

Efficient modeling in ILOG OPL-CPLEX Development System



Efficient modeling in

ILOG OPL-CPLEX Development System

White Paper

© ILOG, June 2007 – Do not duplicate without permission.

ILOG, CPLEX and their respective logotypes are registered trademarks.

All other company and product names are trademarks or registered trademarks of their respective holders.

The material presented in this document is summary in nature, subject to change,
not contractual and intended for general information only and does not constitute a representation.

Contents

Introduction	2
Relation between memory and running times.....	2
Sparse model data	3
Efficient data initialization.....	7
Efficient model construction	10
Conclusion	12

Introduction

ILOG OPL-CPLEX Development System is a rapid development system for optimization models with interfaces to embed models into standalone applications. It streamlines modeling with high-level data types and an algebraic language specifically designed for building optimization models. This results in compact, straightforward code that is easier to maintain than general-purpose languages such as Java™ and C++. OPL models can be linked to Microsoft Excel spreadsheets or popular databases, and OPL's syntax cleanly separates models from input data so that no changes are needed to move a model from a test data file to a real production database. OPL directly supports ILOG CPLEX, enabling models written in OPL to be solved with CPLEX algorithms. Finally, there are two ways to turn OPL models into standalone applications. OPL models can be integrated into custom applications via application programming interfaces (APIs) for Java, C++ and Microsoft .NET. Alternately, OPL models can be deployed via ILOG Optimization Decision Manager (ODM), a complete system for building interactive planning applications. ODM supports what-if analysis, soft constraints and trade-offs among business goals so that business managers can utilize OPL models for analysis and planning. Together, these features make OPL the fastest system for building and deploying CPLEX applications.

A common concern with any rapid development system is that the savings in development time may come at the cost of excessive computing overhead. In other words, memory consumption or runtimes could be excessive when compared with calling CPLEX directly. Ideally, a modeling system like OPL should have only a small impact on memory and running time – the performance cost of the modeling system should be low when compared with the effort to solve the model. ILOG achieved this goal when it released OPL 4.0 in 2005, using ILOG Concert Technology and numerous changes to the OPL system. By cutting computation time and memory requirements, OPL solves very large optimization models with the simplicity of a modeling system. In fact, developers have been more successful in building applications with OPL than with code written to directly call CPLEX. One reason is that errors and inefficiencies can be readily spotted in OPL code, but they tend to be hidden in a general purpose code like Java or C++. While many of the techniques described in this paper can be applied to models built with general-purpose programming languages, they are far easier to implement with OPL.

Even so, as with any programming language, proper design is important to produce an efficient OPL model that will scale to solve large optimization problems. This paper describes techniques to compress models and data so that the runtime performance and memory consumption of OPL applications are virtually the same as pure CPLEX applications.

Relation between memory and running times

Performance tuning focuses on two areas: speed and memory. Greater speed reduces the time to solve a model and allows more what-if scenarios in a fixed amount of time, while reduced memory consumption allows larger models to be run on a particular computer.

When solving an OPL model, the role of the OPL system is to process model data and convert the algebraic model into the sparse matrix coefficient representation used internally by CPLEX. To evaluate performance in terms of runtime, it is important to distinguish between the time OPL requires to initialize a mathematical programming instance and the time CPLEX takes to solve this instance. To evaluate performance in terms of memory, one should compare the memory overhead required by OPL against the memory required when the same model is implemented with a traditional programming language. Once the coefficient matrix is passed to CPLEX, the time and speed CPLEX takes to solve the model should be equivalent regardless of whether OPL or a traditional programming language is used. In this discussion, the primary concern is the memory and speed required for OPL to fully initialize the optimization instance.

With optimization models, there is a relationship between memory use and runtime. Eliminating redundant data through sparse data structures can both reduce memory requirements and improve the time to initialize and solve the optimization model. The worst case occurs when optimization requires more physical memory (RAM) than available, forcing the computer to use *virtual memory*. The computer uses hard-disk space to supplement RAM, repeatedly swapping data between the physical memory and the disk. The relatively slow speed of disk access dramatically increases running time. Virtual memory is generally OK in small amounts, but for top performance, the optimization model should be fully loaded into the physical memory with enough space remaining for calculations. Otherwise, OPL and CPLEX must access the hard disk throughout the solving process for the optimization model. But even when no virtual memory is required, the model should be built to efficiently use memory to optimize performance.

