High Performance Prototyping of Decomposition Methods in GAMS

Timo Lohmann
Division of Economics and Business, Colorado School of Mines, Golden, USA, tlohmann@mines.edu

Michael R. Bussieck
GAMS Development Corp., Washington, DC, USA, mbussieck@gams.com

Lutz Westermann
GAMS Software GmbH, Braunschweig, Germany, lwestermann@gams.com

Steffen Rebennack
Division of Economics and Business, Colorado School of Mines, Golden, USA, srebenna@mines.edu

Prototyping algorithms in algebraic modeling languages has a long tradition. Despite the convenient prototyping platform that modeling languages offer, they are typically seen as rather inefficient with regard to repeatedly solving mathematical programming problems, a concept that many algorithms are based on. The most prominent examples of such algorithms are decomposition methods, such as Benders decomposition, column generation and Dantzig-Wolfe decomposition. In this work, we discuss the underlying reasons for repeated solve deficiency with regard to speed in detail and provide an insider’s look into the algebraic modeling language GAMS. Further, we present recently added features in GAMS that mitigate some of the efficiency drawbacks inherent to the way modeling languages represent model data and ultimately solve a model. In particular, we demonstrate the grid-enabled Gather-Update-Solve-Scatter facility and the GAMS object oriented application programming interface (API) on a large scale case study that involves a Bender decomposition type algorithm.

Key words: Algebraic modeling languages, GAMS, data structures, parallel computing, Benders decomposition, expansion planning.

1. Introduction.

Algebraic modeling languages (AMLs), such as AIMMS (Bisschop and Roelofs 1999), AMPL (Fourer et al. 2003), GAMS (GAMS Development Corporation 2014), or LINGO (Schrage 2006) are widely accepted for their ease of implementation and the way they represent and maintain mathematical programming problems (Kallrath 2004). The AML’s very idea is to represent the problem in their algebraic form and closely follow its mathematical notation, while allowing the separation of model, data, and solution algorithm. Modeling systems do not solve the problem, they communicate it to the solver of choice. They can therefore be seen as interfaces between the user and the mathematical programming solvers. Modeling systems are highly efficient when used to solve a monolithic model (i.e., the solver is called only once). However, if the solver is called repeatedly, commu-
communication effort between modeling system and solver comes at the cost of computational speed.

Many practical optimization problems cannot be solved efficiently as a monolithic model. Decomposition approaches are often the method of choice in these cases, because they decompose the problem into a series of smaller problems by exploiting its structure (Conejo et al. 2006). Examples are Benders decomposition, column generation and Dantzig-Wolfe decomposition (Conejo 2006), and polylithic approaches in general (Kallrath 2011).

Although modeling systems may lack efficiency when repeatedly solving models, they do offer a convenient prototyping environment. For instance, GAMS, with its mix of imperative and declarative elements, has a long tradition of being used for a prototype implementation of solution algorithms. Solvers like BARON (Tawarmalani and Sahinidis 2005), Dicopt (Duran and Grossmann 1986) or SBB (Bussieck and Drud 2001) were first implemented in GAMS directly. Similarly, AIMMS’s outer approximation algorithm for mixed-integer nonlinear programming problems (Hunting 2011) is an implementation of Duran and Grossmann (1986) and AMPL offers prototyped decomposition algorithms on its website\(^1\). We would argue that recent developments in those AMLs have mitigated some of their mentioned efficiency deficits. Table 1 lists the most important GAMS features supporting the efficient implementation of such algorithms. This table serves as a timeline in the following discussion.

\(^1\)http://www.ampl.com/NEW/LOOP2/index.html

<table>
<thead>
<tr>
<th>Version</th>
<th>Year</th>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.25†</td>
<td>1992</td>
<td>Put Facility, if/else</td>
<td>Writing of external ASCII files, execution time if/else structures</td>
</tr>
<tr>
<td>2.50†</td>
<td>1995</td>
<td>for/while/repeat/until, execute</td>
<td>More looping constructs and the ability to execute external programs during GAMS run-time</td>
</tr>
<tr>
<td>2.50†</td>
<td>1996</td>
<td>External Equations</td>
<td>Public API to connect external user constraints to GAMS and solvers</td>
</tr>
<tr>
<td>19.4</td>
<td>2000</td>
<td>GDX</td>
<td>GAMS Data eXchange to conveniently exchange data between GAMS and other programs</td>
</tr>
<tr>
<td>20.1</td>
<td>2001</td>
<td>CONVERT</td>
<td>Conversion of model instance into different formats</td>
</tr>
<tr>
<td>22.3</td>
<td>2006</td>
<td>Grid Facility</td>
<td>Facilitates parallel execution of GAMS solve statements</td>
</tr>
<tr>
<td>‡</td>
<td>2007</td>
<td>GMO/GAMSLinks</td>
<td>Public API to build solver links for GAMS model instances</td>
</tr>
<tr>
<td>22.7</td>
<td>2008</td>
<td>Solvelink=5</td>
<td>Facilitates in-core communication with the solver without temporary files</td>
</tr>
<tr>
<td>22.7</td>
<td>2008</td>
<td>GDX API</td>
<td>Published the expert level GDX API</td>
</tr>
<tr>
<td>22.9</td>
<td>2008</td>
<td>Scenario Solver/GUSS</td>
<td>Solves collection of data-related models, applying advanced solution information</td>
</tr>
<tr>
<td>23.7</td>
<td>2011</td>
<td>Asynchronous Execution</td>
<td>Allows asynchronous job handling, i.e., parallel execution of GAMS and other jobs</td>
</tr>
<tr>
<td>23.7</td>
<td>2011</td>
<td>Extrinsic Functions</td>
<td>Public API to connect user functions to GAMS and solvers</td>
</tr>
<tr>
<td>23.9</td>
<td>2012</td>
<td>OO API</td>
<td>Object Oriented API to GAMS</td>
</tr>
<tr>
<td>24.3</td>
<td>2014</td>
<td>Grid-enabled GUSS</td>
<td>Solves collection of data-related models, applying advanced solution information, in parallel fashion</td>
</tr>
<tr>
<td>24.3</td>
<td>2014</td>
<td>IDX API</td>
<td>Simplified GDX API for imperative programming languages</td>
</tr>
</tbody>
</table>

Table 1 GAMS features supporting efficient implementation of algorithms.
This paper is built around a case study of a long-term generation expansion planning problem and its algorithmic challenges. The problem is to find an optimal here-and-now investment decision into new power plants for an existing power system. The planning horizon spans more than two decades, from 2008 to 2030. We apply a Benders decomposition type algorithm, first presented in Lohmann and Rebennack (2014), to solve this problem. The main idea of the algorithm is to divide the monolithic model into a master problem and a subproblem, connected by so-called complicating variables. By iterating between these two problems, the algorithm builds an outer approximation, representing the subproblem with a finite number of hyperplanes in the master problem (Benders 1962). The algorithm poses strong requirements on the modeling system. We therefore begin by highlighting two recently added GAMS features whose development was strongly motivated by this case study. Their development challenges are described both from a GAMS standpoint and from a broader modeling systems perspective.

The first challenge is the size of the discussed model, which requires efficient handling of sparse large-scale multi-dimensional data sets. The largest instance of the underlying monolithic model that we solve has 89 million variables, 56 million constraints, and 180 million nonzeros. We show how an imperative programming language, in this case C#, can be seamlessly integrated into the algorithm in GAMS, thus combining the best of two worlds. While all model solves are still handled by GAMS, certain computational components are turned over to C#. To be efficient, the interface transferring data from the modeling language to the programming language must be able to manage large data sets like this. Modeling languages typically represent data similar to relational data bases (i.e., only nonzero records are stored with their labels in a sparse fashion), whereas imperative programming languages use specialized data structures and an indexed data space (i.e., records are densely stored to work with index arithmetics). The conversion from one space into the other has the potential to become a bottleneck when large-scale data sets are involved.

