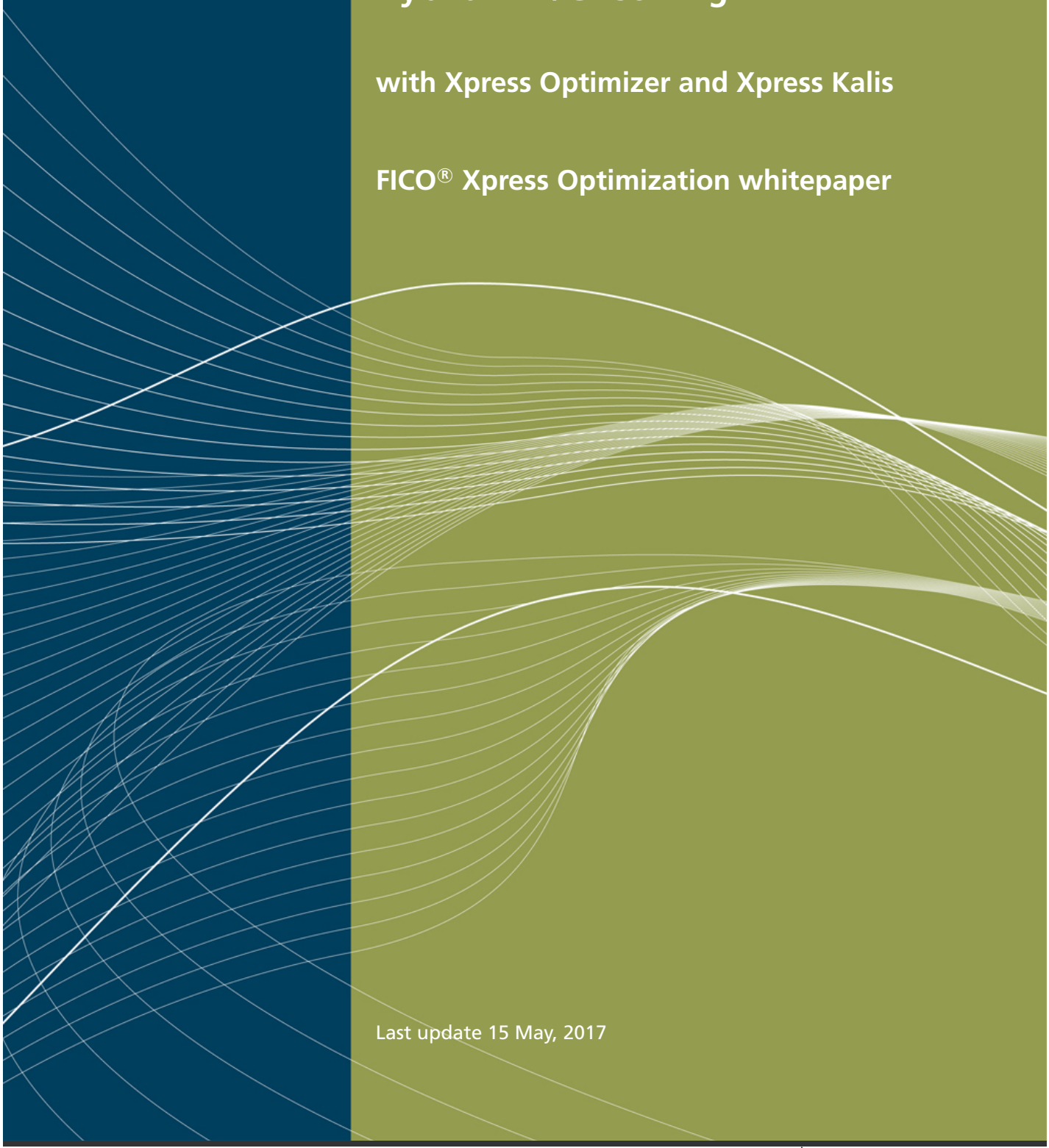
Whitepaper

FICO® Xpress Optimization

Hybrid MIP/CP solving

with Xpress Optimizer and Xpress Kalis

FICO® Xpress Optimization whitepaper



Last update 15 May, 2017

Hybrid MIP/CP solving

with Xpress Optimizer and Xpress Kalis

S. Heipcke

Xpress Team, FICO, FICO House, Starley Way, Birmingham B37 7GN, UK
http://www.fico.com/xpress

15 May, 2017

Abstract

This paper describes several examples of combining Mathematical Programming (LP and MIP) solution techniques with Constraint Programming.
For the implementation we use Xpress Optimizer and Artelys-Kalis from the Mosel language (Mosel modules *mmxprs* and *kalis*)
In the first example CP propagation is used as preprocessor for LP solving; in the second example CP solving is employed as a cut generation heuristic for a MIP branch-and-cut algorithm.

Contents

1 Introduction

The representation of real-world conditions with more and more detail sometimes render optimization applications difficult to represent and solve with a given solution technique. In such

a case it may become necessary to develop some type of a decomposition approach, possibly treating parts of a problem with different solution methods and tools for modeling and solving them.

In the modeling and solving environment Xpress Mosel several (solver) modules may be used jointly to implement and solve such difficult problems. In particular, it is possible to implement solution algorithms combining Constraint Programming (CP) with Linear or Mixed Integer Programming by using Xpress Kalis with the *mmxprs* module that provides access to Xpress Optimizer.

This paper discusses two schemes of combining CP with LP/MIP for problem solving:

- CP and MIP solving may be used *sequentially*, for instance, employing CP constraint propagation as a *preprocessing* routine for LP/MIP problems, as shown in the project planning example in Section 2.
- An example of *parallel* MIP-CP problem solving is given in Section 3 where CP solving is used as *cut generation routine* during the MIP branch-and-bound search in a scheduling with machine assignment problem.

The examples described in this paper require Xpress Kalis to be installed and licensed in addition to a standard installation of FICO Xpress Optimization (Optimizer and Mosel).

Combining different solution methods for solving problems requires some knowledge of all involved solvers and techniques. This paper assumes that the reader is familiar with both, Mathematical Programming and Constraint Programming, and also has a certain experience with using the involved solvers from the Mosel language.