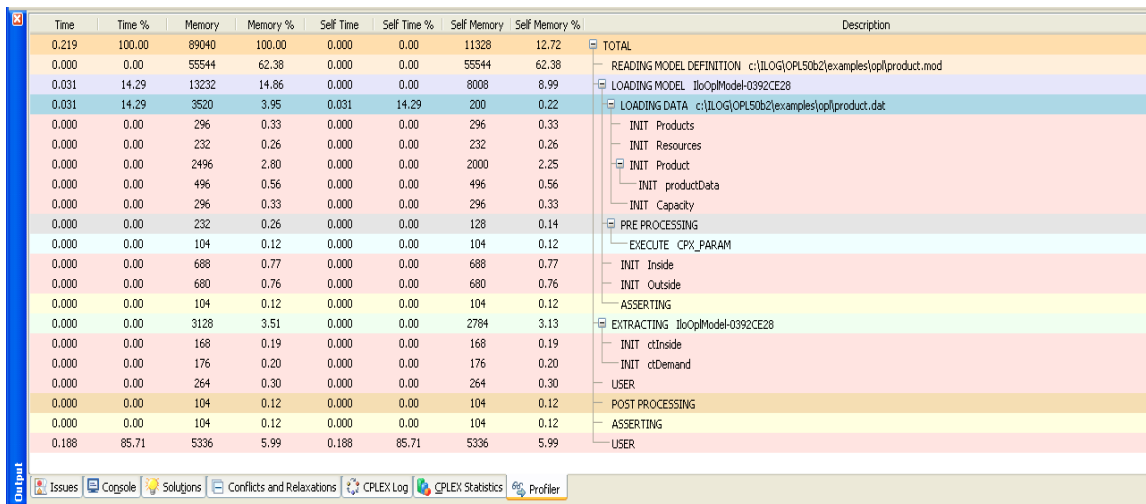


Figure 1: Profile in ILOG OPL IDE

The Profiler in the OPL Integrated Development Environment (IDE) helps developers track the memory usage of model elements. It offers more detailed performance analysis than the general-purpose tools provided by operating systems.

In particular, the Profiler shows how much memory and time is used to initialize OPL data. The Profiler reports information in a tree structure, and provides totals for groups as well as individual values. With a large, complex model, the Profiler can direct the developer to the data elements that consume the most memory and, therefore, are the best candidates for tuning. Detailed instructions for operating the Profiler can be found in the section Tutorial: **Profiling the Execution of a model**.

Sparse model data

Developers select CPLEX because it is the best system for solving large, complex optimization problems. The combination of various decision alternatives produces large problems. For instance, a manufacturer must make a vast number of decisions about products, machines, factories and warehouses, while a trucking company must do so for trucks, drivers and routes. Regardless of the application area, most optimization models also involve choices in terms of time.

Not all combinations are valid. In a manufacturing model, some products may not be produced at all the factories. Similarly, some airports are not equipped to handle certain airplanes, and some workers are not qualified to perform certain tasks. The key to obtaining an efficient model is to use only the valid combinations, rather than defining data and decision variables for every single

combination. For instance, there are 1 million individual combinations for assigning 1,000 workers to 1,000 tasks. However, if each worker is capable of doing only one of five tasks, then there are only 5,000 valid combinations of workers and tasks. Now, suppose there is a time element – the workers have to be scheduled to tasks for each day. There could be tasks that cannot be done on certain days, and days when some workers are unavailable. By working with just the valid combinations of workers, tasks and days, the size of the data model can be significantly reduced.

The reason this is so important for optimization modeling is that a typical optimization model is filled with data and decision variables that are indexed as combinations. Removing the invalid combinations eliminates the need to create, store and iterate unnecessary values, so less time and memory are required to build and solve an optimization model. This allows larger models to be solved, as well as a reduction in time to solve existing models.

OPL provides two key features for working efficiently with sparse data: *tuple sets* to specify the subsets of valid combinations, and *slicing* to help OPL iterate efficiently over the subsets. The only hard thing is to remember to use sparse data when starting to build the optimization model! A *tuple* contains a combination of data elements, generally integer and string data. Tuple sets are defined in the ***OPL Language Reference Manual*** under ***OPL, the Modeling Language > Data Types > Data Structures***. The following example illustrates a tuple set using a problem for assigning workers to tasks.

Remember: Use sparse data when starting to build the optimization model!

