Grid Enabled Optimization with GAMS*

Michael R. Bussieck† Michael C. Ferris‡

Alexander Meeraus§

September 2007

Abstract

We describe a framework for modeling optimization problems for solution on a grid computer. The framework is easy to adapt to multiple grid engines, and can seamlessly integrate evolving mechanisms from particular computing platforms. It facilitates the widely used master/worker model of computing and is shown to be flexible and powerful enough for a large variety of optimization applications. In particular, we summarize a number of new features of the GAMS modeling system that provide a lightweight, portable and powerful framework for optimization on a grid. We provide downloadable examples of its use for embarrasingly parallel financial applications, decomposition and iterative algorithms and for solving very difficult mixed integer programs to optimality. Computational results are provided for a number of different grid engines, including multi-core machines, a pool of machines controlled by the Condor resource manager and the grid engine from Sun Microsystems.

1 Introduction

There are probably two main sources for parallelism within optimization algorithms. First, and foremost, there is the opportunity to use parallel computations to aid in the search for *global* solutions, typically in a nonconvex (or discrete) setting. Important techniques of this kind either involve multiple trial points or search processes (including pattern searches, evolutionary algorithms [2, 35], heuristics [41] or multi-start methods) or computations to efficiently

*This work is supported in part by Air Force Office of Scientific Research Grant FA9550-07-1-0389, and National Science Foundation Grants DMI-0521953, DMS-0427689 and IIS-0511905.

†GAMS Software GmbH, Eupener Str. 135-137, 50933 Cologne, Germany. (MBussieck@gams.com)

‡Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, Wisconsin 53706, USA. (ferris@cs.wisc.edu)

§GAMS Development Corp., 1217 Potomac St, NW Washington, DC 20007, USA. (AMeeraus@gams.com)

explore a complete enumeration of a large set of trial points (including branch-and-bound [33], or branch-and-cut methods [50]). Secondly, optimization algorithms have utilized building blocks, most prominently decomposition and parallel linear algebra techniques, to exploit the computational powers of high performance machines.

Examples of the success of these techniques in the optimization field include many implementations of parallel heuristics such as genetic algorithms, pattern search [18] and multi-start methods, the multi-threaded version of CPLEX [9, 38], MOSEK [46], XA [53] and XPRESS [17] (both Mixed Integer Programming (MIP) and Barrier solvers), and the PICO [20], NINF [51] and TAO [48] toolboxes. In discrete optimization, see [32, 36, 37] for lists of references, while the texts [12, 13] provide a fuller perspective.

However, parallel computation has made far less difference within optimization than it has to computing or computational science in general. While there are many multi-threaded applications employing numerous algorithms developed from a multi-processing perspective, the number of (commercial-strength) parallel optimization codes is limited. This is in stark contrast to the availability of machines that can perform parallel computations. Even modern day laptops and desktop workstations typically come with multiple processing units.

Some of the reasons for the limited success is due to the nature of optimization. In most cases, optimization is concerned with a single objective function, and implicitly involves a synchronization step to determine if progress has been made. While there is some literature on partially synchronous and asynchronous methods [5, 22, 23], this step has been a significant limiting factor for the adoption of parallelism within optimization.

Other reasons are more generally applicable to high performance computing and involve the cost of accessing these machines, the difficulty in porting applications to these platforms, and the effects of Moore's law in making commodity computers quick to compete. Porting applications has become somewhat easier in the past decade due to the increased support for environments such as MPI (message passing interface) and PVM (parallel virtual machine)[31].

Since commodity computational components have become increasingly cheap and accessible, there has been an increasing interest in the last decade in grid computing [27, 45], a somewhat poorly defined but well known notion. Grid computing treats a confederation of loosely coupled heterogeneous computing resources as a single object and attempts to execute algorithms on such a platform. Here, there have been some notable successes, including the solution of large scale travelling salesman problems [4], the processing of difficult quadratic assignment problems [3, 54] and the resolution of optimality of some hard mixed integer programs [25]. The attraction of such an environment is that it can provide an enormous amount of computing resources, many of which are simply commodity computing devices, with the ability to run commercial quality codes, to a larger community of users. As such, this is sometimes called *computing for the masses*, or *poor man's parallelism*. This is the platform that we intend to exploit in this paper.

However, we believe that grid computational resources are not enough to

make parallel optimization mainstream. Setting aside the issue of data collection, it is imperative that we provide simple and easy to use tools that allow distributed algorithms to be developed without knowledge of the underlying grid engine. While it is clear that efficiency may well depend on what resources are available, it must be easy to generate large scale, difficult optimization problems, and high level implementations of methods to solve them. Stochastic programming is perhaps a key example, where in most of the known solution techniques, large numbers of scenario subproblems need to be generated and solved [42].

A modeling language [6, 28] provides a natural, convenient way to represent mathematical programs. These languages typically have efficient procedures to handle vast amounts of data and can quickly generate a large number of models. For this reason, modeling languages are heavily used in practical applications. This paper outlines some basic grid computing tools within the General Algebraic Modeling System (GAMS) that facilitate the parallel and asynchronous solution of models. A series of examples, from simple to complex will be used to illustrate the use of grid computing facilities. The examples are grouped into three sections, tracing of efficiency frontiers and scenario evaluations, implementing parallel decomposition methods, and developing asynchronous algorithms to solve extremely difficult mixed integer programming problems to optimality. As we move from simple to complex examples, the use of those techniques shifts from the practical to research. The use of a modeling language is an essential factor in the management and solution of realistic, application driven, large scale problems and allows us to take advantage of the numerous options for solvers and model types, and therefore enhance the applicability of these tools. Large scale nonlinear optimization problems can just as easily be decomposed using model level knowledge, and the techniques outlined here can be applied directly in such settings. Although, we will use GAMS, the system we are intimately familiar with, most what will be said could as well be applied to other algebra based modeling systems like AIMMS, AMPL, MOSEL, MPL, OPL and others.

The GAMS grid facility allows multiple optimization problems to be instantiated or generated from a given set of models. Each of these problems is solved concurrently on a grid computing environment. This grid computing environment can just be a laptop or desktop computer with one or more CPUs. Today's operating systems offer excellent multi processing scheduling facilities and provide a low cost grid computing environment. Most of the work reported here was done on a grid facilitated by the Condor system [43], a resource management scheme developed at the University of Wisconsin. A number of commercial grid computing resources are now available on an as-you-go basis and optimization software is beginning to appear. For example, GAMS and its grid facility is now available on SUN's network.com [52].

The paper is organized as follows. In Section 2, we outline the grid computing environment we are using while Section 3 explains the grid computing tools that are provided by GAMS. The following sections then outline specific examples of the use of these tools, ranging from applications in finance, through iterative approaches for linear and nonlinear systems of equations, and decompo-

sition approaches for structured problems and difficult mixed integer programs. Computational results detailing specific application of these tools to the solution of various optimization problems on a number of grid computing engines are provided, coupled with the use of advanced features of the modeling system for generation, collection and analysis of results.

2 Grid Computation Resources

As outlined in the introduction, grid computing attempts to take a confederation of loosely coupled heterogeneous computing resources and treat them as a single computing entity with the aim of executing parallel algorithms on it. Such systems can be rented or distributively owned, and examples include the Grid Engine from Sun Microsystems, various competing commercial systems from IBM and Oracle, and the Condor system, a system developed at the University of Wisconsin. International collaborations like the open source Globus Alliance [34] promote software research and development in fundamental Grid technologies. One aim of such systems is to provide effective sharing of the CPU power and to allow massive parallel task execution with a scheduler handling the management tasks. Clearly, there are issues (that we will not address here) related to licensing of software for these systems, communication between distributed components, and a suite of new security issues. As such, this remains an active area of research in distributed computing. While we do not focus on these issues here, there is considerable research related to the use of distributed shared data assets, and virtualization and integration technology.