The second challenge is that the decomposition approach relies on efficiently solving a series of linear programming (LP) problems, preferably in a parallel fashion. In this context, modeling systems are typically seen as slow due to the effort required to generate a model instance repeatedly and the inability to apply advanced solution information from a previous instance solve. A model instance is the instantiation of an algebraic model with specific data. Bussieck et al. (2011) demonstrate how GAMS’s Gather-Update-Solve-Scatter (GUSS) facility can overcome these challenges in certain cases. We demonstrate the natural extension of GUSS into a parallel environment by using the GAMS Grid Computing Facility (Bussieck et al. 2009). We also provide background on how algebraic modeling languages typically represent model instances, as compared to a solver representation, and emphasize the development effort going into “hybrid” methods such as GUSS.

This paper contributes to the field in three major ways:

- First, we provide an insider’s look into the underlying data management methods of an algebraic modeling language and discuss how an imperative programming language, in this case C#, and GAMS can be efficiently interfaced in four different ways.
- Second, we describe how GAMS’s new grid-enabled GUSS facility successfully mitigates common efficiency deficits in repeated solver calls in AMLs with regard to model instance representation and updates, bridging the gap between the model representation of a solver and a modeling language.
- Third, we test these features using a large scale case study that challenges their practical feasibility and demonstrate their viability in such a setting.
The paper is organized as follows. Section 2 describes the solution process of a model inside an algebraic modeling language, relates challenges from a conceptual standpoint, and presents two recent GAMS features to tackle these. Section 3 discusses how data can be transferred between a modeling language and an imperative programming language using new GAMS features. Following this, we apply two selected features to a power generation expansion problem solved by a Benders decomposition type approach in Section 4. Lastly, we discuss computational results in Section 5 before we conclude in Section 6.

For ease of presentation, we use verbatim font throughout this paper when referring to a keyword or item in a specific modeling or programming language.


We begin with the history and concepts of AMLs and compare them to modern solver interfaces. In particular, we focus on the concept of a model instance and how AMLs historically approach it differently than a solver interface. Highlighting the key differences helps the reader understand the development efforts going into facilities such as GUSS and GAMS’ object oriented API (OO API), which are presented later in this section. Table 1 above provides a timeline of the features discussed both in this section and in Section 3. This section contains several examples taken from CPLEX’s and Gurobi’s C++ API. We refer the reader to the respective API user manuals for a detailed description.

2.1. A Background on Algebraic Modeling Languages and Solvers

Historically, AMLs were preceded by matrix generators - custom programs that generated a (sparse, solver) matrix with dense one-dimensional row and column vectors from sparse multi-dimensional problem data and mathematical algebra. Consider an example of a one-dimensional array, assuming there are 15 data points:

\[
[0, 0, 1, 3, 0, 0, 7, 2, 0, 0, 2, 0, 0].
\]

The data points can be densely stored in an array of size 15. Or, they can be sparsely stored in two arrays of size 5 each, only representing the nonzero data points. One array stores the original indices (one-based in this example), the other stores the associated values:

\[
[3, 4, 8, 9, 13] \quad \text{and} \quad [1, 3, 7, 2, 2].
\]

Modern programming paradigms, such as object oriented programming, have helped tremendously with issues in early matrix generators implementations such as code encapsulation and memory management. Nevertheless, even modern programming languages and solver APIs like CPLEX’s Concert, or programming based modeling languages like significant parts of the Python Optimization Modeling Objects (Pyomo), suffer from the same principle problem. Each presents a model close to the solver representation (for example, Concert’s dense one-dimensional IloNumVarArray), rather than presenting the model in the sparse multi-dimensional user space.

One of the reasons AMLs can efficiently generate a (sparse, solver) matrix with dense one-dimensional row and column vectors from sparse multi-dimensional problem data and mathematical algebra is that the AMLs execute a one-off job for a particular model instance. Consequently, no data structures for model instance modifications need to be set up and maintained. All information about the model generation can be abandoned after
the generation takes place, as long as the map from the dense one-dimensional variables and constraints vectors to the sparse multi-dimensional variable and constraint symbols inside the AML is kept alive. In some AMLs the solution of a model instance, such as the solve statement in GAMS, is a very complex operator against the internal database. The primal and dual solution of the model instance is merged back into the database, similar to SQL insert and/or update statements for relational database management systems.

In the classical AML approach, therefore, a model instance does not exist as an object with access and modification methods. This is very different compared to model instance representations close to the solver, such as Lindo’s API. In this case, a model instance object is built and a set of methods to query and modify the model instance is provided. Gurobi is another example. In Gurobi’s C++ API, related constructs are GRBnewmodel (create an empty model instance), GRBaddconstrs (add constraints), GRBchgcoeffs (modify matrix coefficients) and GRBoptimize (solve model instance). The more we abstract from the particular solver representation, the more difficult it becomes to apply model instance modifications. For instance, in Gurobi’s C++ API and CPLEX’s Concert API, constraints can be specified by complex expressions, GRBLinExpr and IloExpr, respectively, but an update of the expression cannot be performed by updating the individual terms of the expression. Instead, the resulting final matrix coefficients must be updated (GRBModel::chgCoeffs(...) with Gurobi), or the solver representation from the more abstract model representation must be re-extracted (IloCplex::extract(const IloModel) with CPLEX).

Both approaches have advantages and disadvantages. On the one hand, the approach “operator against the database” allows the user to specify the model modifications in the user space. However, AMLs have not found ways to utilize previous model generation information. The work needs to be done from scratch each time the solution of a model instance is requested, which results in a significant performance hit, especially if the ratio of time solving and time generating the model instance is small. On the other hand, the “model as an object” approach can be highly efficient for modifying the model instance and resolving the problem, but the user must efficiently and correctly map the update data from the sparse multi-dimensional user space into the model instance object representation.

2.2. Grid-Enabled GUSS

The Gather-Update-Solve-Scatter (GUSS) facility introduces a representation of a model instance that provides advantages from both the “operator against the database” and the “model as an object” approaches, at the cost of some functional limitations. The main principle of GUSS is to specify possible modifiers upfront. GUSS only allows changes to named parameters appearing in the equations and lower and upper bounds used in the model definition. Sets in the model cannot be updated, which implies that the model rim cannot change (i.e., no addition or removal of rows and columns). All input updates and desired outputs are defined in the so-called GUSS dictionary before the solve statement.

We refer the reader to the GAMS documentation\(^2\) for a detailed description of GUSS and its implementation.

However, the update of the model parameters goes far beyond updating the coefficients of the constraint matrix, objective function or the right-hand side of an equation, as possible in other modeling systems. GAMS stores all necessary expressions of the constraints with

\(^2\)http://gams.com/dd/docs/solvers/guss.pdf
the model instance. Hence, a change in the constraint matrix coefficient is the result of an expression evaluation. Consider the following network example. Let the objective function contain the term

\[ d_{ij} \cdot f \cdot x_{ij}, \]

where \( x_{ij} \) represent the shipment of goods between cities \( i \) and \( j \), \( d_{ij} \) the distance between the cities, and \( f \) the freight rate. In order to determine the sensitivity of the solution with respect to \( f \), one can solve the same model for different values of \( f \). In a matrix representation of the model, one would need to calculate the coefficients of \( x_{ij} \) which are \( d_{ij} \cdot f \), but with GUSS it is sufficient to supply different values for \( f \) that potentially result in many modified coefficients on the matrix level. The evaluation of the shipping cost term and the communication of the resulting matrix coefficient to the solver are reliably done by GUSS behind the scenes.