For examples of problem solving with Xpress Optimizer the reader is referred to the 'Mosel User Guide', in particular Chapter *'More about Integer Programming'*. All functionality of the Mosel modules *mmxprs* and *mmjobs* is documented in the 'Mosel Language Reference Manual'. The 'Xpress Optimizer Reference Manual' is the complete reference for the solver. The Xpress Whitepaper 'Mosel: multiple models and parallel solving' discusses examples of using the module *mmjobs*. The documentation of Xpress Kalis is available in the 'Xpress Kalis Reference Manual'. A detailed introduction to working with this software is given in the 'Xpress Kalis User Guide'.

2 Using CP propagation as preprocessor

When the constraints in a CP model are posted to the solver, they often immediately trigger some reductions to the domains of the involved variables. In certain (easy) cases, it may even happen that the constraint propagation is sufficient to obtain a solution without having to start an enumeration.

The domain reductions obtained through the constraint propagation can be passed on to an LP or MIP model, thus replacing, or re-inforcing the preprocessing algorithms that are used by LP and MIP solvers.

The example description in the following sections is taken from Section 7.1 of the book 'Applications of optimization with Xpress-MP'.

2.1 Project scheduling example

A town council wishes to construct a small stadium in order to improve the services provided to the people living in the district. After the invitation to tender, a local construction company is awarded the contract and wishes to complete the task within the shortest possible time. All the

major tasks are listed in the following table. The durations are expressed in weeks. Some tasks can only start after the completion of certain other tasks. The last two columns of the table refer to question 2 which we shall see later.

Table 1: Data for stadium construction

Task	Description	Duration	Predecessors	Max. reduct.	Add. cost per week (in 1000)
1	Installing the construction site	2	none	0	–
2	Terracing	16	1	3	30
3	Constructing the foundations	9	2	1	26
4	Access roads and other networks	8	2	2	12
5	Erecting the basement	10	3	2	17
6	Main floor	6	4,5	1	15
7	Dividing up the changing rooms	2	4	1	8
8	Electrifying the terraces	2	6	0	–
9	Constructing the roof	9	4,6	2	42
10	Lighting of the stadium	5	4	1	21
11	Installing the terraces	3	6	1	18
12	Sealing the roof	2	9	0	–
13	Finishing the changing rooms	1	7	0	–
14	Constructing the ticket office	7	2	2	22
15	Secondary access roads	4	4,14	2	12
16	Means of signaling	3	8,11,14	1	6
17	Lawn and sport accessories	9	12	3	16
18	Handing over the building	1	17	0	–

Question 1: Which is the earliest possible date of completing the construction?

Question 2: The town council would like the project to terminate earlier than the time announced by the builder (answer to question 1). To obtain this, the council is prepared to pay a bonus of € 30K for every week the work finishes early. The builder needs to employ additional workers and rent more equipment to cut down on the total time. In the preceding table he has summarized the maximum number of weeks he can save per task (column 'Max. reduct.') and the associated additional cost per week. When will the project be completed if the builder wishes to maximize his profit?

2.2 Model formulation for question 1

The representation of this classical project scheduling problem as a CP model is quite straightforward. We add a fictitious task with duration zero that corresponds to the end of the project. We thus consider the set of tasks $TASKS = \{1, \ldots, N\}$ where N is the fictitious end task. Let DUR_i be the duration of task i. The precedences between tasks are represented by a set of arcs, $ARCS$ (an arc $(i, j) \in ARCS$ symbolizes that task i precedes task j). The fictitious task follows all tasks that have no successor.

We define decision variables $start_i$ for the start time of every task i. These variables take values within the interval $\{0, \ldots, HORIZON\}$ where $HORIZON$ is the upper bound on the total duration given by the sum of all task durations.

We obtain the following simple CP model:

$$\forall i \in TASKS : start_i \in \{0, \ldots, HORIZON\}$$
$$\forall (i, j) \in ARCS : start_i + DUR_i \leq start_j$$

2.3 Implementation of question 1

Since there are no side constraints, the earliest possible completion time is the earliest start of the fictitious task *N*. We can obtain this value without enumeration thanks to the propagation of constraints. After posting the precedence constraints we retrieve the earliest completion time *bestend* and set this value as the upper bound on the last task. This fixes the start and completion times for the tasks on the critical path; for all other tasks the start and completion times are reduced to the feasible intervals:

```
model "B-1 Stadium construction (CP)"
 uses "kalis"

 declarations
  N = 19                              ! Number of tasks in the project
                                      ! (last = fictitious end task)
  TASKS = 1..N
  ARC: dynamic array(range,range) of integer  ! Matrix of the adjacency graph
  DUR: array(TASKS) of integer        ! Duration of tasks
  HORIZON : integer                   ! Time horizon

  start: array(TASKS) of cpvar        ! Start dates of tasks
  bestend: integer
 end-declarations

 initializations from 'Data/b1stadium.dat'
  DUR ARC
 end-initializations

 HORIZON:= sum(j in TASKS) DUR(j)

 forall(j in TASKS) do
  0 <= start(j); start(j) <= HORIZON
 end-do

! Task i precedes task j
 forall(i, j in TASKS | exists(ARC(i, j))) do
  Prec(i,j):= start(i) + DUR(i) <= start(j)
  if not cp_post(Prec(i,j)) then
   writeln("Posting precedence ", i, "-", j, " failed")
   exit(1)
  end-if
 end-do

! Since there are no side-constraints, the earliest possible completion
! time is the earliest start of the fictitiuous task N
 bestend:= getlb(start(N))
 start(N) <= bestend
 writeln("Earliest possible completion time: ", bestend)

! For tasks on the critical path the start/completion times have been fixed
! by setting the bound on the last task. For all other tasks the range of
! possible start/completion times gets displayed.
 forall(j in TASKS) writeln(j, ": ", start(j))

end-model
```

2.4 Results for question 1

The model in the previous section prints the following output. The earliest completion time is 64 weeks. For every operation (task), its start time or interval of possible start times is indicated.

```
Earliest possible completion time: 64
1: 0
```

```
2: 2
3: 18
4: [18..29]
5: 27
6: 37
7: [26..61]
8: [43..59]
9: 43
10: [26..59]
11: [43..58]
12: 52
13: [28..63]
14: [18..53]
15: [26..60]
16: [46..61]
17: 54
18: 63
19: 64
```

2.5 Model formulation for question 2

This second problem is called *scheduling with project crashing*. To reduce the total duration of the project, we need to take into account the result of the preceding optimization run, *bestend*. This value is now the upper bound on all start and completion time intervals.

Furthermore, for every task i let $MAXW_i$ denote the maximum number of weeks by which the task may be shortened.