As a particular example, the results contained here utilize the Condor system. Condor [21, 43] is a resource management tool that can provide a user with enormous computing resources. Originally designed upon the premise that most workstations are severely under-utilized, Condor notices when machines are idle and schedules tasks on them. When the owner wants to perform work on the machine, Condor removes the computation and returns the machine to its owner. More recently, Condor has also been extensively deployed on dedicated clusters and multiprocessor architectures running a variety of different operating systems, significantly increasing the amount of computational resources available. While the underlying hardware may be extremely varied, scheduling and control of jobs remains the same, and operates seamlessly across the different operating systems and distributed file systems.

The programming paradigm envisioned in this paper is the master/worker model in which a master program generates a large number of independent subproblems that can be solved in parallel by the workers. Once the subproblems finish, the master program performs additional computations and creates a new set of subproblems. The resources provided by Condor are typically not specialized high performance machines but large collections of underutilized workstations and clusters that are connected either via fast switches or possibly ethernet. As such, a user typically should aim to garner large amounts of resources for large periods of computation, and thus utilize *high throughput*

techniques as opposed to high performance (or real-time) methods.

Condor has been extended to allow parallelism [49], built largely using PVM. A further extension, the MW [40] API, facilitates specific *master-worker* tasks to be completed effectively. To some extent this abstracts the grid to a higher level, and allows a user to devise a parallel algorithm without understanding the underlying implementation on the grid. An important feature of this system (from a users perspective) is that whenever individual machines are updated, the power of the overall grid increases seamlessly. Furthermore, the system is available for download [44].

Since these systems are extremely powerful and becoming pervasive, we believe it is important to utilize such systems to carry out difficult optimization problems. The MIP solver Fatcop [14, 15] utilizes the MW API, as do several other applications. We aim to take the level of abstraction one level higher, and allow Condor (or any other grid computing system) to be used directly from within a modeling language. Our linkage of grid computing to modeling systems is an attempt to allow grid technology to be exercised by (non grid-expert) businesses and application modelers. Previous work in this vein can be found for example in [24].

3 The GAMS Grid Facility

Specifically, new language extensions of the GAMS modeling language are instantiated using a grid engine that can be managed either by the operating system or a specific grid resource manager such as Globus or Condor. The grid and modeling languages form a synergistic combination. Linking them together gives us expressive power and allows us to easily generate simple parallel programs. Successful applications of our mechanism should possess two key properties; they should generate a large number of independent tasks and each individual task should take a long time to complete. Applications with the above two properties cannot be reasonably performed serially. Furthermore, the model generation time and scheduling overhead are ameliorated by the resources spent solving each individual task.

The approach is very simple; we separate the solution into several steps which then can be controlled individually. This separation is not done automatically and it is completely up to the user of the system to decide which serial operations can be reworked in an asynchronous mode. We will first review what happens during the synchronous solution step and then introduce the asynchronous or parallel solution steps.

When GAMS encounters a solve statement during execution it proceeds in three basic steps:

Generation. The symbolic equations of the model are used to instantiate the model using the current state of the GAMS database. This instance contains all information and services needed by a solution method to attempt a solution. This representation is independent of the solution subsystem and computing platform.

Solution. The model instance is handed over to a solution subsystem and GAMS will wait until the solver subsystem terminates.

Update. The detailed solution and statistics are used to update the GAMS database.

In most cases, the time taken to generate the model and update the database with the solution will be much smaller than the actual time spent in a specific solution subsystem. Often the model generation takes just few seconds, whereas the time to obtain an optimal solution may take a few minutes to several hours. If sequential model solutions do not depend on each other, we can solve in parallel and update the database in random order. All we need is a facility to generate models, submit them for solution and continue. At a convenient point in our program we will then look for the completed solution and update the database accordingly. We will term this first phase the submission loop and the subsequent phase the collection loop:

Submission Loop. In this phase we will generate and submit models for solution that can be solved independently.

Collection Loop. The solutions of the previously submitted models are collected as soon as a solution is available. It may be necessary to wait for some solutions to complete by putting the GAMS program to *sleep*.

We now illustrate the use of the basic grid facility. Assume that we have a simple transportation model that is parameterized by given supply and demand information. Then the following statements instantiate those parameters and then solve the model, saving the objective value afterwards into a *report* parameter.

```
demand = 42; cost = 14;
solve mymodel min obj using minlp;
report = obj.l;
```

Note that a model in the GAMS language is just a collection of symbolic relationships, the equations. A solve statement then simply takes those relationships and instantiates the model using the current state of the GAMS database and passes it to some solution engine. Once a solution is obtained, the results are merged back into the GAMS database. To carry out multiple replications, a loop construct can be used. The only changes required are to instantiate different data and to save appropriate values from each solution.

```
loop(scenario,
   demand = sdemand(scenario); cost = scost(scenario);
   solve mymodel min obj using minlp;
   report(scenario) = obj.l );
```

However, the solve statement in this loop is blocking. Essentially, the solver takes the (scalar level) model that was generated, solves it, and returns the solution back to the modeling systems without releasing the process handle. The model attribute *solvelink* controls the behavior of the solve statement and

the value of *3* tells GAMS to generate and submit the model for solution and continue without waiting for the completion of the solution step; this is now a submission loop.

```
mymodel.solvelink=3;
loop(scenario,
   demand = sdemand(scenario); cost = scost(scenario);
   solve mymodel min obj using minlp;
   h(scenario) = mymodel.handle );
```

We obviously cannot save any solution values; instead we need to save some information that will later allow us to identify an instance. The model attribute *handle* provides this unique identifier. We store those handle values, in this case, in the parameter h, to be used later to collect the solutions. The following collection loop retrieves the solutions:

```
loop(scenario$handlecollect(h(scenario)),
    report(scenario) = obj.l );
```

The function handlecollect will interrogate the solution process and will *load* the solution and related information if the solution process has been completed for the given handle. If the solution is not ready, the function will return a value of zero and will continue with the next element in the set scenario. For readers not familiar with GAMS syntax; the *$* should read like a *such that* condition. The above collection loop has one major flaw. If a solution was not ready it will not be retrieved. We need to call this loop several times until all solutions have been retrieved or we get tired of it and quit. We will use a repeat-until construct and the handle parameter h to control the loop to look only for the solutions that have not been loaded yet, as shown below:

```
repeat
   loop(scenario$handlecollect(h(scenario)),
      report(scenario) = obj.l;
      h(scenario)=0 );
   display$sleep(card(h)*0.1) 'sleep a bit';
until card(h)=0 or timeelapsed > 100;
```

We use the handle parameter to control the extraction loop. Once we have extracted a solution we will set the handle parameter to zero. Before running the collection loop again, we may want to wait a while to give the system time to complete more solution steps. This is done with the conditional display statement which just executes a sleep command for 0.1 seconds times the number of solution not yet retrieved. The final wrinkle is to terminate after 100 seconds of elapsed time, even if we did not get all the solutions. This is important, because if one of the solution steps fail, our program would never terminate. The parameter h will now contain the handles of the failed solves for later analysis. As a final note, we have made no assumptions about what kind of solvers and

what kind of computing environment we will operate. The above example is completely platform and solver independent and it runs on a Windows laptop or on a massive grid system without any changes to the GAMS source code.

There are three handle functions that allow more control of the program. For example, we could rewrite the above collection loop:

```
repeat
   loop(scenario
      if(handlestatus(h(scenario))=2,
         mymodel.handle = h(scenario);
         execute_loadhandle mymodel;
         display$handledelete(h(scenario)) 'could not remove handle';
         h(scenario)=0 ));
   display$sleep(card(h)*0.1) 'sleep a bit';
until card(h)=0 or timeelapsed > 100;
```

The function handlestatus returns the current state of the solution process for a specific handle. If the return value is 2, we have a solution and can proceed to *load* all or parts of the solution. This is carried out in two steps; first we have to signal to the model which solution we want to load, and then we use the procedure `execute_loadhandle` to merge the solution into the current database. We also use the function `handledelete` to remove the instance from our system.

A few words regarding the implementation of the solver interface. The solution is returned in a GAMS Data Exchange (GDX) container. This container is a high performance, platform independent data exchange with APIs for most programming languages. Any data contained in GDX meets all syntactic and semantic rules of GAMS information and is used to communicate between GAMS systems components and any other external system like databases, spreadsheets and many other systems. GDX ensures data quality and provides one very important service; it manages the mapping of different name spaces. The GAMS data model is a subset of a relational data model which accesses data by descriptors and not location. Programming the mappings from data structures suitable for algorithms or programming languages to relational data spaces is error prone and expensive; GDX automates this step.