GUSS communicates the changes between the current model instance and the one stored in the solver only for those solvers that provide an API to manipulate a model instance inside them. Examples are CPLEX, Gurobi and Xpress. This reduces the amount of data communicated to the solver and also, in the case of an LP model, allows the solver to restart from an advanced basis and its factorization.

Internally, GUSS replaces all model parameters associated with updates defined by the user with variables. The reason is that the GAMS execution system, when generating a model, folds numerical constants in an expression to a single numeric constant. Some of the constants, however, need to become symbolic constants which remain part of the expression so that they can be reevaluate with different values. Instead of introducing a new symbol type for the modifiable parameters, GAMS introduces a “shadow” variable for every modifiable parameter to prevent this elimination/folding. This approach can cause a linear model to produce some nonlinear instructions. Regarding the example above, \( d_{ij} \cdot f \cdot x_{ij} \) becomes a nonlinear expression since \( f \) is replaced with a variable in the model instance given to GUSS. All newly introduced variables are fixed to their update values before the model instance is solved. Hence, an LP with model instance updates is still solved as an LP.

Starting with GAMS 24.3, GUSS and the Grid Facility can be used in union to get the combined benefit from both approaches, representing a natural extension of GUSS’s capabilities. The GAMS Grid Computing Facility allows the user to take advantage of multiple CPU and distributed computing environments, and the \texttt{solvelink} feature enables this. The process is then divided into two parts. First, a submission loop generates and submits independent model instances to the solver. A job handle is created for each submitted model and can be used to obtain information about its status, \textit{e.g.}, if it is still running or finished. A collection loop then gathers the previously submitted instances as soon as a solution is available, accessing the respective job handle. Again, we refer the reader to Bussieck et al. (2009) and the GAMS documentation.

The restrictions discussed above remain unchanged when using GUSS in a grid environment. The same dictionary can be used, the key difference being to specify the use of the Grid Computing Facility via the \texttt{solvelink} feature. Thus, applying a grid-enabled GUSS is straightforward for users experienced with both GUSS and the Grid Computing Facility:

\begin{verbatim}
1   MyModel.SolveLink=SolveLink.AsyncGrid;
2   loop(job, //Submission
3       scen(s) = threadS(job,s);
4   Solve MyModel using LP max MyObjective scenario MyDict;
\end{verbatim}
Line 1 invokes the Grid Computing Facility. The two loops carry out the submission and collection as described above. Inside the submission loop, on line 4, GUSS is invoked using the keyword `scenario` and specifying the dictionary. The previous line activates the subset of scenarios affiliated with the current thread. This methodology is not restricted to LPs, but can be applied to all existing model types within the GAMS environment. The entire GUSS functionality is maintained within the grid, although some features, such as more complex updating schemes, require precaution when assigning scenarios to jobs.

Note that the realized speed-up from the use of multiple cores might not be as high as anticipated due to several reasons. First, the different jobs may be sharing the same resource. When all processes get the data from scenario files and share common resources such as network, disk and bus speed, possible bottlenecks can result. Second, and even more importantly, the work per scenario may not be uniform, leading to idle threads while the collection loop waits for all jobs to finish. This occurs when the parameter updates are assigned apriori to threads. Producer-consumer systems guarantee a perfect parallelization, but we lose control over the sequence of jobs assigned to each thread. Assigning a fixed sequence of jobs may be a favorable way to exploit advanced solution information for LPs and other problem classes. We provide an example in our case study in Section 5. Overall, however, it is challenging to find a good scenario batch assignment that yields uniform return times.

GUSS was developed particularly to solve many scenarios of the same model. Its utility has been demonstrated on a number of example applications and has shown significant improvements in the speed of processing collections of models. Bussieck et al. (2011) report a speed-up of factor five compared to a “traditional” GAMS implementation and less than factor two compared to the faster, but solver dependent, CPLEX Concert technology. Besides a modifiable model instance, GUSS manages the scenario data input and output. All scenario data has to be known before the GUSS processing. Dynamic scenario data updates that depend on the solution of a previous scenario are therefore not possible. In the next section, we discuss the GAMS OO API which, among other things, overcomes this limitation and allows the user to investigate the issues and potentials of a model instance independent of GUSS.

2.3. Object Oriented APIs for Modeling Systems

APIs facilitate communication and take advantage of two worlds, in GAMS’s case a modeling system and imperative programming environments, e.g., C++, Java, or Python. Imperative programming environments are usually seen as superior for implementing decomposition algorithms due to their ability to update a model instance instead of regenerating it over and over, and their application of advanced solution information from a previous

```plaintext
5  handle(job) = MyModel.handle;
6  );
7  repeat //Collection
8    loop(job$handlecollect(handle(job)),
9      display$handledelete(handle(job)) 'trouble deleting handles' ;
10    handle(job) = 0;
11  );
12  display$sleep(card(job)*0.2) 'was sleeping for some time';
13  until card(handle) = 0;$
```

Listing 1: Grid-enabled GUSS Example.
solve. In general, imperative languages offer general data structures which are more flexible than the constructs found in GAMS and other AMLs.

In distribution 22.3, GAMS first introduced an API that allowed the interaction with GAMS from within different programming languages like C/C++, or Java. More recently, in distribution 23.9, GAMS introduced an OO API that extended this framework to C#, Java, and Python, and also provided new interaction tools. The GAMSModelInstance class, as part of the GAMS OO API, provides a controlled way of modifying a model instance and solving the resulting problem in the most efficient way, by communicating only model changes to the solver and performing a hot start (in case of a continuous model like LP) without the use of disk IO. While GAMSModelInstance uses the same technology as GUSS does with the same limitations, most importantly the fixed model rim (cf. Section 2.2), it also allows more flexible interaction with the model instance, mitigating some of GUSS’s drawbacks.

Without going into the details of the OO API, a brief description of the GAMSModelInstance class is provided below. An instance of this class is generated by some solve statement implicitly executed by some methods of the class. As with GUSS, a list of modifiers must be known and specified before the class is instantiated. Variable bounds, level and marginal values can be provided as modifiers. This is of particular interest for NLP starting points. Unlike GUSS, the user does not need to specify in advance those variable and equation symbols of interest for output, since the entire solution can be selectively queried after the solve method has been called.

The actual modification data, i.e., values of the parameters in the list of modifiers, is stored in a synchronization database. The solve method iterates through all records of all modifiers in the model instance and tries to find corresponding update data in the synchronization database. If a record is found in this database, this data record is copied into the model instance. If no corresponding record is found, there are different choices identical to update type choices in GUSS:

- **UpdateType=BaseCase:** the original data record is restored.
- **UpdateType=Zero:** the default record of zero is applied.
- **UpdateType.Accumulate:** no copies made; previously copied record values are used.

Once the model instance has been updated, the model is passed to the selected solver. After the completion of the solve method, the synchronization database will contain the primal and dual solution of the model just solved. The modifier parameters are also accessible. Their duals provide sensitivity information about the parameter setting because they correspond to variables in the model instance.

### 3. Relational versus Indexed Data Space.

In the previous section we discussed two recent GAMS developments that combine the advantages of a model’s representation by a solver and an algebraic modeling language. The user of an algebraic modeling language prefers to reuse most of the original model formulation as well as data when prototyping algorithms with advanced model instance representations, discussed in Sections 2.2 and 2.3. While GUSS allows the user to do this, the OO API requires a transfer of data into the programming language environment. This can be challenging, because most modeling systems represent data in a different fashion than these environments. In some cases, it can be advantageous to interface the modeling language with a programming language, even if all model solves are still handled by the former. This is because some computations cannot be performed efficiently in a modeling
system. To understand why this is so, we now discuss the data representation differences, and compare three currently available interfaces in C# that facilitate the transfer between the two environments.

