ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

NEOS AND CONDOR: SOLVING OPTIMIZATION PROBLEMS OVER THE INTERNET

Michael C. Ferris, Michael P. Mesnier, and Jorge J. Moré

Mathematics and Computer Science Division

Preprint ANL/MCS-P708-0398

March 1998

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the National Science Foundation under Grants CDA-9726385 and CCR-9619765, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

NEOS AND CONDOR: SOLVING OPTIMIZATION PROBLEMS OVER THE INTERNET*

Michael C. Ferris†, Michael P. Mesnier‡, Jorge J. Moré§

Abstract

We discuss the use of Condor, a distributed resource management system, as a provider of computational resources for NEOS, an environment for solving optimization problems over the Internet. We also describe how problems are submitted and processed by NEOS, and then scheduled and solved by Condor on available (idle) workstations.

1 Introduction

The NEOS Server [8] is a novel environment for solving optimization problems over the Internet. There is no need to download an optimization solver, write code to call the optimization solver, or compute derivatives for nonlinear problems. NEOS provides the user with the input format and a list of solvers for the optimization problem. Given an optimization problem, NEOS solvers compute derivatives and sparsity patterns of nonlinear problems with automatic differentiation tools, link with the appropriate libraries, and execute the resulting binary. The user is provided with a solution and runtime statistics.

Each solver in the NEOS optimization library is maintained by a software administrator that is responsible for providing computing resources and for answering questions related to the solver. Registering the solver [8] on a few workstations provides adequate resources in most cases, but for large problems, however, we need a different approach. The obvious difficulty is that the owner of a workstation is reluctant to provide large amounts of computing cycles and memory. We use Condor [15, 11], a distributed resource management system, as a provider of computational resources for a NEOS solver. The resources that are managed by Condor are typically large clusters of workstations, many of which would otherwise be idle for long periods of time.

We discuss the connection between NEOS and Condor in the context of a single but important optimization problem: mixed complementarity problems. Our discussion shows that this approach can be extended to other problems as well.

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the National Science Foundation under Grants CDA-9726385 and CCR-9619765, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

†Computer Sciences Department, University of Wisconsin – Madison, 1210 West Dayton St., Madison, Wisconsin 53706. `ferris@cs.wisc.edu`

‡Department of Computer Science, University of Illinois, 1304 W. Springfield, Urbana, Illinois, 61801. `mesnier@cs.uiuc.edu`

§Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439. `more@mcs.anl.gov`

Many different applications can be formulated as mixed complementarity problems; examples are given in [9, 13]. If the nonlinear function $F : \mathbb{R}^n \mapsto \mathbb{R}^n$ describes the interactions of a nonlinear process as a function of the variables $x \in \mathbb{R}^n$, then the mixed complementarity problem is to find a vector x, with components between lower and upper bounds ℓ and u (with $\ell < u$), such that

$$\begin{array}{lll} F_i(x) = 0 & \text{if} & \ell_i < x_i < u_i, \\ F_i(x) \geq 0 & \text{if} & x_i = \ell_i, \\ F_i(x) \leq 0 & \text{if} & x_i = u_i. \end{array} \tag{1.1}$$

Solving a mixed complementarity problem in the typical computational environment requires that the user first develop code for the evaluation of F. The user must then decide on an appropriate solver, retrieve the solver, develop code to evaluate the Jacobian matrix $F'(x)$ and sparsity pattern of $F'(x)$, link their code with the necessary libraries, and finally execute the solver locally. With NEOS, the user need only specify the mixed complementarity problem by providing code to evaluate the function F, the lower and upper bounds ℓ and u, and a starting point. The NEOS solver then generates code to compute the Jacobian matrix and sparsity pattern, compiles the user subroutines, links with the appropriate libraries, executes the solver on a NEOS machine, and returns a solution to the user.

NEOS uses the Condor pool at the University of Wisconsin for solving complementarity problems. The pairing of NEOS with Condor is an ideal combination. NEOS provides an interface that is problem oriented and independent of the computing resources. Users need only provide a specification of the problem; all other information needed to solve the problem is determined by the NEOS solver. Condor provides the computational resources to solve the problem.

Condor acts as a matchmaker, pairing computational resources with jobs that require processing. The job executes on the allocated workstation until completion or until the workstation becomes unavailable. In the latter case, the job is frozen in its current state and the workstation is returned to the owner. Condor is then contacted once again for pairing and the job is restarted from its frozen state on the newly allocated resource. Condor pays special attention to the needs of the workstation owner by allowing the owner to define the conditions under which the workstation can be allocated. This policy encourages workstation owners to place their resources in the Condor pool, and as a consequence, the Wisconsin pool currently has over 400 workstations.