The actual submission of the model instance to the operating system for further processing is done via scripts. Whenever a solve statement is encountered while executing a GAMS program, control is passed to a script. In general this script is responsible for running the solver on the problem instance and passing back the solution to GAMS. In the grid environment, we simply use the file system to give each instance its own environment and its own directory. The script then schedules the solver execution. The solvers then generate the solution file and a flag to signal completion which the collection loop will understand and commence retrieval. This submission script centralizes all information required to tailor the system to a specific grid engine. Under Windows this would look like:

```
@echo off
: gams grid submission script
:
: arg1 solver executable
:    2 control file
:    3 scratch directory
:
: gmscr_nx.exe processes the solution and produces 'gmsgrid.gdx'
:
echo @echo off                        > %3runit.cmd
echo %1 %2 ^& gmscr_nx.exe %2       >> %3runit.cmd
echo echo OK ^> %3finished ^& exit >> %3runit.cmd
start /b /low %3runit.cmd  > nul
echo @start /b %3runit.cmd ^> nul  > %3gmsrerun.cmd
exit
```

In this case we use the `start` command to submit our instance to Windows. The script for Condor or any other grid system will look very similar. Once the script submits the job, the scheduler then queues the job, monitors its execution, and returns any solution back to the submitting machine (possibly the same machine). Once the solution is returned, a flag is set and the scheduler releases the job from the workpool. Note that this implementation conforms to the master worker paradigm, with the GAMS program being the master, and the grid resources taking on the role of the worker. GAMS generates tasks and is responsible for synchronization of the results, while the grid processes individual tasks and simply reports back results. This simplicity is critical in dealing with the dynamic nature of our grid system and help to enhance our fault tolerance, as we now briefly discuss, using Condor as an example.

A key feature of the Condor system, that we believe is of critical importance, is the dynamic nature of the provided grid resources. Machines may come and go, either as the demand for resources by other users increases or decreases, or due to failures of certain components of the grid, etc. The system is designed to be fault tolerant, with built in attributes that restart jobs after system crashes or other failures. However, as the number of resources used increases, so does the probability of a failure. Therefore, our algorithms must be designed to be fault-tolerant. In some cases the executable performing the computations can be linked with special libraries that allow the computation to be checkpointed and migrated to another machine. In cases where relinking is not possible, the tasks are migrated to another machine and the computation starts anew. For example, all of the solvers available in GAMS can be run on this system without recompilation, provided the user is willing to accept the loss of work when migration occurs.

GAMS provides another facility to help a modeler build in more fault tolerance, namely the handlesubmit function. When the initial task is generated for Condor submission, a resubmission script is also generated. Whenever a user determines that an optimization task did not perform as expected, the task can

be resubmitted using the syntax:

```
rc = handlesubmit(handle(scenario));
```

The return code indicates if the resubmission was successful or not. Several other modeling facilities help with fault tolerance including a persistent directory in which to store all the output from the grid jobs (`GRIDDIR`).

In the next sections we will illustrate the use of the grid facility in research and production environments with three sets of examples. The first set shows how to exploit the ability to do parallel solution when tracing efficiency frontiers or evaluate independent scenarios, the second set implements parallel decomposition schemes. The last set of examples shows how one can extend the simple grid facility to implement sophisticated methods to solve very difficult mixed integer models.

Before proceeding with these examples, we would like to give a brief summary of features introduced in this section. It is important to note that no new language constructs were required to program and manage the asynchronous execution of the model solution step, only minor extensions to existing structures have been added. For convenience we will group those extensions into:

Instance Identification. The identity of a specific model instance, the model handle, is simply encoded into a GAMS scalar parameter value which can be managed like any other data item.

Solution Strategy. The existing model attribute `<model>.solvelink`, that specifies the implementation of the solution process of a model instance, has two additional values to specify what type of asynchronous processing should be used. A new attribute, the `<model>.handle`, is used to communicate instances of model instantiation and the collection of solution values.

Solution Management. Only four new functions were needed to manage the asynchronous model instances: `HandleCollect` checks to see if the solution of a model instance has been completed. If a solution to the instance exists, it will be collected and merged into the existing GAMS database and solution specific model attributes are reset. `HandleStatus` returns the current status of the model instance without taking any further action. `HandleDelete` will attempt to remove the model instance from the system. `HandleSubmit` resubmits a model instance for solution. The outcome of the attempt to solve a model instance is stored in a GDX container, which gives independent access to all solution process related information, including the solution values if the process was successful. Instead of using the `HandleCollect` function to retrieve all solution values, the procedure `execute_loadhandle` has been added to collect all or parts of a solution.

Process Management. We may not be able to find the desired solution with just one GAMS process or submission. This could be because the solution process may have failures, may take days or weeks to complete, or has to interact with other external systems.This requires fail-safe design and time distributed processes. The existing `SAVE` and `RESTART` facilities, which allow GAMS processes to be interrupted and restarted at a later time, possibly on a different

computing platform, provides effective process management. The new GAMS process parameter `GRIDDIR` allows us to conveniently name a collection of model instance related information, usually the name of a node in a file system. This facilitates sharing of grid related information between different processes.

System Management. The actual implementation of the grid submission and management procedure is concentrated in one single script file, the `gmsgrid` script. Each GAMS system comes with a default `gmsgrid` script, which implements the grid facility for a given platform using standard operating system functionality. These scripts can easily be modified to adapt to different system environments or application needs. A GAMS user does not have to have any knowledge about the existence or content of those scripts.

4 Processing Independent Scenarios

The most immediate use of parallel solution is for the generation of independent scenarios arising in many practical applications. Monte Carlo simulations, scenario analyses and the tracing of efficiency frontiers are just a few examples. The modifications to the existing sequential GAMS code are minor and require no understanding of any platform specific features and no additional constraints are imposed on the application. We will illustrate this by using the model QMEANVAR from the GAMS Model Library. This model is used to restructure an investment portfolio using the traditional Markovitz style return variance tradeoffs under additional trading restrictions which make the model a mixed integer quadratic programming model, in GAMS classified as an MIQCP. Practical models of this kind can be very large, the scope for using advanced starting points is limited and the model instantiation time is very small compared to the solution time. The potential savings in elapsed time are then closely proportional to the numbers of processing nodes available which makes it already attractive on multi CPU systems.

Before one starts to convert an application from serial to parallel operation, it is important to verify that the parallel features are working as advertised. This can easily be accomplished by taking the existing application by setting the `<model>.solvelink` option to the value *4*. This will instruct GAMS to execute all the solve statements in serial mode, as before, but use the asynchronous solution strategy. This feature is also available via a GAMS process parameter which does not require any change to the GAMS program and is used extensively by our QA (Quality Assurance) processes. Once we have verified that our model and the required solvers can operate in asynchronous mode on the different target platforms, we are ready to parallelize the code.