Data is stored sparse in GAMS, which means that GAMS only stores nonzero values together with the necessary set labels (see Section 2.1). This is true for both zeros defined by the user and zeros created during execution. The data structures inside GAMS are designed to take advantage of the sparse storage of data when executing assignment statements and generating a mathematical programming model. Sparse storage of data is an important design feature that allows GAMS to work with large, multi-dimensional sets and parameters, as long as the total amount of data is limited. It distinguishes GAMS from regular languages like Fortran, C, Java, or C# in which arrays are allocated dense, \textit{i.e.}, space for all combinations of all indices exists. The dense storage scheme has the advantage that, for a set of indices, the location of the data element can be computed using simple index-arithmetic and accessed directly. GAMS tries to automatically adjust data structures to perform calculations in the most efficient way, but one can always find examples where programs using specialized data structures significantly improve performance. In such situations the exchange of GAMS data to specialized data structures in regular programming languages and back is of importance.

Over the past few years, GAMS has introduced tools and libraries that focus on data exchange. The most important step towards GAMS data connectivity was GDX (Gams Data eXchange, introduced in version 19.4 in August 2000, see Table 1) and its associated API. Most of the tools around GDX focus on data exchange with other systems based on a relational data model. There was significantly less attention on exchange with data structures of programming languages. GDX is very efficient when it comes to the sequential reading and writing of enormous amounts of data, but it also requires a thorough understanding of how GAMS/GDX stores data, and is limited in functionality (\textit{e.g.} read/write one symbol at a time, there is no direct access), and it is tied to a file which can cause a problem in some environments.

One important aspect of the OO API discussed in Section 2.3 is the class \texttt{GAMSDatabase}, which overcomes some of the GDX limitations at the price of potential loss of speed and increased memory consumption. The \texttt{GAMSDatabase} class is based on an in-core representation of GDX data that currently relies on standard C++ data structures such as \texttt{std::map}. The data structure provides efficient direct data access and also allows the user to iterate through the data in a sorted fashion. Single item insertion, deletion, and search have complexity $O(\log(n))$, with $n$ being the number of elements stored in the data structure.

A few primary issues occur when transferring sparse multi-dimensional data from GAMS into dense arrays. Dense arrays represent a widely used data structure that allows constant time access, but serves only as an example for other more specialized data structures. In our examples, we choose the language C# and restrict ourselves to two dimensions, although real world models often have symbols with ten and more dimensions. That being said, the presented code looks very similar in other languages. Some examples make use of advanced data structures such as dictionaries, \textit{e.g.} \texttt{Dictionary<,>} in C#, that might not be readily available in the standard libraries of languages like C or Fortran.

Let us first consider some GAMS code that generates a sparse two-dimensional parameter \texttt{ij} as shown in Listing 2:

1. \texttt{Set i / i1*i15000 /, j / j1*j15000 /;}
2. \texttt{Parameter ij(i,j);}
ij(i,j)$(uniform(0,1)<0.1) = normal(1,0.1);
execute_unload 'sparse.gdx',i,j,ij;

Listing 2: Unloading a two-dimensional parameter into a GDX container in GAMS.

While parameter \( ij \) could in principle hold \( 2.5 \cdot 10^7 \) elements, the filtering \( $(uniform(0,1)<0.1) \) in line 3 ensures that, on average, only every tenth \((i,j)\) pair is entered into \( ij \). The number stored is a random number distributed according to the normal distribution with mean 1 and standard deviation 0.1. It is important to understand that set labels in GAMS, such as \( i_1 \) and \( j_1 \), are strings, even if the string represents a simple number and no address calculations can be performed using the set labels. In most real world applications, set labels come from data bases and are rarely as structured as in our example. The statement in line 4 exports the sets \( i \) and \( j \) as well as the parameter \( ij \) to the GDX container “sparse.gdx”.

In order to make the example more interesting, we compute the inverse of \( ij \) in our C# code\(^3\). The following GAMS code then executes the C# program, imports the inverse \( ji \), and verifies the result:

execute 'calcInverse'; abort$errorlevel 'problems running calcInverse'
Parameter ji(i,j) inverse of ij;
execute_load 'inverse.gdx',ji;
alias (i,ip); parameter id(i,ip) identity;
id(i,ip) = round(sum(j,ij(i,j)*ji(j,ip)),12);
abort$(not(card(id)=card(i) and sum(i, id(i,i))=card(i))) 'ji is not inverse of ij';

Listing 3: Loading a two-dimensional parameter from a GDX in GAMS.

The task is now to transfer the \( ij \) GAMS data into a dense array of size \(|i| \cdot |j|\), calculate the inverse and write the dense result array back as sparse GAMS data. In GAMS, the order of set elements is usually of little significance for the user, although exceptions such as lead or lag operators exist. Storing the data in a dense array therefore requires the user to establish a map providing a unique number between 1 and \(|i|\), or 0 and \(|i|−1\) for each element, depending on how the programming language bases their array index. Four different ways to accomplish this task and the pros and cons of each are discussed below. All computations were carried out using Visual Studio 2010 on a 64 bit Windows 7 machine with an i7-4770 CPU at 3.40 GHz.

### 3.1. Data Transfer Using the GAMS OO API.

The first two examples use the GAMS OO API in which many GAMS components have their counterparts as classes. For example, the sets \( i \) and \( j \) are instances of the class \texttt{GAMSSet}. An instance of the \texttt{GAMSDatabase} class can be initialized from a GDX container, and provides access to the symbols and their data that were stored in the GDX container. The API provides methods for random access, such as \texttt{GAMSPParameter.FindRecord()}, and iteration through the records of a symbol, such as \texttt{foreach (GAMSSetRecord iRec in iSet)}. The following piece of code sets the stage for the actual data mapping:

\(^3\)This code is omitted here. It can be found at [http://www.rkinteractive.com/blogs/SoftwareDevelopment/post.aspx?id=ec104367-ece5-4ebe-a376-15defc5419b1](http://www.rkinteractive.com/blogs/SoftwareDevelopment/post.aspx?id=ec104367-ece5-4ebe-a376-15defc5419b1)
GAMSWorkspace ws = new GAMSWorkspace();
GAMSDatabase db = ws.AddDatabaseFromGDX("sparse.gdx");
GAMSSet iSet = db.GetSet("i"), jSet = db.GetSet("j");
GAMSParameter ijPar = db.GetParameter("ij");
GAMSParameter jiPar = db.AddParameter("ji", "inverse of ij", jSet, iSet);
int i=0, iCnt = iSet.NumberRecords, j=0, jCnt = jSet.NumberRecords;
double[][] ij = new double[iCnt][];

Listing 4: Preparations for importing a parameter from a GDX in C# (OO API).

In the following code, the C# data structure Dictionary<string, int> is used to map from the set elements of i and j to numbers 0 to |i|−1 and |j|−1, respectively. A reverse map is also required to write back the resulting ji:

Dictionary<string, int> iMap = new Dictionary<string, int>();
string[] iRMap = new string[iCnt];
i = 0; foreach (GAMSSetRecord iRec in iSet) {
  iRMap[i] = iRec.Key(0);
  ij[i] = new double[jCnt];
  iMap[iRMap[i]] = i++;
}

Listing 5: Importing data while mapping record labels into a string array in C# (OO API).

Except for line 5, we can do the same for set j. After the variables iMap and jMap have been established, we are ready to receive records from parameter ijPar to store into array ij by assigning the proper value to ij[iMap["i..."],jRMap["j..."]]. After calling the inverse routine, we write the result back to the jiPar by finding correct label string using the reverse map iRMap[i]:

double[][] ji = InvertMatrix(ij);
for (j = 0; j < jCnt; j++)
  for (i = 0; i < iCnt; i++)
    jiPar.AddRecord(jRMap[j], iRMap[i]).Value = ji[j][i];
db.Export("inverse.gdx");

Listing 6: Exporting data into a GDX using the record label map in C# (OO API).