2.5.1 CP model

The durations of tasks are not fixed any more and are therefore now represented by decision variables $duration_i$ that take values in the range $\{DUR_i - MAXW_i, \ldots, DUR_i\}$.

With these additional variables the CP model now looks as follows:

$$\forall i \in TASKS : start_i \in \{0, \ldots, bestend\}$$
$$\forall i \in TASKS : duration_i \in \{DUR_i - MAXW_i, \ldots, DUR_i\}$$
$$\forall (i, j) \in ARCS : start_i + duration_i \leq start_j$$

This model does not include the objective function (*i.e.*, maximization of the total gain from finishing the project early). This objective is dealt with by the LP model (see the following sections). As a result of the constraint propagation in the CP model, we merely obtain reduced start time intervals for the operations; the durations need to be determined either by enumeration or as shown below, by the LP solution algorithm.

2.5.2 LP model

We introduce variables $start_i$ to represent the earliest start time of tasks i and variables $save_i$ that correspond to the number of weeks that we wish to save for every task i. The only constraints that are given are the precedences. A task j may only start if all its predecessors have finished, which translates into the following constraints: if there is an arc between i and j, then the completion time of i (calculated as $start_i + DUR_i - save_i$) must not be larger than the start time of j.

$$\forall (i, j) \in ARCS : start_i + DUR_i - save_i \leq start_j$$

The variables $save_i$ are bounded by the maximum reduction in weeks, $MAXW_i$. These constraints

must be satisfied for all tasks i except the last, fictitious task N.

$$\forall i \in TASKS \backslash \{N\} : save_i \leq MAXW_i$$

For the last task, the variable $save_N$ represents the number of weeks the project finishes earlier than the solution *bestend* calculated in answer to question 1. The new completion time of the project $start_N$ must be equal to the previous completion time minus the advance $save_N$, which leads to the following constraint:

$$start_N = bestend - save_N$$

The objective defined by the second question is to maximize the builder's profit. For every week finished early, he receives a bonus of BONUS k. In exchange, the savings in time for a task i costs $COST_i$ k (column 'Add. cost per week' of Table 1). We thus obtain the following objective function.

$$\text{maximize } BONUS \cdot save_N - \sum_{i \in TASKS \backslash \{N\}} COST_i \cdot save_i$$

The complete mathematical model consists of the constraints and the objective function explained above, and the non-negativity conditions for variables $start_i$ and $save_i$:

$$\begin{aligned}
&\text{maximize } BONUS \cdot save_N - \sum_{i \in TASKS \backslash \{N\}} COST_i \cdot save_i \\
&\forall (i,j) \in ARCS : start_i + DUR_i - save_i \leq start_j \\
&start_N = bestend - save_N \\
&\forall i \in TASKS \backslash \{N\} : save_i \leq MAXW_i \\
&\forall i \in TASKS : start_i \geq 0,\ save_i \geq 0
\end{aligned}$$

2.6 Implementation of question 2

The second problem can be solved with the following algorithm:

- Solve the CP problem for question 1 and retrieve the solution, in particular the earliest completion time *bestend*.
- Solve the CP problem for question 2 with the time horizon *bestend* and retrieve the start time intervals.
- Define and solve the LP model for the second problem, using the bounds on the start times from CP as bounds on the LP variables $start_i$.

For the implementation, we would like to build on the CP model that we have shown above in the solution to question 1. However, since there is no means of modifying constraints that have been posted to the Kalis solver, we cannot simply work with a single file. Instead, we are going to split the implementation into two model files, one with the definition of the algorithms and the LP problem, and a second, the submodel, with the definition and solving of the CP problems. Depending on the model parameter `MODE`, the CP model either defines the model for question 1 or question 2. Instead of printing out an error message if an infeasibility is detected while posting the constraints, the CP model now sends a failure message to the master problem.

```
model "B-1 Stadium construction (CP submodel)"
 uses "kalis", "mmjobs"
```

```
parameters
 MODE = 1                              ! Model version: 1 - fixed durations
                                       !                2 - variable dur.
 HORIZON = 100                         ! Time horizon
end-parameters

declarations
 N = 19                                ! Number of tasks in the project
                                       ! (last = fictitious end task)
 TASKS = 1..N
 ARC: dynamic array(range,range) of integer  ! Matrix of the adjacency graph
 DUR: array(TASKS) of integer          ! Duration of tasks
 MAXW: array(TASKS) of integer         ! Max. reduction of tasks (in weeks)

 start: array(TASKS) of cpvar          ! Start dates of tasks
 duration: array(TASKS) of cpvar       ! Durations of tasks

 lbstart,ubstart: array(TASKS) of integer  ! Bounds on start dates of tasks
 EVENT_FAILED=2                        ! Event code sent by submodel
end-declarations

initializations from 'Data/b1stadium.dat'
 DUR ARC
end-initializations

forall(j in TASKS) setdomain(start(j), 0, HORIZON)

if MODE = 1 then                       ! **** Fixed durations
 ! Precedence relations between tasks
 forall(i, j in TASKS | exists(ARC(i, j))) do
  Prec(i,j):= start(i) + DUR(i) <= start(j)
  if not cp_post(Prec(i,j)) then
   send(EVENT_FAILED,0)
  end-if
 end-do

 ! Earliest poss. completion time = earliest start of the fictitiuous task N
 start(N) <= getlb(start(N))
else                                   ! **** Durations are variables
 initializations from 'Data/b1stadium.dat'
  MAXW
 end-initializations

 forall(j in TASKS) setdomain(duration(j), DUR(j)-MAXW(j), DUR(j))

 ! Precedence relations between tasks
 forall(i, j in TASKS | exists(ARC(i, j))) do
  Prec(i,j):= start(i) + duration(i) <= start(j)
  if not cp_post(Prec(i,j)) then
   send(EVENT_FAILED,0)
  end-if
 end-do
end-if

! Pass solution data to the master model
 forall(i in TASKS) do
  lbstart(i):= getlb(start(i)); ubstart(i):= getub(start(i))
 end-do

 initializations to "raw:"
  lbstart as "shmem:lbstart" ubstart as "shmem:ubstart"
 end-initializations

end-model
```