The first step is to declare the tuple. This tuple declaration connects workers to tasks:

```
tuple workTask {
    string  workerID;
    string  taskID;
};
```

This creates a new type, called `workTask`, which will be used to hold pairs of workers and tasks. Now, a tuple set of valid pairs of workers and tasks is created:

```
setof(workTask) WorkerTasks = ...;
```

The sparse set `WorkerTasks` will be initialized via an OPL data file (a file with a “.dat” extension), which can take values from an external source such as a database. Here are some sample values:

```
WorkerTasks = {
<"GWashington" "Plumbing">
<"GWashington" "Carpentry">
<"NBonaparte" "Carpentry">
<"NBonaparte" "Painting">
<"WChurchill" "Plumbing">
<"WChurchill" "Painting">
};
```

Again, the `WorkerTasks` set only contains an element if it is possible to assign a worker to a task. There is no `<"GWashington", "Painting">` pair because the worker `GWashington` is not allowed to do the painting task. The array of decision variables `x` is indexed over the sparse set `WorkerTasks`:

```
dvar boolean x[WorkerTasks];
```

Now, the `WorkerTasks` set is used in constraints that assign tasks to workers and workers to tasks:

```
forall (t in Tasks)
    assignTasks: sum (<w,t> in WorkerTasks) x[<w,t>] == 1;
```

```
forall (w in Workers)
    assignWorkers: sum (<w,t> in WorkerTasks) x[<w,t>] == 1;
```

The first constraint (`assignTasks`) ensures that every task is covered: for each task, it sums over the workers who are capable of performing the task. The second constraint (`assignWorkers`) ensures that workers are assigned to exactly one task: for each worker, it sums over the tasks that worker is capable of performing.

These constraints are written using *slicing* in OPL, which streamlines iteration over sparse sets. Instead of iterating over *all* the pairs of workers and tasks, this code sample only iterates over the valid pairs specified in the sparse set `WorkerTasks`. Through slicing, the `assignTasks` constraint finds the matching `<w,t>` pair for each `t` in `Tasks`, and the `assignWorkers` constraint finds the matching `<w,t>` pair for each `w` in `Workers`. The decision variable declaration saves memory by only creating decision variables `x` for valid pairs of workers and tasks, while the constraint declarations save time and memory by only iterating over valid pairs of workers and tasks.

Now, suppose it is necessary to model the assignment of workers to tasks on specific days. Each task requires one worker for one day, and each task has a set of days when it can be performed. Each worker can do at most one task per day, and each worker has a set of days when he or she is available to work. This requires two more tuple sets for the worker-day and task-day pairs:

```
tuple workDay {
    string  workerID;
    int     dayID;
}
tuple taskDay {
    string  taskID;
    int     dayID;
}
setof(workDay) WorkerDays = ...;
setof(taskDay) TaskDays = ...;
```

The set `WorkerDays` has elements with two fields. The first field corresponds to a worker, while the second field corresponds to a day. An element of the set indicates that the worker is available to work on that day. Similarly, the set `TaskDays` represents the eligible task/day combinations. Sample values of `WorkerDays` and `TaskDays` look like the following:

```
WorkerDays = {
    <"GWashington" 2>      <"GWashington" 3>
    <"GWashington" 5>      <"NBonaparte" 1>
    <"NBonaparte" 2>       <"NBonaparte" 3>
    <"NBonaparte" 4>       <"WChurchill" 2>
    <"WChurchill" 4>       <"WChurchill" 5>
    <"WChurchill" 6>
};
TaskDays = {
    <"Carpentry" 1>        <"Carpentry" 2>
    <"Carpentry" 3>        <"Plumbing" 3>
    <"Plumbing" 4>         <"Plumbing" 5>
    <"Painting" 5>
};
```

It is also necessary to define a tuple to link the tasks, workers and days:

```
tuple workTaskDay {
    string  workerID;
```

```

    string    taskID;
    int       dayID;
}
setof(workTaskDay) WorkerTaskDays = {<w,t,d> |
    <w,t> in WorkerTasks, <w,d> in WorkerDays,
    <t,d> in TaskDays };

