcplex.a

CPLEXBINDIR =$(CPLEXDIR)/bin/$(SYSTEM)

# CHANGEME: Additional libraries
ADDLIBS =

# CHANGEME: Additional flags for compilation (e.g., include flags)
ADDINCFLAGS =
##########################################################################
#  Usually, you don't have to change anything below. Note that if you   #
#  change certain compiler options, you might have to recompile the     #
#  package.                                                             #
##########################################################################
# C++ Compiler command
#CXX = g++
CXX=mpic++

# C++ Compiler options for g++
#CXXFLAGS = -O3 -fomit-frame-pointer -pipe -DNDEBUG -pedantic-errors -Wimplicit
    #-Wparentheses -Wreturn-type -Wcast-qual -Wall -Wpointer-arith -Wwrite-
    strings
#-Wconversion -Wno-unknown-pragmas -O -fPIC -fexceptions  -DIL_STD

# C++ Compiler options for mpic++
CXXFLAGS = -O3 -fomit-frame-pointer -pipe -DNDEBUG  -Wimplicit -Wparentheses -
    Wreturn-type -Wcast-qual -Wall -Wpointer-arith -Wwrite-strings -Wconversion
    -Wno-unknown-pragmas -O -fPIC -fexceptions  -DIL_STD

# additional C++ Compiler options for linking
CXXLINKFLAGS =  -Wl,--rpath -Wl,$(COINLIBDIR)  -Wl,$(CPLEXLIBDIR)

# Libraries necessary to link with Clp
LIBS = -L$(COINLIBDIR) -lCbcSolver -lCbc -lCgl -lOsiClp -lOsiCbc -lOsi -lClp -
    lCoinUtils -L$(COINLIBDIROSI) -lOsiCpx -L$(CPLEXBINDIR) -lcplex122 -lpthread
     \
  -lm

# Necessary Include dirs (we use the CYGPATH_W variables to allow
# compilation with Windows compilers)
INCL =  -I`$(CYGPATH_W) $(COININCDIR) $(CPLEXINCDIR)` $(ADDINCFLAGS)

# The following is necessary under cygwin, if native compilers are used
```

```
CYGPATH_W = echo

# Here we list all possible generated objects or executables to delete them
CLEANFILES =

all: $(EXE)

.SUFFIXES: .cpp .c .o .obj

$(EXE): $(OBJS)
	bla=;\
	for file in $(OBJS); do bla="$$bla `$(CYGPATH_W) $$file`"; done; \
	$(CXX) $(CXXLINKFLAGS) $(CXXFLAGS) -o $@ $$bla $(ADDLIBS) $(LIBS)
clean:
	rm -rf $(CLEANFILES) $(OBJS)
.cpp.o:
	$(CXX) $(CXXFLAGS) $(INCL) -c -o $@ `test -f '$<' || echo '$(SRCDIR)/'`$<
.cpp.obj:
	$(CXX) $(CXXFLAGS) $(INCL) -c -o $@ `if test -f '$<'; then $(CYGPATH_W) '$<';
	    else $(CYGPATH_W) '$(SRCDIR)/$<'; fi`
.c.o:
	$(CC) $(CFLAGS) $(INCL) -c -o $@ `test -f '$<' || echo '$(SRCDIR)/'`$<
.c.obj:
	$(CC) $(CFLAGS) $(INCL) -c -o $@ `if test -f '$<'; then $(CYGPATH_W) '$<';
	    else $(CYGPATH_W) '$(SRCDIR)/$<'; fi`
```

Makefile

7 Acknowledgements

This research has been partially supported by the projects ECO2008-00777 ECON from the Ministry of Education and Science, Grupo de Investigación IT-347-10 from the Basque Government, Spain. Technical and human support provided by Dr. E. Ogando and Dr. T. Mercero from IZO-SGI, SGIker (UPV/EHU, MICINN,GV/EJ, ERDF and ESF) is gratefully acknowledged. We would like to express our gratefulness to Prof. Dr. F. Tusell for making easier the access to the Laboratory of Quantitative Economics from the University of the Basque Country (UPV/EHU, Bilbao, Spain) to perform and check some of our computational experiments.

References

[1] ARINA cluster. IZO-SGI, SGIker (UPV/EHU). Website. `http://www.ehu.es/sgi/recursos/cluster-arina-2`.

[2] COIN-OR. COmputational INfrastructure for Operations Research. Website. `http://www.coin-or.org/`.

[3] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[4] Laureano F. Escudero, María Araceli Garín, María Merino, and Gloria Pérez. An algorithmic framework for solving large-scale multistage stochastic mixed 0-1 problems with nonsymmetric scenario trees. *Comput. Oper. Res.*, 39:1133–1144, May 2012.

[5] GCC. GNU compiler collection. Website. `http://gcc.gnu.org/`.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[7] IBM. ILOG CPLEX optimizer. Website `.http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

[8] J T Linderoth. *Topics in Parallel Integer Optimization*. PhD thesis, Georgia Intitute of Technology, 1998.

[9] R Lougee-Heimer. The common optimization Interface for operations research: Promoting open-source software in the operations research community. *IBM JOURNAL OF RESEARCH AND DEVELOPMENT*, 47(1):57–66, JAN 2003.

[10] Peter S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[11] Gloria Pérez Sainz de Rozas and María Araceli Garín Martín. On Downloading and Using CPLEX within COIN-OR for Solving Linear/Integer Optimization Problems. Biltoki, Universidad del País Vasco - Departamento de Economía Aplicada III (Econometría y Estadística), 2011.

[12] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.