After compiling the CP submodel, the master model first runs the version for question 1, and retrieves the start time intervals. It then executes the CP submodel again, but now in the form for

question 2 and with the time horizon *bestend*. The start time intervals from the solution of the second CP run are used in the subsequent definition of the LP problem.

```
model "B-1 Stadium construction (CP + LP) master model"
 uses "mmxprs", "mmjobs"

 forward procedure print_CP_solution(num: integer)

 declarations
  N = 19                               ! Number of tasks in the project
                                       ! (last = fictitious end task)
  TASKS = 1..N
  ARC: dynamic array(range,range) of integer  ! Matrix of the adjacency graph
  DUR: array(TASKS) of integer         ! Duration of tasks
  BONUS: integer                       ! Bonus per week finished earlier
  MAXW: array(TASKS) of integer        ! Max. reduction of tasks (in weeks)
  COST: array(TASKS) of real           ! Cost of reducing tasks by a week
  lbstart,ubstart: array(TASKS) of integer  ! Bounds on start dates of tasks
  HORIZON: integer                     ! Time horizon
  bestend: integer                     ! CP solution value

  CPmodel: Model                       ! Reference to the CP model
  msg: Event                           ! Termination message sent by submodel
 end-declarations

 initializations from 'Data/b1stadium.dat'
  DUR ARC MAXW BONUS COST
 end-initializations

 HORIZON:= sum(o in TASKS) DUR(o)

! **** First CP model ****

 res:= compile("b1stadium_sub.mos")    ! Compile the CP model
 load(CPmodel, "b1stadium_sub.bim")    ! Load the CP model
 setworkdir(CPmodel, ".")
 run(CPmodel, "MODE=1,HORIZON=" + HORIZON)  ! Solve first version of CP model
 wait                                  ! Wait until subproblem finishes
 msg:= getnextevent                    ! Get the termination event message
 if getclass(msg)<>EVENT_END then      ! Check message type
  writeln("Submodel 1 is infeasible")
  exit(1)
 end-if

 initializations from "raw:"
  lbstart as "shmem:lbstart" ubstart as "shmem:ubstart"
 end-initializations

 bestend:= lbstart(N)
 print_CP_solution(1)

! **** Second CP model ****

 run(CPmodel, "MODE=2,HORIZON=" + bestend)  ! Solve second version of CP model
 wait                                  ! Wait until subproblem finishes
 msg:= getnextevent                    ! Get the termination event message
 if getclass(msg)<>EVENT_END then      ! Check message type
  writeln("Submodel 2 is infeasible")
  exit(2)
 end-if

 ! Retrieve solution from memory
 initializations from "raw:"
  lbstart as "shmem:lbstart" ubstart as "shmem:ubstart"
 end-initializations
```

```
 print_CP_solution(2)

! **** LP model for second problem ****
 declarations
  start: array(TASKS) of mpvar        ! Start times of tasks
  save: array(TASKS) of mpvar         ! Number of weeks finished early
 end-declarations

! Objective function: total profit
 Profit:= BONUS*save(N) - sum(i in 1..N-1) COST(i)*save(i)

! Precedence relations between tasks
 forall(i,j in TASKS | exists(ARC(i,j)))
  Precm(i,j):= start(i) + DUR(i) - save(i) <= start(j)

! Total duration
 start(N) + save(N) = bestend

! Limit on number of weeks that may be saved
 forall(i in 1..N-1) save(i) <= MAXW(i)

! Bounds on start times deduced by constraint propagation
 forall(i in 1..N-1) do
  lbstart(i) <= start(i); start(i) <= ubstart(i)
 end-do

! Solve the second problem: maximize the total profit
 setparam("XPRS_VERBOSE", true)
 setparam("XPRS_PRESOLVE", 0)   ! We use constraint propagation as preprocessor
 maximize(Profit)

! Solution printing
 writeln("Total profit: ", getsol(Profit))
 writeln("Total duration: ", getsol(start(N)), " weeks")
 forall(i in 1..N-1)
  write(strfmt(i,2), ": ", strfmt(getsol(start(i)),-3),
   if(i mod 6 = 0,"\n",""))
 writeln

!*********************************************************************
 procedure print_CP_solution(num: integer)
  writeln("CP solution (version ", num, "):")
  writeln("Earliest possible completion time: ", lbstart(N), " weeks")
  forall(i in 1..N-1)
   write(i, ": ", lbstart(i), if(lbstart(i)<ubstart(i), "-"+ubstart(i), ""),
        if(i mod 6 = 0, "\n", ", "))
 end-procedure

end-model
```

2.7 Results for question 2

The model above produces the following output. Setting `XPRS_VERBOSE` to `true` makes the software display the log of the LP solver: some information about the problem size (numbers of constraints, variables, non-zero coefficients, and MIP entities) and a log of the simplex algorithm. If you re-run the model without the bound updates from CP to LP you may observe a slightly larger number of Simplex iterations.

```
CP solution (version 1):
Earliest possible completion time: 64 weeks
1: 0, 2: 2, 3: 18, 4: 18-29, 5: 27, 6: 37
7: 26-61, 8: 43-59, 9: 43, 10: 26-59, 11: 43-58, 12: 52
13: 28-63, 14: 18-53, 15: 26-60, 16: 46-61, 17: 54, 18: 63
CP solution (version 2):
Earliest possible completion time: 52 weeks
```

```
1: 0-12, 2: 2-14, 3: 15-27, 4: 15-37, 5: 23-35, 6: 31-43
7: 21-62, 8: 36-60, 9: 36-48, 10: 21-60, 11: 36-60, 12: 43-55
13: 22-63, 14: 15-57, 15: 21-62, 16: 38-62, 17: 45-57, 18: 51-63

Reading Problem /xprs_6cf5_404d0008
Problem Statistics
          28 (      0 spare) rows
          38 (      0 spare) structural columns
          83 (      0 spare) non-zero elements
Global Statistics
           0 entities        0 sets        0 set members

   Its         Obj Value      S   Ninf  Nneg        Sum Inf  Time
     0        360.000300      D     17     0      29.000010     0
    17         87.000000      D      0     0        .000000     0
Optimal solution found
Total profit: 87
Total duration: 54 weeks
 1: 0   2: 2   3: 15  4: 15  5: 23  6: 31
 7: 23  8: 36  9: 36 10: 23 11: 36 12: 45
13: 25 14: 15 15: 23 16: 39 17: 47 18: 53
```

3 Combining CP and MIP

Application problems often combine different subproblems that are solved better with one or another solver, making the complete problem difficult or unmanageable for a single solver. A typical example is production planning and scheduling applications. The long-term (planning) aspects are usually more easily handled by LP solvers whereas the short-term (scheduling) subproblems are better suited for CP solvers. Solving these two parts completely independent of each other may lead to infeasible scheduling subproblems or plans that do not correspond to the reality of production. A possible solution to this dilemma is to iteratively solve LP planning problems and CP scheduling problems, until a feasible schedule for the planned quantities is obtained.