As we have shown in the previous section, we need to separate the serial solution loop into a submission and collection loop. The original serial loop was:

```
Loop(p,
   ret.fx = rmin + (rmax-rmin)/(card(p)+1)*ord(p);
```

```
   Solve minvar min var using miqcp;
   xres(i,p)         = x.l(i);
   report(p,i,'inc') = xi.l(i);
   report(p,i,'dec') = xd.l(i) );
```

We just need to define a place where we store the solution handle and set the solution strategy for this model to *3*, which instructs GAMS to generate an instance of the model and submit this instance to the grid system for solution. Instead of saving the solution values, we just save the handle when we retrieve the solution.

```
parameter h(p) solution handle; minvar.solvelink=3;
Loop(p,
   ret.fx = rmin + (rmax-rmin)/(card(p)+1)*ord(p);
   Solve minvar min var using miqcp;
   h(p) = minvar.handle; );
```

In the following collection loop, we will also set a real time limit and mark the missed points.

```
Repeat
   loop(p$handlecollect(h(p)),
      xres(i,p)         = x.l(i);
      report(p,i,'inc') = xi.l(i);
      report(p,i,'dec') = xd.l(i);
      h(p) = 0 ) ;
   display$sleep(card(h)*0.1) 'sleep some time';
until card(h) = 0 or timeelapsed > maxtime;
xres(i,p)$h(p) = na;
```

The results of the model are now ready to be processed further by GAMS or passed on to some other system for further analysis and visualization. No other parts of the GAMS program had to be changed. The complete grid ready model is also available via the GAMS Model Library under the name QMEANVARG. There may remain one more question: Are the solutions from the serial model the same as from the parallel one? This kind of question arises in many situations when doing maintenance or enhancements of existing applications. It is simple to capture a snapshot of the current state of some or all the data items, model inputs and results, in a GDX container. The easiest way is to add the GAMS parameter GDX to the job submission for both versions. The two GDX containers can then be compared with a special difference utility, which produces a new GDX container which contains only the differences. Those differences can then be further processed or visualized in the GAMS IDE. It should be noted that there could be large differences in the equations and variables because the order in which solutions are retrieved is different, but all other items should be the same. In a shell environment this would look like:

```
gams qmeanvar  gdx=qmeanvar
gams qmeanvarg gdx=qmeanvarg
gdxdiff qmeanvar qmeanvarg qmeandiff RelEps=1e-12
```

It is typical for strategic modeling application to require the generation of large numbers of scenarios and their solutions need to be retained for further analysis. Furthermore, we may just want to submit a number of instances and disconnect from the system, analyze the results and prepare a new set of scenarios. This working style can be supported by splitting our GAMS code into several parts. The first one will contain the complete model and data definition and possibly the submission loop. Once all problems have been submitted, the program will terminate. In order to be able to inquire about the status of each instance and collect solutions, we need to save the GAMS environment for later restart and provide a known place where we can find the solution information. The GAMS parameters `SAVE` and `GRIDDIR` (or short, `S` and `GDIR`) need to be added when submitting the job. When using a Windows command shell it may look like:

```
> gams ... save=<sfile> gdir=<gdir>
```

The GAMS environment and the information in the grid directory are platform independent, for example, you can start submitting the jobs on a Windows platform and continue your analysis on a SUN Solaris system running on SPARC hardware. After some time we may want to check on the status of our work. This can be carried out by the following code:

```
parameter status(p,*); scalar handle;
acronym BadHandle,Waiting,Ready;
loop(p,
   handle := handlestatus(h(p));
   if(handle=0,
      handle := BadHandle
   elseif handle=2,
      handle := Ready;
      minvar.handle = h(p);
      execute_loadhandle minvar;
      status(p,'solvestat') = minvar.solvestat;
      status(p,'modelstat') = minvar.modelstat;
      status(p,'seconds') = minvar.resusd;
   else
      handle := Waiting );
   status(p,'status') = handle );
display status;
```

To run the above program we will have to restart from the previously saved environment and provide the location of the grid information. A job submission then may look like:

```
> gams ... restart=<sfile> gdir=<gdir>
```

The output of this run may then look like:

```
----   173 PARAMETER status

       solvestat   modelstat   seconds     status

p1         1.000       1.000     0.328      Ready
p2         1.000       1.000     0.171      Ready
p3                                          Waiting
p4                                          Waiting
p5         1.000       1.000     0.046      Ready
```

Once we are satisfied that a sufficient number of instances completed successfully, we are ready to execute the second part of our program using the same `RESTART` and `GDIR` values.

The model solutions retained are stored in GDX containers and can be operated on like any other GDX container. In large scale applications, it may not be feasible to merge all solution values and we only need to extract certain ones.

5 Decomposition Approaches

While it is clear that many model solution procedures can be enhanced via the use of *embarassingly parallel* techniques implemented on a grid computer, there are an enormous number of problems which take too long to solve in a serial environment and do not have such easy parallelization. Examples of such problems arise due to the sheer size of the problems, a requirement of global or stochastic optimization guarantees, or simply due to a combinatorial explosion in the search processes related to discrete choices.

In such cases, a modeler can resort to a problem decomposition approach. The GAMS language is rich enough to allow algorithms such as Benders decomposition, column generation, Jacobi and Gauss-Seidel iterative schemes for systems of equations, and even branch-and-bound algorithms to be coded directly. Many of these approaches generate subproblems that can be solved concurrently, and the grid extensions of GAMS enable these subproblems to be solved on the grid in such a manner.

In this section we give two examples of decompositions, namely an implementation of a Dantzig-Wolfe decomposition procedure for solving a multicommodity network flow problem, and an asynchronous Jacobi method for a system of equations.

5.1 Asynchronous Jacobi Method

As our first example of such a procedure, we outline how to implement various iterative schemes for the solution of systems of linear equations in a serial and

grid or parallel fashion. We choose this example for simplicity, but note that extension of this code to nonlinear equations or mixed complementarity systems is straightforward.

The following GAMS code snippet is the core of a simple problem which is solved as a mixed complementarity problem (MCP):

```
variables x(i); equations e(i); parameter A(i,j), b(i);

e(i)..  sum(j, A(i,j)*x(j)) =e= b(i);

model lin /e.x/;

b(i) = 1; A(i,i) = 1; A(i,j)$(not sameas(i,j)) = 0.001;

solve lin using mcp;
```

When the problem size becomes too large, we can use a partitioning scheme where the model domain is split into a collection of non-overlapping subdomains. A simple code to do this is outlined below, where we use the two dimensional sets `active` and `fixed` to denote those variables that are part of partition k, and those which should be treated as fixed in partition k.

```
sets k           problem partition blocks / block_1*block_%b% /
     active(k,i) active vars in partition k
     fixed(k,i)  fixed vars in partition k;
alias(kp,k);

active(k,i) = ceil(ord(i)*card(k)/card(i)) = ord(k);
fixed(k,i)  = not active(k,i);
```

A popular method for solving such systems is the Gauss-Seidel method, whereby the problem is split into several blocks, and each block problem is solved one after the other with the variables that are not in the block being fixed at their current values. An excellent reference for this method and the ones we outline below is given in [5].

```
parameter res(iters) sum of residual
          tol        convergence tolerance / 1e-3 /
          iter       iteration counter;

lin.solvelink = 2;   ! keep gams memory resident
lin.solprint  = 2;   ! suppress solution output
lin.holdfixed = 1;   ! treat fixed vars as constants

x.l(i) = 0; res(iters) = 0; res('iter0') = smax(i, abs(b(i)));

loop(iters$(res(iters) > tol),
```

```
  loop(k,
    x.fx(i)$fixed(k,i) = x.l(i);
    solve lin using mcp;
    x.lo(i)$fixed(k,i) = -inf;
    x.up(i)$fixed(k,i) =  inf );
  res(iters+1) = smax(i, abs(b(i) - sum(j, A(i,j)*x.l(j)))) );
```

Note several points here. The first is that since some variables are fixed, the holdfixed option of GAMS generates a model simply in the smaller *block* dimension space. Secondly, when the solution is read back into GAMS, a merge is performed and hence only the values of the variables that have been updated by the solver are changed.

This process, while it reduces the size of the problems being solved, may take large numbers of iterations to converge and is carried out in a serial fashion. For parallelization or grid solution, an even simpler technique is commonly used, and is typically referred to as a Jacobi scheme. In this setting, each block problem is solved *concurrently* with the variables that are not in the block being fixed at their current values. After all block subproblems are completed, the variables are updated simultaneously with the block solutions before the next iteration of the process is started.

```
parameter h(k) handles;

lin.solvelink = 3; ! set grid mode

x.l(i) = 0; res(iters) = 0; res('iter0') = smax(i, abs(b(i)));

loop(iters$(res(iters) > tol),

  loop(k,                              ! submitting loop
    x.fx(i)$fixed(k,i) = x.l(i);
    solve lin using mcp; h(k) = lin.handle;
    x.lo(i)$fixed(k,i) = -inf;
    x.up(i)$fixed(k,i) =  inf );

  repeat                               ! collection loop
     loop(k$handlecollect(h(k)),
       display$handledelete(h(k)) 'could not remove handle';
       h(k) = 0  );                    ! mark problem as solved
  until card(h) = 0;

  res(iters+1) = smax(i, abs(b(i) - sum(j, A(i,j)*x.l(j)))) );
```

Notice a couple of points here. The GAMS grid option is used to spawn all the block subproblems in the first loop. The second loop retrieves all the solutions, utilizing the built-in merge procedure of GAMS to overwrite the appropriate variable values. While such codes are easy to write with the extensions

described in this paper, it is typically the case that the Jacobi process is slow to converge (more so than the previous Gauss-Seidel scheme) since it does not use the most up-to-date information as soon as it is available.