```

Using the earlier sample data for WorkerTasks, WorkerDays and TaskDays, the values of the WorkerTaskDays set will be:

```

WorkerTaskDays = {
<"GWashington" "Plumbing" 3>
<"GWashington" "Plumbing" 5>
<"GWashington" "Carpentry" 2>
<"GWashington" "Carpentry" 3>
<"NBonaparte" "Carpentry" 1>
<"NBonaparte" "Carpentry" 2>
<"NBonaparte" "Carpentry" 3>
<"WChurchill" "Plumbing" 4>
<"WChurchill" "Plumbing" 5>
<"WChurchill" "Painting" 5>
};

```

By initializing WorkerTaskDays in terms of the three sparse sets WorkerTasks, WorkerDays and TaskDays, WorkerTaskDays represents only the valid combinations of workers, tasks and days. In other words, there is an element <w, t, d> in WorkerTaskDays if worker w is qualified to perform task t, if task t is required on day d, and if worker w is available to work on day d. Note that slicing is used to efficiently match the valid pairs when initializing the set WorkerTaskDays. The array of decision variables x are indexed over the sparse set WorkerTaskDays and declared as:

```
dvar boolean x[WorkerTaskDays];
```

The two assignment constraints are written as:

```

forall (<t,d> in TaskDays)
    assignTasks:
        sum (<w,t,d> in WorkerTaskDays) x[<w,t,d>] == 1;

forall (<w,d> in WorkerDays)
    assignWorkers:
        sum (<w,t,d> in WorkerTaskDays) x[<w,t,d>] <= 1;

```

The first constraint (assignTasks) ensures that every task is covered: for each day d that a task t is required, it sums over the workers w who are capable of performing that task. The second constraint (assignWorkers) ensures that workers are assigned to at most one task per day: for each day d that a worker w is available, it sums over the tasks t that worker w is capable of performing.

Finally, tuple keys can also help with efficient storage and iteration over sparse data. Tuple keys were introduced in OPL 5.0 to indicate the fields that uniquely identify each element in a tuple set. Tuple keys are similar to primary keys in a relational database. Slicing is more efficient with tuple keys, and tuple keys can reduce the need for array data. For instance, keys can be used with tuple sets that represent the workers:

```

tuple worker {
    key string    workerID;
    string       officeID;
    float        salary;
}

```

```
setof(worker) Workers = ...;
```

The values of the `Workers` set are initialized as normal:

```
Workers = {  
<"GWashington" "Virginia" 1000>  
<"NBonaparte" "Paris" 1100>  
<"WChurchill" "London" 1300>  
};
```

Here, the field `workerID` identifies each worker. By designating `workerID` as a key, OPL can efficiently search the set of workers in order to retrieve the associated `officeID` and salary values. OPL also verifies that there are no duplicate elements for a particular key.

These simple examples illustrate the power of sparse data in OPL. Tuple sets make it easy to define data in terms of valid combinations only, saving considerable memory. Inside `forall` and `set` operations, tuple sets are used to iterate over just the matching combinations, saving considerable time and memory.

Efficient data initialization

In many cases, OPL input data is generated via arithmetic or other calculations. For instance, in a staff planning model, the number of workers available each week may be determined by adding the available hours for each worker. In a manufacturing planning model, the manufacturing costs may be calculated by multiplying the hours to make each product by the average hourly wage for a factory worker. In a transportation model, the transportation costs may be calculated by finding the shortest paths between pairs of cities.

These calculated inputs can be generated by an OPL model or an external system. Outside the OPL model, virtually any database or programming language can be used to compute data values. With a database, both SQL queries and calls to stored procedures can be embedded inside an OPL `.dat` file. OPL tuple sets can then be initialized with the database values. The mapping between databases and OPL data structures is illustrated in the ***OPL IDE Tutorial*** in the section **Tutorial: Working with the External Data > The Oil Database Example**. Using a programming language, the computed data can be formatted and saved as an OPL `.dat` file. This is very useful in applications where the OPL Interfaces are not used, such as a command-line environment. When the model is embedded in a C++, Java or .NET application, the OPL Interfaces provide three ways to push data directly to OPL without writing intermediate files. First, the OPL Interfaces contain methods that read OPL-formatted data as a large string or from an external file. Second, the Interfaces provide `set()` functions to set or modify OPL data values, which is illustrated in the "`mulprod_main.prj`" example in the "`examples\opl`" subdirectory of OPL. Third, the class `IloCustomOplDataSource` can be extended to implement a custom OPL data source, as illustrated in the ***OPL Interfaces User's Manual*** under **Working with the OPL Interfaces > Custom Data Sources**.