Another method of combined MIP-CP problem solving that provides a tighter integration of the two techniques consists of solving CP subproblems for generating cuts at the nodes of a MIP Branch-and-Bound search. This technique has already been applied successfully to several large-scale planning and scheduling applications by PSA and BASF (see for instance the description of hybrid MIP-CP algorithms implemented with Mosel in [BP03] and [Sad04]). This type of combination is more technical than sequential CP and LP/MIP solving since it requires the developer of the algorithm to interact with the MIP search at every node.

Cut generation algorithms can be implemented with the help of the Xpress Optimizer callbacks (see the 'Mosel Language Reference Manual' for the definition of callbacks with Mosel and the 'Xpress Optimizer Reference Manual' for an explanation of the Optimizer callback functions).

The original description of the example in this section was published in [JG01]. A prototype implementation was developed by N. Pisaruk in the context of the EU-project LISCOS.

3.1 Example: Machine assignment and sequencing

We need to produce 12 products on a set of three machines. Each machine may produce all of the products but processing times and costs vary (Table 2). Furthermore, for every product we are given its release and due dates (Table 3). We wish to determine a production plan for all products that minimizes the total production cost.

Table 2: Machine-dependent production costs and durations

	Production costs			Durations		
Prod. \ Mach.	**1**	**2**	**3**	**1**	**2**	**3**
1	12	6	7	10	14	13
2	13	6	10	7	9	8
3	10	4	6	11	17	15
4	8	4	5	6	9	12
5	12	6	7	4	6	10
6	10	5	6	2	3	4
7	7	4	5	10	15	16
8	9	5	5	8	11	12
9	10	5	7	10	14	13
10	8	4	5	8	11	14
11	15	8	9	9	12	16
12	13	7	7	3	5	6

Table 3: Release dates and due dates of products

Product	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Release	2	4	5	7	9	0	3	6	11	2	3	4
Due date	32	33	36	37	39	34	30	26	36	38	31	22

3.2 Model formulation

We are going to represent this problem by two subproblems: the machine assignment problem and the sequencing of operations on machines. The former is implemented by a MIP model; the latter is formulated as a CP (single machine) problem.

3.2.1 MIP model

Let $COST_{pm}$ denote the production cost and DUR_{pm} the processing time of product p ($p \in PRODS$, the set of products) on machine m ($m \in MACH$, the set of machines).

To formulate the machine assignment problem we introduce binary variables use_{pm} that take the value 1 if product p is produced on machine m and zero otherwise. The objective function is then given as

$$\text{minimize} \sum_{p \in PRODS} \sum_{m \in MACH} COST_{pm} \cdot use_{pm}$$

The assignment constraints expressing that each order needs exactly one machine for processing it are defined as follows:

$$\forall p \in PRODS : \sum_{m \in MACH} use_{pm} = 1$$

In addition to these constraints that already fully state the problem we may define some additional constraints to strengthen the LP relaxation. All production takes place between the earliest release date and the latest due date. If we denote the length of this interval by MAX_LOAD, we may formulate the following valid inequalities expressing that the total processing time of products assigned to a machine cannot exceed MAX_LOAD:

$$\forall m \in MACH : \sum_{p \in PRODS} DUR_{pm} \cdot use_{pm} \leq MAX\_LOAD$$

3.2.2 CP model

Once the set of operations assigned to machine m, $ProdMach_m$ ($ProdMach_m \subseteq PRODS$), is known, we obtain the following sequencing problem for this machine:

$$\forall p \in ProdMach_m : start_p \in \{REL_p, \ldots, DUE_p - DUR_{pm}\}$$
$$\forall p, q \in ProdMach_m, p < q : start_p + DUR_{pm} \leq start_q \vee start_q + DUR_{qm} \leq start_p$$

3.3 Implementation

We are using the following algorithm for modeling and solving this problem:

```
Define the MIP machine assignment problem.
Define the operations of the CP model.
Start the MIP Branch-and-Bound search.
At every node of the MIP search:
   while function generate_cuts returns true
      re-solve the LP-relaxation

Function generate_cuts
   for all machines m call generate_cut_machine(m)
   if at least one cut has been generated
      Return true
   otherwise
      Return false

Function generate_cut_machine(m)
   Collect all operations assigned to machine m
   if more than one operation assigned to m
      Solve the CP sequencing problem for m
      if sequencing succeeds
         Save the solution
      otherwise
         Add an infeasibility cut for machine m to the MIP
```

The implementation of this model is split into two Mosel models: the first, `sched_main.mos`, contains the MIP master problem and the definition of the cut generation algorithm. The second model, `sched_sub.mos`, implements the CP single machine sequencing model.