For this reason, an asynchronous scheme is often preferred, and the GAMS grid option can be used to facilitate such a process very easily.

```
parameter curres intermediate residual values ;

lin.solvelink = 3;   ! set grid mode

x.l(i) = 0; res(iters) = 0; res('iter0') = smax(i, abs(b(i)));
iter = 0;

loop(k,  ! initial submission loop
  x.fx(i)$fixed(k,i) = x.l(i);
  solve lin using mcp;
  h(k) = lin.handle;
  x.lo(i)$fixed(k,i) = -inf;
  x.up(i)$fixed(k,i) =  inf );

repeat   ! retrieve and submit
  loop(k$handlecollect(h(k)),
     display$handledelete(h(k)) 'could not remove handle';
     h(k) = 0;
     iter = iter + 1;
     curres = smax(i, abs(b(i) - sum(j, A(i,j)*x.l(j))));
     res(iters)$(ord(iters) = iter + 1) = curres;
     if(curres > tol,
        loop(kp$(h(kp)=0 and
        smax(active(kp,i), abs(b(i) - sum(j, A(i,j)*x.l(j)))) > tol),
           x.fx(i)$fixed(kp,i) = x.l(i);
           solve lin using mcp;  ! submit new problem
           h(kp) = lin.handle;
           x.lo(i)$fixed(kp,i) = -inf;
           x.up(i)$fixed(kp,i) =  inf ) ) );
until card(h) = 0 or iter ge card(iters);
```

Note that all the block subproblems are spawned initially. After GAMS detects that a subproblem is solved, the results are retrieved and the residual of the system of equations is calculated. If this is not small enough, then each block subproblem that is not currently running, but for which the residual in this block is large, is spawned. Note that if a current block already has a small residual, a new subproblem is not spawned, but may be started at a later time when a different set of variables is updated. This process typically converges much faster than the Jacobi scheme, and in a parallel or grid environment outperforms the (serial) Gauss-Seidel scheme given above.

5.2 Dantzig Wolfe Decomposition

The second example is an implementation of the Dantzig-Wolfe decomposition method [16]. This method exploits structure in the linear program

$$\min c^T x \text{ subject to } Ax = b,\ x \geq 0$$

where for example A has the following form:

$$A = \begin{pmatrix} B_0 & B_1 & B_2 & \cdots & B_K \\ & A_1 & & & \\ & & A_2 & & \\ & & & \ddots & \\ & & & & A_K \end{pmatrix}.$$

Our specific implementation is in terms of a multi-commodity network flow problem (where each A_k corresponds to a commodity flow, and the joint constraints utilizing B_k represent arc capacities).

Using Minkowski's representation theorem [47], the feasible region of the problem can be expressed as a convex combination of extreme points and extreme rays. In the specific example above, we can write the extreme points and rays as a Cartesian product of extreme points and rays of a collection of *commodity sets* $\{x_k | A_k x_k = b_k, x_k \geq 0\}$. A (restricted) master problem finds the best combination of the existing subset of extreme points and rays, and a collection of subproblems generates new extreme points or rays by solving (in parallel) a pricing problem in each commodity based on dual information obtained from the master solution. In reality, the pricing subproblems correspond to evaluating the reduced costs of the full master problem and adding only those extra columns to the reduced master problem which are necessary.

The pertinent details of an implementation for the GAMS grid environment now follow:

```
scalar done  loop indicator /0/, iter /0/;
parameter h(k) model handles;

pricing.solvelink=3;

While(not done, iter=iter+1;
   done = 1;
   pricing.number = 0;
   loop(k,
     [set up data for kth pricing subproblem]
     solve pricing using lp minimizing z;
     h(k) = pricing.handle );

   repeat
     loop(k$h(k),
```

```
      if(handlecollect(h(k)),
         if ( z.l < -tol or iter eq 1,
           [add extreme point/ray from kth subproblem to master]
           done = 0;
         )
         h(k) = 0 ) ) ;
   until card(h) = 0;
   solve master using lp minimizing z;
);
```

The process starts by determining a set of columns to include in the master problem. Subsequently, columns are only added to the master problem if the reduced costs are negative. The master problem is solved to determine new multipliers on its constraints which thereby determine new pricing problems. The iterations of this method continue until no negative reduced cost is found. Note that for simplicity of exposition we have omitted the details of how the data for the kth subproblem is generated or how the solution of the kth subproblem is used to add an extreme point or ray to the (restricted) master problem.

A final observation concerns the line *pricing.number = 0*. This resets the solution counter and then generates the data of each pricing subproblem in the same grid directories for each iteration of the Dantzig-Wolfe procedure. This may reduce generation overhead, and future extensions of the GAMS grid procedure could exploit this further.

The complete examples from this section are available in the GAMS Model Library and were successfully run on all three of our grid engines, namely as background processes on a multiprocessor desktop, using the Sun Grid, or using the Condor resource manager.

6 Solving Intractable Mixed Integer Problems

MIP has been a proving ground for parallel computation for the last 15 years [33]. While most modern commercial MIP solvers support symmetric multiprocessing, or SMP, based on shared memory, the early '90s also saw implementations of the branch-and-bound (B&B) algorithm on distributed memory including academic codes (e.g. [19]) which lay the groundwork for the PICO solver [20] as well as commercial codes like parallel OSL on IBM SP2. The tree search in the B&B algorithm is a clear invitation for massive parallel processing. The master process farms out the relative expensive operation of solving the linear programs (LP) at the nodes to the workers. Even though communication standards like PVM or MPI have developed over time, no commercial MIP solver supports a distributed memory environment today. One reason for the failure of this simple parallelization scheme is the large volume of data communicated between master and worker compared to the relative short solution times of the LPs at the nodes.

A different kind of parallelization for MIP is based on a-priori decompositions of the solution space. In 2001, Ferris et. al. [25] applied a few rounds of manual

strong branching with a breadth-first-search node selection strategy to generate 256 subproblems of the Seymour problem. These 256 problems were solved in parallel with CPLEX 6.6.

There are various ways of decomposing the feasible region of a MIP with discrete variables $L_I \leq x_I \leq U_I$. Assuming $L_I = 0$ and $U_I = 1$ we can easily calculate the Hamming Distance from a reference point x'_I:

$$h := \sum_{i \in I, x'_i = 0} x_i + \sum_{i \in I, x'_i = 1} (1 - x_i)$$

Adding this linear constraint to the problem and forcing $h_j < h \leq h_j + 1$ for $j = 0, \ldots, k$ with $h_0 = -1$, $h_j < h_j + 1$, and $h_k = |I|$ decomposes the original problem into k+1 subproblems. Another simple decomposition scheme is based on splitting the domain of *important variables*. Here a small number of important discrete variables x_J with $J \subset I$ is selected and their bound range gets split into two regions:

$$L_J \leq x_J \leq \left\lfloor \frac{U_J - L_J}{2} \right\rfloor \quad \text{and} \quad \left\lceil \frac{U_J - L_J}{2} \right\rceil \leq x_J \leq U_J$$

The power set of the new regions results in a decomposition of $2^{|J|}$ subproblems.