Inside the OPL model, there are two ways to compute data: inline initialization and ILOG Script. Inline initialization is generally faster, but some types of data cannot be computed via inline initialization. Inline initialization performs data initialization in the same line of code as data declaration. For example, the following code declares the integer data `n` and assigns it the value 10:

```
int n = 10;
```

More important, set and array values can be set via inline initialization. For instance, the following code initializes a set called `Pos`, then creates and initializes an array called `c`, which is indexed by `Pos`:

```
setof(int) Pos = { i | i in Index : x[i] > 0.0 };  
float c[i in Pos] = i*i;
```

Note that the set `Pos` contains the indices matching the condition `x[i] > 0`. Although inline initialization is limited to a single line of code, OPL provides a powerful set of functions and aggregate operators that can compute most initial data values. For instance, the `max1` function takes the maximum value of a fixed list, which can be used to set an array equal to only positive values:

```
float cost[i in S] = max1(0, val[i]);
```

When building a network flow model, the conditional operator “`? :`” can be used to determine flow conservation at the nodes:

```
float flow[i in Nodes] =
    (i == src) ? -total : ((i == dst) ? total : 0.0);
```

This assigns the value of `-total` at the source, `total` at the destination and zero elsewhere.

One special case of inline initialization is *generic indexed array initialization*, in which index values and array values are specified jointly. It is primarily used to efficiently initialize array values from a tuple set, particularly when converting database records to OPL arrays. For example, the tuple set `Workers` provides basic information about each worker:

```
tuple worker {
    key string    workerID;
    string       officeID;
    float        salary;
}
setof(worker) Workers = ...;
```

Generic indexed array initialization is the most efficient way to convert the salary tuple values into an array of weekly pay:

```
float weekPay[Workers] = [ w : w.salary/52 | w in Workers ];
```

The expression `w:w.salary/52` states that the index is specified by `w` and the value is `w.salary/52`. The array is computed efficiently: OPL considers each worker `w` in the set `Workers`, and uses those values to determine how to fill the array. Unlike the salary field of the worker tuple, the values of `weekPay` can be modified using ILOG Script:

```
execute updatePay {
    for (var i in Workers)
        weekPay[i] = 1.05 * weekPay[i];
}
```

For a complete explanation of generic indexed array initialization, see the [OPL, the Modeling Language > Data Sources > Data Initialization > Initializing Arrays](#) section in the *OPL Language Reference Manual*.

The secret to success with inline data initialization is to utilize the rich set of OPL functions and operators. Here is a table of useful functions and operators for inline data initialization:

Function or operator	Meaning
<code>cond ? trueExpr : falseExpr</code>	Conditional operator: returns <code>trueExpr</code> if the condition <code>cond</code> is satisfied and <code>falseExpr</code> otherwise
<code>abs()</code>	Absolute value
<code>min1(value1, value2, ...)</code>	Minimum over a fixed list of values
<code>max1(value1, value2, ...)</code>	Maximum over a fixed list of values

<code>min (conditions) expr</code>	Minimum of <code>expr</code> over a set of conditions
<code>max (conditions) expr</code>	Maximum of <code>expr</code> over a set of conditions
<code>sum (conditions) expr</code>	Sum of <code>expr</code> over a set of conditions
<code>prod (conditions) expr</code>	Product of <code>expr</code> over a set of conditions

With these functions, most types of data computation can be done via inline initialization. However, when something more powerful is needed, the ILOG Script language is also available for data processing. ILOG Script for OPL is a standard implementation of JavaScript that conforms to the ECMA-262 specification. As a full-fledged scripting language, ILOG Script supports loops, recursion and custom functions, so it can compute virtually any kind of input data.

Often, the best practice is to define initial values through inline initialization and then reset individual values through ILOG Script. For example, the earlier example of flow conservation data could be written as follows:

```
float flow[i in Nodes] = 0;
execute setFlowConservation {
    flow[src] = -total;
    flow[dst] = total;
}
```

Here, inline initialization is used to set the flow to zero for all nodes, and then a script block is used to change the flow value for the source and destination elements. As a simple example of recursion, a set of Fibonacci numbers can be computed via a script block:

```
range rng = 1..10;
int fib[i in rng] = 1;
execute initFib {
    for (var i in rng)
        if (i > 2)
            fib[i] = fib[i-1]+fib[i-2];
}
```

Again, inline initialization is used to set the initial values of `fib`, and then ILOG Script is used to set the Fibonacci values for `i > 2`. Although this is a simple example, the same principles apply when using iteration or recursion to calculate data such as shortest paths or spanning trees.