The first part of the master model sets up the data arrays, compiles and loads the CP submodel, calls subroutines for the model definition and problem solving, and finally produces some summary result output. We have defined the filename of the data file as a *parameter* to be able to change the name of the data file at the execution of the model without having to change the model source. Correspondingly, all data, including the sizes of index sets, are read in from file. At first, we read in only the values of `NP` and `NM`. Subsequently, when declaring the sets and arrays that make use of these values, `NP` and `NM` are known and the arrays are created as fixed arrays. Otherwise, if their indexing sets are not known, these arrays would automatically be declared as dynamic arrays and for all but arrays of basic types (real, integer, *etc.*) we have to create their entries explicitly.

```
model "Schedule (MIP + CP) master problem"
 uses "mmsystem", "mmxprs", "mmjobs"

 parameters
```

```
 DATAFILE = "Data/sched_3_12.dat"
 VERBOSE = 1
end-parameters

forward procedure define_MIP_model
forward procedure setup_cutmanager
forward public function generate_cuts: boolean
forward public procedure print_solution

declarations
 NP: integer                            ! Number of operations (products)
 NM: integer                            ! Number of machines
end-declarations

initializations from DATAFILE
 NP NM
end-initializations

declarations
 PRODS = 1..NP                          ! Set of products
 MACH = 1..NM                           ! Set of machines

 REL: array(PRODS) of integer           ! Release dates of orders
 DUE: array(PRODS) of integer           ! Due dates of orders
 MAX_LOAD: integer                      ! max_p DUE(p) - min_p REL(p)
 COST: array(PRODS,MACH) of integer     ! Processing cost of products
 DUR: array(PRODS,MACH) of integer      ! Processing times of products
 starttime: real                        ! Measure program execution time
 ctcut: integer                         ! Counter for cuts
 solstart: array(PRODS) of integer
                                        ! **** MIP model:
 use: array(PRODS,MACH) of mpvar        ! 1 if p uses machine m, otherwise 0
 Cost: linctr                           ! Objective function

 totsolve,totCP: real                   ! Time measurement
 ctrun: integer                         ! Counter of CP runs
 CPmodel: Model                         ! Reference to the CP sequencing model
 ev: Event                              ! Event
 EVENT_SOLVED=2                         ! Event codes sent by submodels
 EVENT_FAILED=3
end-declarations

! Read data from file
initializations from DATAFILE
 REL DUE COST DUR
end-initializations

! **** Problem definition ****
define_MIP_model                        ! Definition of the MIP model
res:=compile("sched_sub.mos")           ! Compile the CP model
load(CPmodel, "sched_sub.bim")          ! Load the CP model

! **** Solution algorithm ****
starttime:= gettime
setup_cutmanager                        ! Settings for the MIP search

totsolve:= 0.0
initializations to "raw:"
 totsolve as "shmem:solvetime"
 REL as "shmem:REL" DUE as "shmem:DUE"
end-initializations

minimize(Cost)                          ! Solve the problem

writeln("Number of cuts generated: ", ctcut)
writeln("(", gettime-starttime, "sec) Best solution value: ", getobjval)
initializations from "raw:"
```

```
 totsolve as "shmem:solvetime"
end-initializations
writeln("Total CP solve time: ", totsolve)
writeln("Total CP time: ", totCP)
writeln("CP runs: ", ctrun)
```

The MIP model corresponds closely to the mathematical model that we have seen in the previous section.

```
procedure define_MIP_model

! Objective: total processing cost
 Cost:= sum(p in PRODS, m in MACH) COST(p,m) * use(p,m)

! Each order needs exactly one machine for processing
 forall(p in PRODS) sum(m in MACH) use(p,m) = 1

! Valid inequalities for strengthening the LP relaxation
 MAX_LOAD:= max(p in PRODS) DUE(p) - min(p in PRODS) REL(p)
 forall(m in MACH) sum(p in PRODS) DUR(p,m) * use(p,m) <= MAX_LOAD

 forall(p in PRODS, m in MACH) use(p,m) is_binary

end-procedure
```

The cut generation callback function `generate_cuts` is called at least once per MIP node. For every machine, it checks whether the assigned operations can be scheduled or whether an infeasibility cut needs to be added. If any cuts have been added, the LP relaxation needs to be re-solved and the cut generation function will be called again, until no more cuts are added. It is important to set and re-set the values of `XPRS_solutionfile` as shown in our example at the beginning and end of this function if it accesses Xpress Optimizer solution values.
The function `generate_cut_machine` first collects all tasks that have been assigned to the given machine `m` into the set `ProdMach` by calling the procedure `products_on_machine`. If there are still unassigned tasks the returned set is empty, otherwise, if the set has more than one element it tries to solve the sequencing subproblem (function `solve_CP_problem`). If this problem cannot be solved, then the function adds a cut to the MIP problem that makes the current assignment of operations to this machine infeasible.

```
procedure products_on_machine(m: integer, ProdMach: set of integer)

 forall(p in PRODS) do
  val:=getsol(use(p,m))
  if (val > 0 and val < 1) then
  ProdMach:={}
   break
  elif val>0.5 then
   ProdMach+={p}
  end-if
 end-do

 end-procedure

!-------------------------------------------------------------------
! Generate a cut for machine m if the sequencing subproblem is infeasible
 function generate_cut_machine(m: integer): boolean
  declarations
   ProdMach: set of integer
  end-declarations

  ! Collect the operations assigned to machine m
  products_on_machine(m, ProdMach)
```

```
 ! Solve the sequencing problem (CP model): if solved, save the solution,
 ! otherwise add an infeasibility cut to the MIP problem
  size:= getsize(ProdMach)
  returned:= false
  if (size>1) then
   if not solve_CP_problem(m, ProdMach, 1) then
    Cut:= sum(p in ProdMach) use(p,m) - (size-1)
    if VERBOSE > 2 then
     writeln(m,": ", ProdMach, " <= ", size-1)
    end-if
    addcut(1, CT_LEQ, Cut)
    returned:= true
   end-if
  end-if

 end-function

!-------------------------------------------------------------------
! Cut generation callback function
 public function generate_cuts: boolean
  returned:=false; ctcutold:=ctcut

  setparam("XPRS_solutionfile", 0)
  forall(m in MACH) do
   if generate_cut_machine(m) then
    returned:=true                    ! Call function again for this node
    ctcut+=1
   end-if
  end-do
  setparam("XPRS_solutionfile", 1)
  if returned and VERBOSE>1 then
   writeln("Node ", getparam("XPRS_NODES"), ": ", ctcut-ctcutold,
           " cut(s) added")
  end-if

 end-function
```

The solving of the CP model is started from the function `solve_CP_problem` that writes out the necessary data to shared memory and starts the execution of the submodel contained in the file `sched_sub.mos`.

```
 function solve_CP_problem(m: integer, ProdMach: set of integer,
                           mode: integer): boolean
  declarations
   DURm: dynamic array(range) of integer
   sol: dynamic array(range) of integer
   solvetime: real
  end-declarations

 ! Data for CP model
  forall(p in ProdMach) DURm(p):= DUR(p,m)
  initializations to "raw:"
   ProdMach as "shmem:ProdMach"
   DURm as "shmem:DURm"
  end-initializations

 ! Solve the problem and retrieve the solution if it is feasible
  startsolve:= gettime
  returned:= false
  if(getstatus(CPmodel)=RT_RUNNING) then
    fflush
    writeln("CPmodel is running")
    fflush
    exit(1)
  end-if
```

```
 ctrun+=1
 run(CPmodel, "NP=" + NP + ",VERBOSE=" + VERBOSE + ",MODE=" + mode)
 wait                                  ! Wait for a message from the submodel
 ev:= getnextevent                     ! Retrieve the event
 if getclass(ev)=EVENT_SOLVED then
  returned:= true
  if mode = 2 then
   initializations from "raw:"
    sol as "shmem:solstart"
   end-initializations
   forall(p in ProdMach) solstart(p):=sol(p)
  end-if
 elif getclass(ev)<>EVENT_FAILED then
  writeln("Problem with Kalis")
  exit(2)
 end-if
 wait
 dropnextevent                         ! Ignore "submodel finished" event
 totCP+= (gettime-startsolve)
end-function
```