The advantages of a-priori decomposition schemes are clear. The ratio between work and communication at the worker exceeds the one from simple parallel B&B algorithms. Off-the-shelf industrial strength MIP solvers can be used at the workers. Subproblems arising from bound tightening allow for additional round of MIP preprocessing which in general results in a tighter relaxation and faster solution times. The problem with a-priori decompositions is that the computational effort required solving the subproblems differs significantly. For example, we used the DICE [10, 30] model from the GAMS Model Library and partitioned the problem into 64 subproblems using the Important Variable and Hamming Distance decompositions. The figure below shows the solution time factor
Most of the
hat is almos
ming Dista

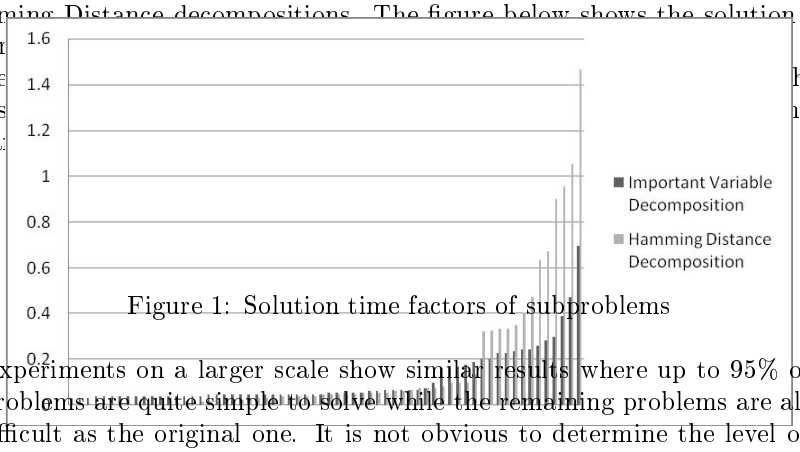

Figure 1: Solution time factors of subproblems

Experiments on a larger scale show similar results where up to 95% of the subproblems are quite simple to solve while the remaining problems are almost as difficult as the original one. It is not obvious to determine the level of difficulty of a subproblem prior to solving it. One way of ranking subproblems is to look at the value of their LP relaxation (possibly improved by extensive

MIP preprocessing and cut generation). The design of the branch-and-bound algorithm suggests that on average the closer the value of the LP relaxation of a subproblem is to the value of the root relaxation, the longer it takes to solve the subproblem. This measure has its limitations: For example, a subproblem could have an LP relaxation value close to one of the root relaxation, but the feasible region is extremely small so exploring the subproblem will quickly terminate. Nevertheless, computational experiments show that subproblem generation according to this measure produces decompositions with subproblems of similar but reduced level of difficulty compared to the original problem. Moreover, we can use a MIP solver to produce such decompositions without deeper knowledge of the problem itself and the importance of the discrete decision variables.

6.1 Decompositions by Branch-and-Bound

The branch-and-bound algorithm successively partitions the feasible region by branching on a discrete variable x_i with fractional value f in the LP relaxation. Two new subproblems are generated with new bounds on variable x_i: $x_i \leq \lfloor f \rfloor$ and $x_i \geq \lceil f \rceil$. One of the newly created subproblems or open nodes of the successively developing tree, which consist of the original problem plus tightened bounds on the discrete variables $L_I \leq L'_I < x_I \leq U'_I \leq U_I$, is selected and the LP relaxation is solved providing a) a fractional solution which leads potentially to further partitioning, b) an integer feasible solution, or c) an infeasible LP relaxation. In the latter two cases, the subproblem does not need to be explored further.

If during the process of the B&B algorithm an integer solution has been found, this can be used to stop partitioning of subproblems even if the LP relaxation contains discrete variables with fractional values (case a). If the objective value of the LP relaxation is not smaller than the best integer solution z_{incb} found so far, also known as the incumbent, an integer solution better than the incumbent cannot be found by this subproblem. Therefore, in the search for the optimal solution, the exploration of this subproblem can be stopped.

At any time during the B&B algorithm the set of open nodes $J = \{1, \ldots, k\}$ corresponds to a decomposition of the relevant unexplored feasible region of a mixed integer program. The minimum of the incumbent and the optimum objective value of all subproblems defined by the open nodes J determines the optimal solution of the original problem: $z_{\text{opt}} = \min(z_{\text{incb}}, z^1_{\text{opt}}, z^2_{\text{opt}}, \ldots, z^k_{\text{opt}})$ where z^j_{opt} is the optimum objective value of the subproblem corresponding to the jth open node.

MIP solvers like COIN-OR's CBC, CPLEX, and XPRESS provide strategies for variable branching (e.g. strong branching) and node selection (e.g. best bound) that focus on reducing the largest LP relaxation value of all open nodes. This value is also known as the *best bound* or *best estimate*. Decompositions based on open nodes from B&B trees developed by such strategies tend to produce subproblems with similar LP relaxation values that are significantly reduced from the initial root relaxation value. Following the suggested relation

between the level of difficulty and the deviation of subproblem LP relaxation values from the root, the scheme will produce subproblems of equal but reduced level of difficulty. Figure 2 shows for model DICE the relative difference of the LP values at 16 open nodes from the root node LP value with different settings for CPLEX parameter `mipemphasis`.

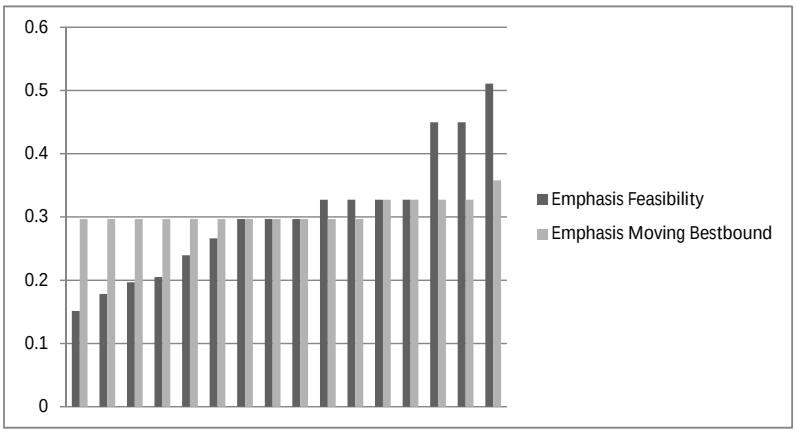

Figure 2: Relative distance of LP at node from root LP

Using the approach described above, we can use any MIP solver for automatic generation of a-priori decompositions which provides tree development strategies that focus on moving the best bound and that give access to the open nodes during the B&B algorithm. Depending on the number of available machines in the grid we can stop the B&B algorithm as soon as the number of open nodes reaches a specified number n resulting in n subproblems.

6.2 Implementation Details

The following work has been implemented using GAMS/CPLEX but as outlined in previous sections, a very similar implementation could have been done with COIN-OR's CBC or XPRESS.

The GAMS/CPLEX interface supports the Branch-and-Cut-and-Heuristic facility (BCH) [11] allowing GAMS to supply user cuts and incumbents to the running branch-and-cut algorithm inside CPLEX using CPLEX callback functions. A minor extension to this facility resulted in the GAMS/CPLEX option `dumptree`, providing the tightened bounds of discrete variables for each of the subproblems corresponding to the open nodes as soon as their number reached the specified value n. If we want to further process the subproblems on the GAMS language level, the variables and their tightened bound values need to be provided in the namespace of the original GAMS model and not in the internal namespace of CPLEX (usually $x_1, x_2, \ldots$). When GAMS generates a model to be passed to a solver, a dictionary is created that allows the mapping from the solver variable and constraint space to the GAMS namespace and vice versa.

The GAMS/CPLEX option `dumptree` stores the tightened bounds of the subproblems for offline processing in separate GDX containers using the dictionary to determine the original GAMS namespace.

After an initial *solve* of the original problem using GAMS/CPLEX with option `dumptree` n, at most[1] n subproblems by the individual GDX bound containers are available for submission to the grid. The submission and collection is similar to the first example. Here the data for different scenarios/subproblems does not come from some external data source, but gets loaded from the GDX containers.