Because most implementations of dictionaries or maps have complexity \(O(\log(n))\) for element insertion, deletion, and search, the entire data transfer from parameter ijPar to dense array ij has complexity \(O(|ij| \cdot \log(|i|)). A lookup in the reverse map has complexity \(O(1)\) and the writing therefore has complexity \(O(|ji|) = O(|j| \cdot |i|)\) because the inverse ji is dense. This consideration leaves out the effort done by the GAMS class methods like GAMSParameter.AddRecord. As mentioned above, the data structures used inside the OO API also require \(O(\log(n))\) for search and insert, so it takes some time to build up the data structure by implicit or explicit insert operations, such as GAMSDatabase.AddDatabaseFromGDX() or GAMSParameterRecord.AddRecord().

Next, we briefly describe some power calls, currently only available in the GAMS .NET OO API, that efficiently map between GAMSParameters and multi-dimensional C# arrays. Please note the slight difference between double[][] (jagged array) and double[,] (multi-dimensional array):
double[,] ij = new double[iCnt,jCnt], ji = new double[jCnt,iCnt];
ijPar.CopySparseToDenseArray(ij, iSet, jSet);
...
jiPar.CopyFromDenseArray(ji, jSet, iSet);
db.Export("inverse.gdx");

Listing 7: Power calls to import and export data from and to a GDX in C# (OO API).

The `CopySparseToDenseArray` in line 2 takes the records of the parameter `ijPar` and copies them into the multi-dimensional array `ij`, ensuring that the array `ij` has the correct dimension two and the correct size $|i| \cdot |j|$. The resulting `ij` is identical to the one built in the first example. The `CopyFromDenseArray` in line 4 does the reverse job, copying nonzero elements from the dense array `ji` (ensuring the correct dimension and size via the arguments `jSet` and `iSet`) into the parameter `jiPar`. The user does not have to build-up maps and reverse maps between `i` and number 0 to $|i| - 1$. This mapping is done internally and uses techniques described in the next example.

Before we move on to the next example, let us record some timing. As mentioned above, effort is required to set up the internal data structures when building a `GAMSDatabase` from a GDX container. This effort usually pays off, but in this example, we do not take advantage of the added capability of the `GAMSDatabase` class because we are only using it to write to a GDX. Such effort is therefore wasted. If the API offered a direct way to write to GDX, we would have used that and skipped the insertion into `std::map`. The setup time for both examples is identical and takes about 1.43 seconds. The transfer time from `ijPar` into array `ij` using C# `Dictionary<string,int>` takes about 2.71 seconds. The transfer of `ji`, which is typically a dense matrix, i.e., 10 times larger than `ij`, to `jiPar` using the reverse map consumes 47.47 seconds. Writing the GDX file via `db.Export("inverse")` requires 4.84 seconds. The majority of the time is spent entering the dense result data due to the combined effect of string conversion between C# strings and C++ strings and the complexity to build-up the `std::map`. The power calls `CopySparseToDenseArray` and `CopyFromDenseArray` avoid the string conversion and have more direct access to the `std::map`. Hence, this runs significantly faster: 0.12 seconds for `CopySparseToDenseArray` and 9.87 seconds for `CopyFromDenseArray`.

### 3.2. Data Transfer Using the GAMS Expert-Level GDX API.

As noted above, the OO API does a lot of extra work that is not required for this (and many other) examples. The string communication also adds an additional cost. The following example is based on the expert-level GDX API. GDX allows different ways to sequentially access the data, including a string interface. We use the raw interface which, instead of handling the set elements as strings, associates an integer between 1 and $N$. This $N$, also known as the number of unique elements or `UelCnt` in our program, is the total number of distinct labels used in the GAMS program and, therefore in the resulting GDX container. Even for large applications, the total number of labels is usually relatively small compared to the actual records stored. In our example, there are 10,000 distinct labels. The GDX container provides maps from $i$ and $j$ into the numbers 1 to $N$, which can be accessed by the raw interface.

The following lines of code open the GDX container “sparse.gdx” for reading and extracting the total number of distinct labels $N$ (`UelCnt`):

```csharp
string Msg=string.Empty;
```
Next, we need to identify labels associated with sets $i$ and $j$, respectively, in the list of all labels, because labels are not necessarily ordered in this list:

```csharp
int[] KeyInt = new int[2];
double[] Values = new double[0];
int k = 0, iSym = 0, iCnt = 0, DimFirst = 0;
gdx.gdxFindSymbol("i", ref iSym);
gdx.gdxDataReadRawStart(iSym, ref iCnt);
int[] iMap = new int[UelCnt], iRMap[iCnt] = new int[iCnt] ;
while (gdx.gdxDataReadRaw(ref KeyInt, ref Values, ref DimFirst) != 0) {
  iMap[KeyInt[0]-1] = k++;
}
```

Listing 9: Building a map and a reverse map of record labels in C# (Raw API).

Array $iMap$ maps from the label list numbered 1 to $N$ into the numbers from 0 to $|i| - 1$ using a dense one-dimensional array of size $N$. This is achieved on lines 7-10 by iterating through the set $i$, obtaining the current label position in the numbers 1 to $N$ ($KeyInt[0]$), and storing the iteration counter $k$. At the same time, the reverse map $iRMap$ from numbers 0 to $|i| - 1$ into the numbers 1 to $N$ is established and will be of use when writing the results. This process is then repeated for set $j$. After establishing $iMap$ and $jMap$, we are ready to read the sparse $ij$ parameter into the dense $ij$ array:

```csharp
double[,] ij = new double[iCnt, jCnt];
gdx.gdxFindSymbol("ij", ref ijSym);
gdx.gdxDataReadRawStart(ijSym, ref ijCnt);
while (gdx.gdxDataReadRaw(ref KeyInt, ref Values, ref DimFirst) != 0) {
  ij[iMap[KeyInt[0]-1],jMap[KeyInt[1]-1]] = Values[0];
}
```

Listing 10: Importing data using the label map in C# (Raw API).

Let us take a closer look at $gdxDataReadRaw$. The call reads the set element numbers into the array $KeyInt$ and stores the associated value in $Values$. $Values$ is also an array because GDX stores bounds, level, marginal, and scale for variables and equations. GDX retrieves a sequence of record with the property that the $KeyInt$ array is ordered lexicographically. This allows for very efficient data storage. The disadvantage is the need to write data in a lexicographically ordered sequence. Our setup of $iRMap$ and $jRMap$ together with the correct loop order ensures this:

```csharp
gdxcs ogdx = new gdxcs(ref Msg);
ogdx.gdxOpenWrite("inverse.gdx", "calcInverse gdx raw", ref ErrNr);
ogdx.gdxDataWriteRawStart("ij", "inverse of ij", 2, gamsglobals.dt_par, 0);
for (j = 0; j < jCnt; j++) {
  KeyInt[0] = jRMap[j];
  for (i = 0; i < iCnt; i++) {
    KeyInt[1] = iMap[i];
    Values[i] = ij[iRMap[i], jRMap[j]];
KeyInt[1] = iRMap[i];
Values[gamsglobals.val_level] = ji[j,i];
ogdx.gdxDataWriteRaw(KeyInt, Values);
}
ogdx.gdxDataWriteDone();

Listing 11: Exporting data into a GDX using the reverse label map in C# (Raw API).

In a final step, we need to register the string labels that correspond to the numbers 1 to \( N \) in the newly created GDX container. Since we use the map from the input GDX container, we just need to copy:

```
ogdx.gdxUELRegisterRawStart();
for (int uel = 1; uel <= UelCnt; uel++) {
    string label = string.Empty; int UelMap = 0;
    gdx.gdxUMUelGet(uel, ref label, ref UelMap);
    ogdx.gdxUELRegisterRaw(label);
}
ogdx.gdxUELRegisterDone();
ogdx.gdxClose();
```

Listing 12: Registering record labels inside the exported GDX in C# (Raw API).

Because the GDX raw example works without advanced data structures like a dictionary, it is easily translated into languages like Fortran or C. However, compared to the OO API interface, the expert level API requires much more insight into the way GDX works internally, and lexicographical ordering adds another layer of complexity. Nevertheless, it can be extremely fast when implemented correctly. In our example the execution using this interface requires 0.33 seconds for reading and 1.96 seconds for writing.

### 3.3. Data Transfer Using the IDX API.

In all examples so far, the mapping from GAMS labels, either as a string or as an internal label number, was performed in the programming environment, i.e., the API software. This last example, which does not rely on advanced data structures, shifts the work of mapping from \( i \) to numbers 1 to \( |i| \) to the GAMS program. If a symbol in GAMS is indexed over a set whose labels are natural numbers 1, 2, ..., the GDX unload command `execute_unloadidx "filename.gdx", symbol;` can be used instead of a traditional unload command such as the one on line 4 in Listing 2. GAMS creates a special GDX file that provides the data in a format convenient to store in arrays. Listing 2 is modified as follows:

```
Set i / 1*5000 /, j / 1*5000 /;
Parameter ij(i,j);
ij(i,j)$(uniform(0,1)<0.1) = normal(1,0.1);
execute_unloadidx 'sparse_idx.gdx',ij=ijIdx;
```

Listing 13: Unloading a two-dimensional parameter into a special IDX GDX in GAMS.

The C# program for processing the special GDX container looks very similar to the GDX raw example in Section 3.2 with respect to the API usage, but it does not require any mapping. Also, note that sets \( i \) and \( j \) are not exported anymore. The dimension of
data \(ij\) can be retrieved through the expert-level IDX API. We omitted the declaration of variables in the following code example because it is similar to previous examples:

```csharp
int ijDim = 0, ijCnt = 0;
int[] ijDimensions = new int[2];
double Value = 0.0;
idxcs idx = new idxcs(ref Msg);
idx.idxOpenRead("sparse_idx.gdx", ref ErrNr);
idx.idxDataReadStart("ijIdx", ref ijDim, ref ijDimensions, ref ijCnt, ref Msg);
ijCnt = ijDimensions[0]; jCnt = ijDimensions[1];
while (idx.idxDataRead(ref KeyInt, ref Value, ref DimFirst) != 0)
ij[KeyInt[0] - 1][KeyInt[1] - 1] = Value;
idx.idxDataReadDone();
```

Listing 14: Importing a parameter from a special IDX GDX in C# (IDX API).

The dimension of \(ij\) is obtained in line 7 and a map of labels to number 0 to \(|i|−1\) does not need to be established when reading the data in line 9. Also, a reverse map does not need to be built, nor do labels need to be registered, as shown in Section 3.2. The labels 1, 2, ..., \(N\), where \(N\) is the maximum dimension in any symbol, is implicitly registered by the API. Writing the dense parameter \(ji\) back into a GDX reads as follows:

```csharp
int[] jiDimensions = new int[2] { jCnt, iCnt };
oidx.idxDataWriteStart("jiIdx", "jiIdx inverse of ijIdx", 2, jiDimensions, ref Msg);
for (int j = 0; j < jCnt; j++) {
    KeyInt[0] = j + 1;
    for (int i = 0; i < iCnt; i++) {
        KeyInt[1] = i + 1;
        oidx.idxDataWrite(KeyInt, ji[j][i]);
    }
}
oidx.idxDataWriteDone();
oidx.idxClose();
```

Listing 15: Exporting a two-dimensional array to a GDX in C# (IDX API).

As expected, our third example performs very similarly to the example shown in Section 3.2. Indeed, the reading part takes 0.25 seconds and is slightly faster, while the writing part takes 2.12 seconds and is slightly slower. The biggest drawback of the IDX interface is the renaming of labels in the original GAMS program to numbered labels 1, 2, 3, ..., because data typically comes from an external source such as a data base. As a workaround, GAMS provides tools to efficiently build a map from a set with arbitrary labels to numbered labels using the alignment operator ":=". The following code allows use of the labels from Listing 2 and still unloads a special GDX:

```plaintext
Set i / i1*i5000 /, j / j1*j5000 /, iIdx / 1*5000 /, jIdx / 1*5000 /, iMap(i,iIdx) / #i:#iIdx /, jMap(j,jIdx) / #j:#jIdx /;
Parameter ij(i,j), ijIdx(iIdx,jIdx);
ij(i,j)$uniform(0,1)<0.1 = normal(1,0.1);
ijIdx(iIdx, jIdx) = sum((iMap(i,Idx),jMap(j,Idx)), ij(i,j));
execute_unloadidx 'sparse_idx.gdx', ijIdx;
```

Listing 16: Mapping conventional labels to numbered labels in GAMS.
Table 2 summarizes all writing and reading times using the four approaches.

In Sections 2 and 3, we described GAMS features that allow efficient prototyping of algorithms in GAMS. In this section, we present an application of these features in form of a long-term generation expansion planning model which has a structure suited for decomposition approaches. In particular, a state-of-the-art Benders decomposition type approach is applied. Both model and algorithm are taken from Lohmann and Rebennack (2014), which is motivated by Fell and Linn (2013). However, compared to Lohmann and Rebennack (2014), the problem is significantly simplified, missing ramping and transmission constraints as well as multi-period investment and retirement decisions. Solving the original problem is not possible as a monolithic model for relevant planning horizons due to its size and complexity, even on an Amazon Cloud server with 122 GB of memory. The strong simplification allows us to present a benchmark for our algorithm and, more importantly, modeling features.

4.1. Long-Term Generation Expansion Planning Model.
Long-term generation expansion planning models have a long history in linear programming and are among its first applications (Massé and Gibrat 1957). The goal of these problems is to find an optimal generation capacity expansion for an existing system of power plants. The planning horizon typically spans two or more decades and models were originally designed for use in a monopoly (Anderson 1972), rather than a modern electricity wholesale market. Thus, electricity demand is not seen as a function of the electricity price, but as a fixed quantity. Several authors have modeled and discussed the impact of elastic demand (as opposed to fixed demand) in a market environment in which buyers of electricity face producers (Murphy and Smeers 2005, Borenstein 2005, Bushnell 2010, DeJonghe et al. 2012). The model we present in this section features demand as a function of price, an hourly market clearing condition and a planning horizon of more than two decades. More specifically, we assume that in each hour of the planning horizon, demand function $f$ is defined as

$$Q = f(P),$$

where $Q$ is the electricity load and $P$ the price of that hour. We assume $f$ to be a continuously differentiable, strictly monotonically decreasing function. In our context, it is more convenient to refer to the inverse demand function $P = f^{-1}(Q)$, or equivalently $P(Q)$.

The simplified power generation expansion planning (PGEP) model reads as follows

$$\begin{align*}
\text{(PGEP)} \quad w^* := \max & - \sum_{i \in \mathbb{N}} \gamma_i N F_{i}^{I} K_i - \sum_{i \in \mathbb{N}} (\beta)^t \left[ \sum_{i \in \mathbb{N}} \gamma_i N F_{i}^{OM} K_i + \sum_{i \in \mathbb{N}} F_{i}^{OM} K_i \right] \\
& + \sum_{h \in H} \left( \int_0^{Q_{ht}} P_{ht}(x)dx - \sum_{i \in \mathbb{N}} c_i q_{ih} \right)
\end{align*}$$

(2)
\[ Q_{ht} = \sum_{i \in I} q_{iht} \quad \forall h \in \mathbb{H}, \; t \in \mathbb{T}, \]  
\[ 0 \leq q_{iht} \leq C_{ih} K_i \quad \forall i \in I^{EX}, \; h \in \mathbb{H}, \; t \in \mathbb{T}, \]  
\[ 0 \leq q_{iht} \leq \gamma^N_i C_{ih} K_i \quad \forall i \in I^N, \; h \in \mathbb{H}, \; t \in \mathbb{T}, \]  
\[ 0 \leq \gamma^N_i \leq 1 \quad \forall i \in I^N, \]  

where

\( h \in \mathbb{H} \) (set) hours [-]
\( i \in I^{EX} \) (set) existing generators [-]
\( i \in I^N \) (set) new generators [-]
\( t \in \mathbb{T} \) (set) years [-]
\( C_{ih} \) (parameter) capacity factor of generator \( i \) in hour \( h \) [$/MWh]
\( c_{it} \) (parameter) marginal generator cost of generator \( i \) [$/MWh]
\( \hat{c}_{it} \) (parameter) start-up cost of generator \( i \) [$/MW]
\( F_i^I \) (parameter) investment cost of generator \( i \) [$/MW]
\( F_i^{OM} \) (parameter) operation & maintenance cost of generator \( i \) [$/MW]
\( K_i \) (parameter) capacity of generator \( i \) [MW]
\( \beta \) (parameter) discount factor [100%]
\( \gamma^N_{it} \) (continuous variable) investment decision for generator \( i \) in year \( t \) [100%]
\( Q_{ht} \) (continuous variable) electricity load in hour \( h \) of year \( t \) [MWh]
\( q_{iht} \) (continuous variable) dispatch of generator \( i \) in hour \( h \) of year \( t \) [MWh]
\( P_{ht} \) (function) inverse demand function in hour \( h \) of year \( t \) [$/MWh]

The objective function (2) consists of three terms: investment costs for new generators, annual operation and maintenance costs for all generators, and the hourly welfare of the power system. The investment decision into new generators is represented by continuous decision variables \( \gamma^N \) which are restricted by (6). Figure 1 visualizes the hourly market clearing condition.

![Figure 1](image-url)

**Figure 1** Market clearing condition with a linear demand function. The grey boxes represent supply bids.

The blocks represent supply bids into the wholesale market. We assume perfect competition in the market and the bid-based dispatch therefore equals the central dispatch
(Steeger et al. 2013). In other words, supply bids are equivalent to generators’ marginal price and capacity. The equilibrium price and quantity are achieved in \((Q^*, P^*)\). Constraint (3) defines the equilibrium quantity \(Q^*\) and the individual bids are represented by box constraints (4)-(5). Model (PGEP) describes highly stylized electricity market model.

Lohmann and Rebennack (2014) present several extensions to (PGEP) and discuss its limitations in greater detail. However, (PGEP) captures the fundamental functionality of an electricity wholesale market for a small power system. Only a single market price for the entire system is obtained because we do not consider transmission (Frank et al. 2012).

4.2. Solution Algorithm for (PGEP).
Investment decisions \(\gamma^N\) are seen as complicating variables in Benders decomposition methodology. In other words, if we somehow know \(\gamma^N\), we can solve a subproblem, consisting of constraints (3)-(5) and the objective function terms associated with the subproblem variables:

\[
\left(\frac{\partial}{\partial x}\right)^t \left[ \sum_{h \in H} \left( \int_0^{Q^t_{ht}} P_{ht}(x) dx - \sum_{i \in I} c_i q_{ih} \right) \right].
\]  

(7)

All other constraints and objective function terms are contained in a so-called master problem. Benders decomposition builds on the concept that the subproblem can be exactly represented by a finite number of hyperplanes, commonly referred to as Benders optimality and feasibility cuts, in the master problem. In our case, feasibility cuts are not necessary since the subproblem is feasible for any choice of \(\gamma^N\). The Benders optimality cuts are constructed by alternatingly solving the master and subproblem, with the hope that only a subset of all hyperplanes is necessary to arrive at the optimal solution to the full problem. A feasible solution to the full problem is constructed in each iteration. The full problem is solved to optimality when the upper bound, obtained as the objective function value of the master problem with optimality cuts, and the lower bound, described by the objective function value of the feasible solution to the full problem, are equal (or within a given tolerance).

In the case of (PGEP), the objective function of the subproblem (7) is a nonlinear term. With the assumption that \(f\) is a continuously differentiable, strictly monotonically decreasing function, (7) is concave (Lohmann and Rebennack 2014). We propose to dynamically overestimate (7) with linear hyperplanes in \(Q^t_{ht}\), improving the approximation in each iteration of the Benders decomposition approach. The overestimator reads as follows:

\[
\phi_{ht} \leq \omega^s_{sht} Q^t_{ht} + \omega^c_{cht} \forall \kappa \in K_{ht}, h \in H, t \in T,
\]  

(8)

where \(\phi_{ht}\) replaces the integral term in (7). Parameters \(\omega^s_{sht}\) represent the slopes and parameters \(\omega^c_{cht}\) the intercepts of the linear overestimators. It can be proven that the resulting Benders optimality cuts remain valid and that the decomposition converges in finitely many iterations (Lohmann and Rebennack 2014).

Since the hours in (PGEP) are independent for a given \(\gamma^N\), the subproblem itself can be decomposed into \([H] \cdot [T]\) independent one hour subproblems. All parameters of the subproblems are known once \(\gamma^N\) is given, which makes it an ideal application of the grid-enabled GUSS facility discussed in Section 2.2. All parameter updates and desired optimal solution values are prepared in the GUSS dictionary before each subproblem solve. For (PGEP), variable upper bounds \(C_{ih} K_i\) (\(\gamma_i C_{ih} K_i\) for new generators), marginal generator costs \(c_{it}\), overestimator slope \(\omega^s_{sht}\) and intercept \(\omega^c_{cht}\) are updated. Since none of the above
parameters are indexed by both \( h \) and \( t \), an efficient updating scheme can be implemented, bundling hours into scenarios and assigning them to specific threads in the grid enabled GUSS facility. The required optimal solution values besides the objective function value are \( Q_{ht}^* \) and dual variables for (5). These are necessary to update the overestimator function (8) and to construct a new Benders optimality cut in each iteration.