We complete the MIP model with settings for the cut manager and the definition of the integer solution callback. The Mosel comparison tolerance is set to a slightly larger value than the tolerance applied by Xpress Optimizer. It is important to switch the LP presolve off since we interfere with the matrix during the execution of the algorithm (alternatively, it is possible to fine-tune presolve to use only non-destructive algorithms). Sufficiently large space for cuts and cut coefficients should be reserved in the matrix. We also enable output printing by the Optimizer and choose among different MIP log frequencies (depending on model parameter `VERBOSE`.

```
procedure setup_cutmanager
 setparam("XPRS_CUTSTRATEGY", 0)           ! Disable automatic cuts
 feastol:= getparam("XPRS_FEASTOL")        ! Get Optimizer zero tolerance
 setparam("zerotol", feastol * 10)         ! Set comparison tolerance of Mosel
 setparam("XPRS_PRESOLVE", 0)              ! Disable presolve
 setparam("XPRS_MIPPRESOLVE", 0)           ! Disable MIP presolve
 command("KEEPARTIFICIALS=0")              ! No global red. cost fixing
 setparam("XPRS_SBBEST", 0)                ! Turn strong branching off
 setparam("XPRS_HEURSTRATEGY", 0)          ! Disable MIP heuristics
 setparam("XPRS_EXTRAROWS", 10000)         ! Reserve space for cuts
 setparam("XPRS_EXTRAELEMS", NP*30000)     ! ... and cut coefficients
 setcallback(XPRS_CB_CM, "generate_cuts")  ! Define the cut manager callback
 setcallback(XPRS_CB_UIS, "print_solution")! Define the integer solution cb.
 setparam("XPRS_COLORDER", 2)
 case VERBOSE of
 1: do
     setparam("XPRS_VERBOSE", true)
     setparam("XPRS_MIPLOG", -200)
    end-do
 2: do
     setparam("XPRS_VERBOSE", true)
     setparam("XPRS_MIPLOG", -100)
    end-do
 3: do                                     ! Detailed MIP output
     setparam("XPRS_VERBOSE", true)
     setparam("XPRS_MIPLOG", 3)
    end-do
 end-case

end-procedure
```

The definition of the integer solution callback is, in parts, similar to the function `generate_cut_machine`. To obtain a detailed solution output we need to re-solve all CP subproblems, this time with run `MODE` two, meaning that the CP model writes its solution

information to shared memory.

```
public procedure print_solution
 declarations
  ProdMach: set of integer
 end-declarations

 writeln("(",gettime-starttime, "sec) Solution ",
         getparam("XPRS_MIPSOLS"), ": Cost: ", getsol(Cost))

 if VERBOSE > 1 then
  forall(p in PRODS) do
   forall(m in MACH) write(getsol(use(p,m))," ")
   writeln
  end-do
 end-if

 if VERBOSE > 0 then
  forall(m in MACH) do
   ProdMach:= {}

 ! Collect the operations assigned to machine m
   products_on_machine(m, ProdMach)

   Size:= getsize(ProdMach)
   if Size > 1 then
  ! (Re)solve the CP sequencing problem
    if not solve_CP_problem(m, ProdMach, 2) then
     writeln("Something wrong here: ", m, ProdMach)
    end-if
   elif Size=1 then
    elem:=min(p in ProdMach) p
    solstart(elem):=REL(elem)
   end-if
  end-do

! Print out the result
  forall(p in PRODS) do
   msol:=sum(m in MACH) m*getsol(use(p,m))
   writeln(p, " -> ", msol,": ", strfmt(solstart(p),2), " - ",
           strfmt(DUR(p,round(msol))+solstart(p),2), "  [",
           REL(p), ", ", DUE(p), "]")
  end-do
  writeln("Time: ", gettime - starttime, "sec")
  writeln
  fflush
 end-if
end-procedure
```

The following code listing shows the complete CP submodel. At every execution, the set of tasks assigned to one machine and the corresponding durations are read from shared memory. The disjunctions between pairs of tasks are posted explicitly to be able to stop the addition of constraints if an infeasibility is detected during the definition of the problem. The search stops at the first feasible solution. If a solution was found, it is passed back to the master model if the model parameter `MODE` has the value two. In every case, after termination of the CP search the submodel sends a solution status event back to the master model.

```
model "Schedule (MIP + CP) CP subproblem"
 uses "kalis", "mmjobs" , "mmsystem"

 parameters
  VERBOSE = 1
  NP = 12                                ! Number of products
  MODE = 1                               ! 1 - decide feasibility
```

```
                                        ! 2 - return complete solution
 end-parameters

 startsolve:= gettime

 declarations
  PRODS = 1..NP                         ! Set of products
  ProdMach: set of integer
 end-declarations

 initializations from "raw:"
  ProdMach as "shmem:ProdMach"
 end-initializations

 finalize(ProdMach)

 declarations
  REL: array(PRODS) of integer          ! Release dates of orders
  DUE: array(PRODS) of integer          ! Due dates of orders
  DURm: array(ProdMach) of integer      ! Processing times on machine m
  solstart: array(ProdMach) of integer  ! Solution values for start times

  start: array(ProdMach) of cpvar       ! Start times of tasks
  Disj: array(range) of cpctr           ! Disjunctive constraints
  Strategy: array(range) of cpbranching ! Enumeration strategy
  EVENT_SOLVED=2                        ! Event codes sent by submodels
  EVENT_FAILED=3
  solvetime: real
 end-declarations

 initializations from "raw:"
  DURm as "shmem:DURm" REL as "shmem:REL" DUE as "shmem:DUE"
 end-initializations

! Bounds on start times
 forall(p in ProdMach) setdomain(start(p), REL(p), DUE(p)-DURm(p))

! Disjunctive constraint
 ct:= 1
 forall(p,q in ProdMach| p<q) do
  Disj(ct):= start(p) + DURm(p) <= start(q) or start(q) + DURm(q) <= start(p)
  ct+= 1
 end-do

! Post disjunctions to the solver
 nDisj:= ct; j:=1; res:= true
 while (res and j<nDisj) do
  res:= cp_post(Disj(j))
  j+=1
 end-do

! Solve the problem
 if res then
  Strategy(1):= settle_disjunction(Disj)
  Strategy(2):= assign_and_forbid(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX,
                                  start)
  cp_set_branching(Strategy)
  res:= cp_find_next_sol
 end-if

! Pass solution to master problem
 if res then
  forall(p in ProdMach) solstart(p):= getsol(start(p))
  if MODE=2 then
   initializations to "raw:"
    solstart as "shmem:solstart"
   end-initializations
```

```
  end-if
  send(EVENT_SOLVED,0)
 else
  send(EVENT_FAILED,0)
 end-if

! Update total running time measurement
 initializations from "raw:"
  solvetime as "shmem:solvetime"
 end-initializations
 solvetime+= gettime-startsolve
 initializations to "raw:"
  solvetime as "shmem:solvetime"
 end-initializations

end-model
```