In summary, there are multiple ways to calculate input data for an OPL model, each with specific advantages. Anything is possible with external data initialization, though it makes the OPL model dependent on external code. Also, when passing data to OPL, the amount of input data may impact the total requirements for memory or disk space.

As for internal data initialization, ILOG Script is interpreted and generally runs slower than inline initialization. It is better to use inline initialization whenever possible. ILOG Script should be used only to override a small number of values or when the calculation requires complex programming logic such as looping or recursion that cannot be implemented via inline initialization.

Models often use multiple types of data initialization. Basic data should be read from external data sources. Inline initialization should be used for most data computation, and ILOG Script should be used for complex calculations that cannot be done via inline initialization.

Finally, when solving an iterative sequence of models via ILOG Script, care should be taken when creating new objects. ILOG Script does not have an automatic garbage collector, so models and data are retained by default. However, models and data can be freed via the `end()` and `endAll()` methods. Detailed information about these methods can be found in the ***OPL Interfaces User's Manual*** under **Working with the OPL Interfaces > Memory Management**.

An example of this can be found in the “**mulprod_main.mod**” example located in the “**examples\opl**” subdirectory of OPL. Also, the OPL Interfaces manage memory more efficiently than the OPL IDE, so the OPL Interfaces or the oplrun command-line utility may also reduce memory consumption for complex scripts.

Efficient model construction

Efficient data structures and initialization methods are not enough to guarantee good performance for OPL and CPLEX. How a model is designed can also have a large impact on both time and memory. In linear programming, the complexity of a model depends in large part on the number of nonzero coefficients. In fact, the number of nonzero coefficients also affects the memory and setup time for OPL. This section explores ways to compress the model size by eliminating repeated expressions.

The primary candidates for model compression are summations of similar terms. Many models use summations to compute various subtotals, which may be used in the objective function or require upper or lower limits. When a model contains multiple kinds of subtotals, there may be an opportunity to combine similar expressions and save memory and computation time. Returning to the example of assigning workers to tasks over a multi-day period, suppose there is a constraint limiting each worker to five days per week and 18 days per four-week period. One way to write this in OPL would be:

```
forall (w in Workers, m in Mondays)
    weekLimit:
        sum (d in m..m+6, <w,t,d> in WorkerTaskDays)
            x[<w,t,d>] <= 5;

forall (w in Workers, m in Mondays)
    monthLimit:
        sum (d in m..m+27, <w,t,d> in WorkerTaskDays)
            x[<w,t,d>] <= 18;
```

The sum operators first iterate over the days, and then over the sparse set `WorkerTaskDays`. However, note that many of the terms in the `monthLimit` sum are the same as in the `weekLimit` sum. The `weekLimit` constraint is rewritten in terms of a new decision variable called `weekTotal`:

```
dvar int weekTotal[Workers,Mondays] in 0..5;

// ...

forall (w in Workers, i in Mondays)
    weekLimit:
        sum (d in i..i+6, <w,t,d> in WorkerTaskDays)
            x[<w,t,d>] == weekTotal[w,i];

forall (w in Workers, i in Mondays)
    monthLimit:
        sum (j in 0..3) weekTotal[w,i+7*j] <= 18;
```

Here, an array of decision variables `weekTotal` is defined to represent the total work done in a week. An upper bound of 5 is set for each of the decision variables in the `weekTotal` array to limit the total work in a single week. Finally, the monthly limit is defined in terms of the `weekTotal` variables.

To compare the impact of the two formulations, suppose there are T tasks, W workers and K weeks. In the `weekLimit` constraint, the first formulation contains up to $W \cdot K \cdot 7 \cdot T$ nonzeros, while the second formulation contains at most $W \cdot K \cdot 7 \cdot (T+1)$ nonzeros. In the `monthLimit`

constraint, the first formulation contains up to $28*W*K*T$ nonzeros, while the second formulation contains at most $4*W*K$ nonzeros. Despite the $W*K$ new variables, the second formulation poses a huge reduction in memory and computational complexity, and the savings increase as the problem size increases.