Since the subproblem decomposes into \( |\mathcal{H}| \cdot |\mathcal{T}| \) independent one hour subproblems, and more specifically hourly market clearing conditions, we can calculate the optimal solution values for \( Q_{ht} \). These are used as breakpoints in (8) to improve the dynamic overestimator. If \( \gamma_i \) is known and generators \( i \) are sorted by their costs \( c_i \) in ascending order, we have the situation as shown in Figure 1 for a given hour in a given year. The intersection of inverse demand function \( P(Q) \) with the supply curve consisting of all generator bids can be efficiently computed with a tailored algorithm in polynomial time, as shown in Lohmann and Rebennack (2014). This is repeated for all hours, because both the demand curve and the supply curve changes. GAMS as an interpreted language is rather inefficient for computing these intersections as it involves nested looping over years, hours, and generators, yearly sorting of generators, and does not require solver calls. Thus, we rely on GAMS’s IDX API presented in Section 3.3 and implement the breakpoint algorithm in C#. The challenging part is to efficiently communicate the necessary power system data, especially \( C_{ih} \), which has several million entries, between GAMS and C# in every iteration of the Benders decomposition.

5. Computational Results.

We define a set of instances as shown in Table 3 to compare all approaches for differently sized models, even though the intention of the model may be lost. The largest instance we solve has a time horizon of 23 years, ranging from 2008 to 2030, and 410 generators (380 existing and 26 investment options), as first presented by Fell and Linn (2013). In this section, we compare the runtimes for the algorithm from Section 4.2 to a simple GAMS Benders decomposition implementation which does not use the grid-enabled GUSS facility or the breakpoint algorithm in C#, but solves the subproblem in a loop. The 23·8760 hours of (PGEP) are decomposed into blocks of 100 hours, which is possible because they are independent for a fixed \( \gamma_i \). We further apply the following linear demand function for all models

\[
Q_{ht} = g_t \cdot a_h - b_h \cdot P_{ht},
\]

where \( g_t \) is a demand growth parameter, \( a_h \) describes the intercept, and \( b_h \) the slope of the demand curve. It follows that the monolithic models are continuous and convex quadratic programming (QP) problems, and we solve them without overestimators to obtain a benchmark. Nonlinear demand functions would render a monolithic approach as practically intractable for the larger instances. We refrain from solving the subproblem as a QP since GUSS is currently not able to handle QPs in which quadratic matrix coefficients in the objective function, usually referred to as \( Q \) matrix, are updated. Further, LPs allow us to demonstrate the effectiveness of GUSS’s advanced basis capability \(^4\).

All computations were carried out in the Amazon Elastic Compute Cloud (Amazon EC2). This Windows 2012 64 bit server has 16 virtual processors, Intel(R) Xeon(R) CDU E5-2670 v2 at 2.50 GHz and 122 GB of memory. The models were solved using GAMS

\(^4\)All GAMS model files, GDX files and C# files necessary to reproduce our results are available under http://www.gams.com/modlib/adddocs/HighPerformancePrototypingofDecompositionMethodsinGAMS.zip
24.3.3 (64 bit) with CPLEX 12.6.0.1. The convergence criterion for all algorithms is $10^{-7}$ and the approximation tolerance for the linear overestimators is set to $10^{-9}$. Similar to the simple implementation, the hours of (PGEP) are grouped into scenarios for GUSS as this is more efficient than updating and solving single hour problems. We decide to bundle 100 hours into one scenario, resulting in 88 scenarios per year. Since updating $C_{ih}$ is especially costly due to its size, we use a complex updating scheme, assigning the same 100 hours in all 23 year to the same thread. Each one-hour slice of $C_{ih}$ is therefore used for updating the model instance exactly once.
<table>
<thead>
<tr>
<th>#</th>
<th>Hours</th>
<th>Monolith Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#</td>
</tr>
<tr>
<td>1</td>
<td>8.760</td>
<td>0.245</td>
</tr>
<tr>
<td>2</td>
<td>17.520</td>
<td>0.491</td>
</tr>
<tr>
<td>3</td>
<td>35.040</td>
<td>0.981</td>
</tr>
<tr>
<td>4</td>
<td>70.080</td>
<td>1.962</td>
</tr>
<tr>
<td>5</td>
<td>140.160</td>
<td>3.924</td>
</tr>
<tr>
<td>6</td>
<td>201.480</td>
<td>5.641</td>
</tr>
</tbody>
</table>

Table 3  Instance characteristics: All numbers are in millions.

<table>
<thead>
<tr>
<th>#</th>
<th>Monolith [Solution Times]</th>
<th>Simple [Solution Times]</th>
<th>Grid-enabled GUSS [Solution Times]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4  Computational results: All times are in seconds. The second row shows the number of threads used. Columns labeled with "Loops" describe the total time in seconds spent in the submission and collections loops (cf. Listing 1).

<table>
<thead>
<tr>
<th>#</th>
<th>Monolith [Memory]</th>
<th>Simple [Memory]</th>
<th>Grid-enabled GUSS [Memory]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5  Maximum memory required: All memory numbers are in gigabyte. The second row shows the number of threads used. Factor columns represent the improvement factor of the grid-enabled GUSS approach to its respective monolith.
CPLEX is able to solve the QP monolith for all instances with its (default) barrier method, but requires an enormous amount of memory. Recall that the original monolith is simplified to achieve this (cf., Lohmann and Rebennack (2014)). An off-the-shelf desktop computer would only solve the first three instances, cf., Table 3. The decomposition approaches require significantly less memory because they solve a series of 100 hour models. The total memory demand for the algorithm using GUSS is consistent across all instances, never exceeding 2 GB. The memory demand of the simple implementation is significantly higher since all levels and marginals of all equations and variables are stored. Recall that GUSS only stores those values that are requested by the user.

Regarding solution times, the GUSS approach loses to the monolith for instances 1 and 2, but is equally fast or faster for the larger instances. The reason is that the data preparation effort for the smaller instances is proportionally greater than for the large instances because of our updating scheme in GUSS. Furthermore, this part of the algorithm can be parallelized. While Table 4 shows a factor of 1.95 going from one thread to eight threads in the algorithm using GUSS, the time spent in the GUSS itself decreased from 662.8 seconds to 251.9 seconds, or factor 2.63.

The reason why this factor is not closer to 8 was briefly addressed in Section 2.2. We would like to elaborate on the uniform return times with an example. Note that the following times are independent of Table 4 and were obtained during test runs before the final run for this paper. The following list shows the thread return times sorted in ascending order for instance six:

One thread: 34.541
Two threads: 21.905, 29.401
Four threads: 15.420, 16.808, 22.586, 23.770

Even though each thread was assigned the same number of scenarios, the longest return time on eight threads is almost twice as long as the shortest. If we are able to spread the effort for the threads more evenly, better parallelization can be achieved.

While the simple implementation is outclassed by all other approaches, it provides insight into the speed-up achieved with GUSS and the efficiency of Algorithm 1. The time spent solving the subproblem per iteration is at 505.8 seconds for instance six. GUSS only takes 50.9 seconds per iteration on one thread, a factor of almost 10. With little additional effort, the number of threads used with GUSS can be increased, decreasing the subproblem time per iteration to 19.3 on eight threads. Even though one may argue that the subproblem is harder to solve due to the inferior overestimator in the simple implementation, this demonstrates GUSS’s efficiency at updating a 100 hour instance, in this case 88·23 times per iteration. The number of iterations decreases in the presence of the breakpoint algorithm, a component whose efficient integration is made possible because of GAMS’s API.

For the results reported in this paper, the IDX API described in Section 3.3 was used as it proved to be the fastest. Two GDX files are read and one is written back in each iteration. Table 6 shows their content and respective number of records for instance six.

The input data is divided into GDX 1 and GDX 2 to avoid unloading and mapping (IDX API) the contents of GDX 1 in each iteration. Note that all parameters are 100% dense. The average time spent in C# per iteration using the IDX API is 0.912 seconds for instance six on eight threads, which for all iterations equates to 2.7% of total solution time. Table 7 shows the time spent in C# in the first iteration for the OO API with power calls, the GDX Raw API and the IDX API. Similar to Section 3, the IDX and Raw APIs prove to be the most efficient interfaces.
6. Conclusion.

Two recently added GAMS features are demonstrated to be both practical and viable for use on a large scale case study that involves a Benders decomposition algorithm. The grid-enabled GUSS facility, a “hybrid” tool that combines advantages of algebraic modeling languages and solver interfaces, allows the user to efficiently solve collections of models with similar structure. While GUSS allows a model representation close to the solver, it offers all the conveniences of an algebraic modeling language. GUSS efficiently solved the subproblem in our decomposition, outperforming a simple implementation by a factor 12.6 to 24.6 for the largest instance, depending on the number of threads.

The GAMS OO API allows the user to create and manage a model instance from a traditional GAMS model. Since algebraic modeling languages and imperative programming languages typically represent data in different ways, complex mapping mechanisms, mostly invisible to the user, are required. While the four available interfaces discussed facilitate this transfer of data, each has advantages and disadvantages. The fastest interface was chosen to demonstrate its usefulness in a prototyping setting. In our case study, the interfaces communicated close to four million records between GAMS and C#, in each iteration, avering 0.912 seconds in C#. This number includes the time spent for computations.

Both features combined significantly improve GAMS’s potential as a prototyping environment. Interfacing an imperative programming language to outsource inefficient computations and efficiently updating a model instance that is kept alive inside the solver can be key features that decide over the success of a prototyped algorithm.

References