3.4 Results

The best solution produced for the data set `sched_3_12` is the following :

```
Cost: 92
1 -> 3:  2 - 15  [2, 32]
2 -> 3: 15 - 23  [4, 33]
3 -> 2: 15 - 32  [5, 36]
4 -> 1: 24 - 30  [7, 37]
5 -> 2: 32 - 38  [9, 39]
6 -> 2:  0 -  3  [0, 34]
7 -> 1:  3 - 13  [3, 30]
8 -> 1: 16 - 24  [6, 26]
9 -> 3: 23 - 36  [11, 36]
10 -> 1: 30 - 38  [2, 38]
11 -> 2:  3 - 15  [3, 31]
12 -> 1: 13 - 16  [4, 22]
```

A total of 1604 cuts are added to the MIP problem by 2691 CP model runs and the Branch-and-Bound search explores 12295 nodes. Optimality is proven within a few seconds on a Pentium IV PC.

It is possible to implement this problem entirely either with Xpress Optimizer or with Xpress Kalis. However, already for this three machines – 12 jobs instance the problem is extremely hard for either technique on its own. With CP it is difficult to prove optimality and with MIP the formulation of the disjunctions makes the definition of a large number of binary variables necessary (roughly in the order of $number\_of\_machines \cdot number\_of\_products^2$) which makes the problem impracticable to deal with.

3.5 Parallel solving of CP subproblems

Instead of solving the CP single-machine subproblems at every MIP node sequentially, we can modify our Mosel models to solve the subproblems in parallel—especially when working on a multiprocessor machine this may speed up the cut generation process and hence shorten the total run time. We modify the algorithm of Section 3.3 as follows:

```
Define the MIP machine assignment problem.
Define the operations of the CP model.
Start the MIP Branch-and-Bound search.
At every node of the MIP search:
   while function generate_cuts returns true
      re-solve the LP-relaxation

Function generate_cuts
   Collect all machines that are fully assigned into set ToSolve
   for all machines m ∈ ToSolve call start_CP_model(m)
   Wait for the solution status messages from all submodels
     if submodel m is infeasible
        Add an infeasibility cut for machine m to the MIP
   if at least one cut has been generated
      Return true
   otherwise
      Return false

Procedure start_CP_model(m)
   Collect all operations assigned to machine m
   Write data for this machine to memory
   Start the submodel execution
```

The modified version of the function `generate_cuts` looks as follows. For the full example code the reader is referred to the set of User Guide examples provided with the Xpress Kalis distribution (files `sched_mainp.mos` and `sched_subp.mos`).

```
! Collect the operations assigned to machine m
 procedure products_on_machine(m: integer)

  NumOp(m):=0
  forall(p in PRODS) do
   val:=getsol(use(p,m))
   if (! not isintegral(use(p,m)) !) (val > 0 and val < 1) then
    NumOp(m):=0
    break
   elif val>0.5 then
    NumOp(m)+=1
    OpMach(m,NumOp(m)):= p
   end-if
  end-do

 end-procedure

!-------------------------------------------------------------------
! Add an infeasibility cut for machine m to the MIP problem
 procedure add_cut_machine(m: integer)

  Cut:= sum(p in 1..NumOp(m)) use(OpMach(m,p),m) - (NumOp(m)-1)
  if VERBOSE > 1 then
   write(m,": ")
   forall(p in 1..NumOp(m)) write(OpMach(m,p), " ")
   writeln(" <= ", NumOp(m)-1)
  end-if
  addcut(1, CT_LEQ, Cut)

 end-procedure
```

The implementation of the CP submodels remains largely unchanged, with the exception of the

labels employed for passing data via shared memory: we append the machine index to every data item to be able to distinguish between the data used by the different subproblems running in parallel.

For the data set `sched_3_12.dat` we have observed only a few percent decrease of the total running time on a dual processor machine using the parallel implementation: in many nodes only a single CP subproblem is solved and if there are several subproblems to be solved their execution may be of quite different length. For instances with a larger number of machines the parallelization is likely to show more effect.

4 Summary

The examples in this whitepaper show two schemes of combining CP and LP/MIP modeling and problem solving. Whilst the first is a loosly coupled combined algorithm (execution of a CP and an LP problem in sequence) the second is an example of a fairly tight integration, using CP as cut generation algorithm for a MIP branch-and-cut search. Many other schemes are possible (for instance, iterative solving of a series of MIP and CP subproblems as in [Tim02])—an hybridization scheme must always be chosen depending on the particular structure of an application problem and its typical data instances.

Although the combination of different solving techniques has proven successful in a number of applications, the author would like to issue a warning to the interested reader that it is seldom worthwhile spending time on a problem that can be tackled by one of the techniques separately. Hybrid solution algorithms need to be developed, implemented, and tested on a case-by-case basis, meaning a considerable investment in terms of development effort and requiring a good understanding of the solution methods and solvers involved.

Bibliography

[BP03] A. Bockmayr and N. Pisaruk. Detecting Infeasibility and Generating Cuts for MIP Using CP. In *Proceedings of CP-AI-OR 2003*, pages 24—34, Montreal, 2003.

[JG01] V. Jain and I.E. Grossmann. Algorithms for hybrid MILP/CLP models for a class of optimization problems. *INFORMS J. Computing*, 13(4):258—276, 2001.

[Sad04] R. Sadykov. A Hybrid Branch-and-Cut Algorithm for the One-Machine Scheduling Problem. In J.C. Régin and M. Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2004*, pages 409—414, Berlin, 2004. Springer.

[Tim02] C. Timpe. Solving planning and scheduling problems with combined integer and constraint programming. *ORSpectrum*, 24(4):431—448, 2002.