6.3 Need for Communication between Jobs

The n submitted jobs run completely independent without communication before completion. Unlike the first example, we are not necessarily interested in optimal solutions of all n subproblems. We *just* need the best solution of all n subproblems. If we can determine that a subproblem will not provide the best solution, we can terminate the job before it finishes regularly. Assume the minimum of the incumbents of all (running) jobs is $z_{\text{incb}} = \min_{i=1\ldots n} z^i_{\text{incb}}$. Following the arguments from the description of the B&B algorithm we can terminate all subproblems for which the best bound is not smaller than z_{incb}.

The BCH facility can be configured so CPLEX calls out to a user GAMS program whenever a new incumbent is found. In our case, the GAMS program communicates the new incumbent z^i_{incb} of subproblem i to the master job which is the process of collecting results from the subproblems. If this new incumbent is better than the current best incumbent z_{incb} of all subproblems, z_{incb} is updated and communicated to all other subproblems. The value z_{incb} is used as a cutoff value instructing the solver to stop processing nodes whose LP relaxation value is larger than z_{incb}. If the best bound of a subproblem is larger than z_{incb}, processing of all open nodes will be stopped and the job will terminate immediately.

The updated cutoff value is communicated through a GAMS/CPLEX option file. The running GAMS jobs frequently look for a *trigger file*. These trigger files are written by the master process whenever a better incumbent has been found and initiate reading of the GAMS/CPLEX option file with the updated cutoff value. After a job has read and updated the internal cutoff value, it deletes its trigger file to prevent reading the same cutoff value repeatedly. This type of file based communication is straightforward to implement on grid systems with a shared file system. While the Condor system supports communication via a shared file system, this requirement limits the number of available machines significantly since Condor runs on heterogeneous networks not sharing a file system potentially across the world. Condor can mimic a shared file system

[1] Before the subproblem gets dumped to a GDX container, CPLEX will solve the LP relaxation of the subproblem. In case the node is integer feasible or infeasible which would give a trivial subproblem, the dumping of the GDX bound container is skipped resulting potentially in less than n subproblems.

using Condor Chirp which ensures the distribution of files between the master and the workers.

In the introduction, we discussed some specifics of the Condor system. Not all machines in the Condor pool at University of Wisconsin are dedicated machines, some of them are workstations. The Condor system ensures that the interactive user of the workstation does not suffer performance loss from background processes by vacating Condor jobs as soon as a user starts an interactive session. Condor supports check-pointing of jobs meaning that a job frequently creates checkpoint files that allow restarting from the last checkpoint and hence minimizing wasted CPU time in case of a vacated job. Unfortunately, Condor's check-pointing require the use of special link libraries which are incompatible with some third party software (CPLEX in our case). Hence, a vacated GAMS/CPLEX job needs to be started from scratch resulting in a significant amount of wasted CPU time.

6.4 MIP Strategy and Repartitioning

The branch-and-bound algorithm works towards closing the gap between the incumbent and the best bound. Different variable and node selections (among other strategies like cut generation, use of primal heuristics, etc) emphasize the improvement of the incumbent or improvement of the best bound. While a sequential MIP solver needs to balance its computational emphasis for moving the incumbent and the best bound, this is much easier in a parallel setting. In addition to solving the n subproblems for which the MIP solver focuses on moving the best bound, we have one additional job solving the original problem with heavy emphasis on finding good incumbents. GAMS/CPLEX can be easily instructed to place emphasis on improving the incumbent or the best bound by using the CPLEX meta-option `mipemphasis 1` (feasibility) or `mipemphasis 3` (best bound).

In some harder cases, the partitioning strategy may become inefficient on the grid machine in that most of the partitioned jobs have completed their execution, but a very small number of jobs continue to have a nonzero gap. In this case, the job is repartitioned using the `dumptree` option again, and the new jobs are added to the list of outstanding tasks. These new tasks either augment the pool of work that needs to be processed, or they replace the job that was running. In some cases, the repartitioning can significantly reduce the overall computational time. An outstanding issue is to determine when a repartition should be carried out. One trigger for this could simply be time (where a fixed time limit is set for each job), and another could be the number of remaining jobs compared to the size of the available grid. In the latter case, we could prioritize the jobs for repartitioning based on the current value of the gap. An implementation in GAMS would require similar features to those we use to write out an incumbent solution. An example of the use of this process is given in [26].

6.5 Numerical Results

All the techniques described in the previous sections have been implemented for the DICE model in a new model called DICEGRID available from the GAMS Model Library. While the DICEGRID model is of educational and reference value, this approach has also been used on a set of very difficult MIP problems. A source of such MIP problems which are publically available is MIPLIB [7, 8]. The computational experiments were carried out in 2006 on the Condor pool at University of Wisconsin, on four problems from the latest version of MIPLIB 2003 [1] that were unsolved at the time: A1C1S1, ROLL30000, SWATH, and TIMTAB2. The following table contains relevant figures:

	A1C1S1	ROLL30000	SWATH	TIMTAB2
Optimal Solution	11503.4	12890	*467.407	1096557
Number of subproblems	1089	986	1001	3320
Maximum CPU time of individual job (h)	15.9[2]	1.2	-	153.2
Wall clock time (h)	13.5[2]	29.7	424.0	71.5
Total CPU Time (h)	3700.3	50.9	8135.1	2744.7
Fraction of wasted CPU time	6.7%	0.6%	42.8%	13.1%
CPLEX B&B Nodes	1921736	400034	22458649	17092215

* Not solved to optimality

While the problems A1C1S1 and ROLL30000 solved with no modification of our described approach, the TIMTAB2 problem required some additional help. For TIMTAB2 we generated 3320 subproblems and solved these jobs over a period of three days using (at times) over 500 processing units. These units included both Linux and Windows machines, some of which had a shared file system and some of which did not. Not only was this problem solved to optimality, but a new solution (of value 1096557) was generated. The solution required not only the large computational resources from Condor, but also a collection of problem specific cuts generated by colleagues in Berlin[39] for these types of problems. It is important to notice that problem specific expertise, coupled with large amounts of computing resources facilitated this solution.

The standard scheme failed to solve the SWATH problem. After almost a year of cumulated CPU time, there were still 538 of the 1001 problems unfinished. Even after repartitioning and adding user defined cuts, the problem remained unsolved after over 17 years of CPU time of which 2/3 was wasted, exploring over 1.2 billion nodes in about 4 months of wall clock time. However, by understanding the problem structure (SWATH is a 20 node generalized traveling salesman problem with super-nodes involving additional constraints; see the GAMS Model Library for the original and improved formulation) and generating 4 rounds of subtour elimination constraints (resulting in 22 additional

[2]The wall clock and maximum CPU time reported for A1C1S1 are inconsistent. Unfortunately, the jobs are performed on machines that are not centrally maintained and are on different clocks. Therefore slight (timing) inconsistencies can occur in such a diverse computing environment.

cuts) the problem was solved to optimality within seconds. Such experience cautions the applicability of pure *brute-force* methods consuming large amounts of computational resources, and ensures that further work indentifying particular structures and problem specific enhancements is imperative.

6.6 Portability of Models

In August 2007, GAMS Development Corporation and Sun Microsystems teamed to provide GAMS users access to the commercial grid computing facility at Sun's Network.com. We repeated the experiments with the simplest of the four problems ROLL3000 in this environment without the need for changing a single line of GAMS code. We decomposed the MIPs into 256 subproblems (rather than 1000 as for the Condor pool). ROLL3000 finished in less than an hour wall clock time consuming less than 8 hours CPU time. The unchanged model for ROLL3000 was also solved on a four core Sun Sparc Solaris workstation. The scheduling of the 256 jobs was entirely left to the operating system. ROLL3000 was solved in about two hours wall clock time utilizing about 6 CPU hours on the 4 processors. The large difference in computational effort for ROLL3000 between the experiments on the Condor pool and the Sun Grid and workstation can be mainly attributed to two facts. In 2006 on the Condor pool we used CPLEX 9 while the more recent experiments on the Sun Grid and the workstation used CPLEX 10. Secondly, the individual machines in the Sun Grid (and the same holds for the four core workstation[3]) are uniformly equipped with high powered CPUs and sufficient memory[4]. The quality of the machines in the Condor pool is diverse and on average significantly worse compared to the other two computing environments.