Models with time window constraints are another target for reusing similar expressions. In the worker assignment model, suppose the weekly time limit must be respected for every seven-day period, not just weeks from Monday through Sunday. An OPL model might declare the constraint as:

```
forall (w in Workers, i in Days)
  weekLimit:
    sum (d in i..i+6, <w,t,d> in WorkerTaskDays)
      x[<w,t,d>] <= 5;
```

To compress this constraint, *telescoping sums* are used to express the time windows recursively in the model:

```
dvar int weekTotal[Workers,Days] in 0..5;

// ...

forall (w in Workers)
  firstWeek:
    sum (d in 1..7, <w,t,d> in WorkerTaskDays)
      x[<w,t,d>] == weekTotal[w,1];

forall (w in Workers, i in Days : i > 1)
  weekLimit:
    weekTotal[w,i] == weekTotal[w,i-1]
      - sum (<w,t,i-1> in WorkerTaskDays) x[<w,t,i-1>]
      + sum (<w,t,i+6> in WorkerTaskDays) x[<w,t,i+6>];
```

Like earlier, a new variable `weekTotal` is defined to represent the subtotal of the work performed in each seven-day window. The `firstWeek` constraint defines the value `weekTotal` for days 1 through 7, and then the `weekLimit` constraint uses a recursive expression to state the relationship between values of `weekTotal` for adjacent days. The key idea is that the `weekLimit` constraint subtracts the contribution from the prior day and adds the contribution for the end of the week. Each `weekLimit` constraint saves space and time by not having to iterate over the days $i+1$ through $i+5$.

It is worth noting that these types of model substitutions are valuable even if CPLEX presolve may expand the expressions internally. The reason is that these substitutions avoid having to create and iterate over objects inside OPL and ILOG Concert Technology, while CPLEX presolve performs its transformations on the sparse matrix data, which is extremely compact.

Quadratic programming models present a special opportunity for compressing quadratic terms. The canonical form for a quadratic objective is $c^T x + \frac{1}{2} x^T Q x$, where Q is the symmetric quadratic matrix. In many applications such as portfolio optimization, Q is large but it has been derived by computing the product of a smaller matrix and its transpose, i.e. $Q = L^T L$. In this case, the model can be reformulated in terms of the original L matrix by introducing auxiliary variables y . The constraints $y = Lx$ are added and the objective is changed to $c^T x + \frac{1}{2} y^T y$. Despite the additional variables and constraints, this transformation simplifies the problem by creating a quadratic matrix that is small and sparse. These same transformation principles can be applied to quadratic constraints.

Finally, linear programming decomposition can also help solve particularly difficult optimization models. Instead of solving one large problem, the problem is divided into one or more subproblems plus a master problem that links the subproblems. The decomposition process

iterates between solving the master problem and subproblems. The solution to the master problem becomes an input to the subproblems, and the solutions to the subproblems become refinements to the master problem. If the subproblems are easy to solve, decomposition can significantly reduce the time and memory required to solve the overall optimization problem. In some cases, decomposition can also tighten the linear programming (LP) relaxation of a mixed-integer programming model. A full explanation of LP decomposition is beyond the scope of this paper, but it is described widely in optimization literature. With OPL, a decomposition method can be implemented via ILOG Script. The script controls the decomposition process: solving the master problem, updating the subproblems with the solution from the master problem, solving the subproblems, updating the master problem with the solutions from the subproblems, and repeating as necessary. ILOG OPL-CPLEX Development System provides an example of column generation via the classical cutting stock model. It is covered in the projects “`cutstock_main.prj`” and “`cutstock_int_main.prj`” in the “`examples\opl`” subdirectory of OPL.

Conclusion

These techniques reduce the overhead of ILOG OPL-CPLEX Development System, enabling it to solve large-scale optimization models. When tuning model performance, it is important to pay attention to both running time and memory use since they are related. The Profiler can be used to identify the bottlenecks in model and data initialization.

The tuple set in OPL is a powerful data structure for representing sparse model data. With tuple sets, it is important to create data only for valid combinations and use slicing to iterate efficiently over these combinations. Tuple sets save considerable memory and processing time, and are the key to solving large-scale models. Virtually every model can benefit from tuple sets because nearly every application has some form of valid and invalid combinations of alternatives.

The general techniques presented in this white paper can be applied with any optimization modeling interface. However, the beauty of ILOG OPL-CPLEX Development System is that these techniques are easy to write in just a few lines of OPL code. Also, the model logic is clear and simple in OPL, making it easy to identify opportunities to tune the model formulation.