7 Conclusions

In this paper we have shown a number of ways to harness the power of a computational grid for the solution of optimization problems that are formulated in a modeling language. The paper describes GAMS grid, a lightweight, portable, powerful set of extensions of GAMS specifically for managing optimization solution strategies on a grid.

The paper includes a number of expository examples decribing the use of these features to implement both parallel algorithms and distributed solution approaches within a number of important application areas. These examples (QMEANVARG, JOBOBI, DANWOLFE, DICEGRID, SWATH) are all available in the GAMS Model Library for download [29]. We believe the simplicity and generality of the framework lends itself well to a large number of business and research problems. The design facilitates the use of all solvers and model types currently available within GAMS on a grid engine.

[3]Sun Fire X2200 M2 Server with two 2218 Processor (Dual-Core) with 16 GB of RAM.

[4]The Sun Grid consists of Sun Fire dual processor Opteron-based solvers with 4 GB of RAM per CPU.

The framework uses a master/worker computational model that we believe is sufficiently scalable and flexible for many optimization algorithms and applications. It is already available in current releases of GAMS, and demonstrably useful for hard optimization problems. As particular instantiations of these features, we have described the use of GAMS grid and the CPLEX solver in conjunction with grid facilities provided by the Condor resource manager or the Sun Grid Engine to solve MIP problems that have evaded solution by other means.

References

[1] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):1–12, 2006.

[2] E. Alba. *Parallel Metaheuristics: A New Class of Algorithms.* Wiley-Interscience, 2005.

[3] K. M. Anstreicher, N. W. Brixius, J.-P. Goux, and J. Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91:563–588, 2002.

[4] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica*, Extra Volume Proceedings ICM III (1998):645–656, 1998.

[5] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods.* Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, 1989.

[6] J.J. Bisschop and A. Meeraus. On the development of a general algebraic modeling system in a strategic planning environment. *Mathematical Programming Study*, 20:1–29, 1982.

[7] R. E. Bixby, E. A. Boyd, and R. R. Indovina. MIPLIB: A test set of mixed integer programming problems. *SIAM News*, 25:16, 1992.

[8] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, 58:12–15, June 1998.

[9] R. E. Bixby, W. Cook, A. Cox, and E. K. Lee. Computational experience with parallel mixed integer programming in a distributed environment. *Annals of Operations Research (Parallel Optimization)*, 90:19–43, 1997.

[10] R. A. Bosch. Mindsharpener. *Optima*, 70:8–9, 2003.

[11] M. R. Bussieck and A. Meeraus. Algebraic modeling for IP and MIP (GAMS). *Annals of Operations Research*, 149(1):49–56, 2007.

[12] D. Butnariu, Y. Censor, and S. Reich. *Inherently Parallel Algorithms in Feasibility and Optimization and their Applications*. Elsevier Science, Amsterdam, The Netherlands, 2001.

[13] Y. Censor and S. A. Zenios. *Parallel Optimization: Theory, Algorithms and Applications*. Oxford University Press, 1997.

[14] Q. Chen and M. C. Ferris. Fatcop: A fault tolerant condor-pvm mixed integer programming solver. *SIAM J. on Optimization*, 11(4):1019–1036, 2000.

[15] Q. Chen, M. C. Ferris, and J. T. Linderoth. Fatcop 2.0: Advanced features in an opportunistic mixed integer programming solver. *Annals of Operations Research*, 103:17–32, 2001.

[16] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8:101–111, 1960.

[17] Dash Optimization. XPRESS optimizer. `http://www.dashoptimization.com/`.

[18] J. E. Dennis, Jr. and V. Torczon. Direct search methods on parallel machines. *SIAM Journal of Optimization*, 1(4):448–474, 1991.

[19] J. Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. *SIAM Journal on Optimization*, 4:794–814, 1994.

[20] J. Eckstein, C. A. Phillips, and W. E. Hart. PICO: An object-oriented framework for parallel branch and b ound. In *Proceedings of the Workshop on Inherently Parallel Algorithms in Optimization and Feasibility and their Applications*, Studies in Computational Mathematics, pages 219–265. Elsevier Scientific, 2001.

[21] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors: Load sharing among workstation clusters. *Future Generation Computer Systems*, 12:53–65, 1996.

[22] M. C. Ferris and O. L. Mangasarian. Parallel constraint distribution. *SIAM Journal on Optimization*, 1(4):487–500, November 1991.

[23] M. C. Ferris and O. L. Mangasarian. Parallel variable distribution. *SIAM Journal on Optimization*, 4(4):815–832, November 1994.

[24] M. C. Ferris and T. S. Munson. Modeling languages and condor: Metacomputing for optimization. *Mathematical Programming*, 88:487–506, 2000.

[25] M. C. Ferris, G. Pataki, and S. Schmieta. Solving the Seymour problem. *Optima*, 66:1–7, October 2001.

[26] M. C. Ferris, A. Sundaramoorthy, and C. T. Maravelias. Simultaneous batching and scheduling using dynamic decomposition on a grid. Technical report, Computer Sciences Department, University of Wisconsin, 2007.

[27] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[28] R. Fourer, D. M. Gay, and B. W. Kernighan. A modeling language for mathematical programming. *Manage. Sci.*, 36(5):519–554, 1990.

[29] GAMS Development Corp. The GAMS model library. `http://www.gams.com/`.

[30] M. Gardner. *The Colossal Book of Mathematics*. WV Norton, 2001.

[31] A. Geist, A. Beguelin, Jack Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994.

[32] B. Gendron. Parallel computing in combinatorial optimization: List of references. `http://www.iro.umontreal.ca/~gendron/Pisa/References.htm`.

[33] B. Gendron and T. G. Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1060, 1994.

[34] The Globus Alliance. The Globus toolkit. `http://www.globus.org/`.

[35] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, January 1989.

[36] A. Grama and V. Kumar. A survey of parallel search algorithms for discrete optimization problems. *INFORMS Journal on Computing*, 7:365–385, 1995.

[37] A. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *Knowledge and Data Engineering*, 11(1):28–35, 1999.

[38] ILOG CPLEX Division. CPLEX optimizer. `http://www.cplex.com/`.

[39] Christian Liebchen. personal communication, 2006.

[40] J. Linderoth, S. Kulkarni, J.-P. Goux, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, pages 43–50, Pittsburgh, PA, August 2000.

[41] J. T. Linderoth, E. K. Lee, and M. W. P. Savelsbergh. A parallel, linear programming-based heuristic for large-scale set partitioning problems. *INFORMS J. on Computing*, 13(3):191–209, 2001.

[42] J. T. Linderoth, A. Shapiro, and S. J. Wright. The empirical behavior of sampling methods for stochastic programming. *Annals of Operations Research*, 142:219–245, 2006.

[43] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *ICDCS*, pages 104–111, 1988.

[44] M. Livny. PI, the condor project, high throughput computing. `http://www.cs.wisc.edu/condor`.

[45] M. Livny and R. Raman. High-throughput resource management. In *The grid: blueprint for a new computing infrastructure*, pages 311–337. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[46] MOSEK ApS. The MOSEK optimization software. `http://www.mosek.com/`.

[47] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, NY, 1988.

[48] B. Norris, S. Balay, S. J. Benson, L. Freitag, P. Hovland, L. C. McInnes, and B. Smith. Parallel components for PDEs and optimization: Some issues and experiences. *Parallel Computing*, 28(12):1811–1831, 2002.

[49] J. Pruyne and M. Livny. Providing resource management services to parallel applications. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, Townsend, TN, May 1994.

[50] T.K. Ralphs, L. Ladányi, and M.J. Saltzman. Parallel branch, cut, and price for large-scale discrete optimization. *Mathematical Programming*, 98:253–280, 2003.

[51] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A network based information library for global world-wide computing infrastructure. In *HPCN Europe*, pages 491–502, 1997.

[52] SUN Microsystems. Sun grid compute utility. `http://www.network.com/`.

[53] Sunset Software Technology. XA optimizer system. `http://www.sunsetsoft.com/`.

[54] S. Wright. Solving optimization problems on computational grids. *Optima*, 2001.