In Section 2 we describe the three interfaces that are currently available to submit problems to NEOS: e-mail, the NEOS Submission Tool (`neos-submit`), and the NEOS Web interface. These interfaces are designed so that problem submission is intuitive and requires only essential information. Parameters that affect the progress of the solver are not required but can be specified, for example, by an auxiliary file. We concentrate on the NEOS Submission Tool. The NEOS Web interface can be sampled by visiting the URL

for the NEOS Server. We emphasize mixed complementarity problems, but NEOS handles a wide variety of linear and nonlinearly constrained optimization problems; solvers for optimization problems subject to integer variables are being added. We do not discuss the design and implementation of the Server because these issues are covered by Czyzyk, Mesnier, and Moré [8]. Extensions to the NEOS Server and the network computing issues that arise from the emerging style of computing used by NEOS are discussed by Gropp and Moré [14].

Mixed complementarity problems submitted to the NEOS Server are currently solved by the PATH [10, 12] solver, which implements a Newton-type method for solving systems of non-differentiable equations. Sparse matrix techniques are used for large problems. The process used to solve a nonlinear complementarity problem by this NEOS solver includes the generation of derivative and sparsity information with the ADIFOR [4, 5] automatic differentiation tool and the solution of the problem with Condor. The process is governed by a *solver script* that must check the user data and provide appropriate messages in the case of errors. In Section 3 we describe the various issues that must be addressed by the solver script. These issues are important to the development of reliable optimization software and problem-solving environments.

The automatic differentiation techniques used to generate derivatives and sparsity patterns for nonlinear complementarity problems are described in Section 4. In particular, we explain how to obtain a sparse representation of the Jacobian matrix that is suitable for PATH. The Jacobian matrix generated by ADIFOR is accurate to full machine precision, while the Jacobian matrix generated by differences of function values suffers from truncation errors. Moreover, the code produced by ADIFOR for the computation of the sparse Jacobian matrix is typically more efficient than the code produced by differences. On the other hand, the code produced by ADIFOR may not be as efficient as a hand-coded Jacobian matrix. See [1] for a full comparison (in terms of memory and speed) of ADIFOR-generated Jacobian matrices with both hand-coded and difference approximations, and [2] and [7] for performance issues related to the automatic computation of gradients.

The automatic generation of the Jacobian matrix and sparsity pattern in the NEOS version of PATH makes the code more accessible and useful than requiring the hand-coding of the Jacobian matrix. Indeed, all nonlinear solvers in NEOS use automatic differentiation tools to compute gradients, Jacobians, and sparsity patterns. We intend to incorporate ADIC [6] into most of the nonlinear NEOS solvers to allow problems to be specified in C, as well as Fortran.

The final section of the paper describes how the Condor system at the University of Wisconsin is used to process the submitted jobs. Only large jobs are scheduled on Condor because there may be a delay in execution while waiting for an idle workstation. Small

jobs are executed immediately on non-clustered workstations. No computational results are given here. Users are encouraged either to submit one of the supplied sample problems or to generate new mixed complementarity problems to test the system.

2 The NEOS Server

The NEOS Server provides Internet access to a library of optimization solvers with user interfaces that abstract the user from the details of the optimization software. The user needs only to describe the optimization problem in a particular format; all additional information required by the optimization solver is determined automatically. This abstraction is similar to that provided by modeling languages. The NEOS solvers provide several different input formats to allow users to specify optimization problems in a convenient manner without necessarily rewriting their problem in a modeling language.

The NEOS approach offers considerable advantages over a conventional environment for solving optimization problems. Consider, for example, mixed complementarity problems. A NEOS solver for mixed complementarity problems requires that the user specify the number of variables `n`, a subroutine `initpt(n,x)` that defines the starting point, a subroutine `xbound(n,xl,xu)` that sets the lower and upper bounds, and a subroutine `fcn(n,x,f)` that evaluates the function F. Since there is no need to provide the Jacobian matrix or the sparsity pattern of the Jacobian matrix, the user can concentrate on the specification of the problem. Changes to the `fcn` subroutine can be made and tested immediately; the advantages in terms of ease of use are considerable.

Other optimization problems can be specified in a similar manner. For example, the nonlinearly constrained optimization problem

$$\min \{f(x) : x_l \leq x \leq x_u,\ c_l \leq c(x) \leq c_u\}$$

can be specified by four subroutines. The bounds x_l and x_u are specified with the subroutine `xbound(n,xl,xu)`, the constraint bounds c_l and c_u are specified with the subroutine `cbound(m,cl,cu)`, the objective function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is defined by the subroutine `fcn(n,x,f)`, and the nonlinear function $c : \mathbb{R}^n \mapsto \mathbb{R}^m$ is defined by `cfcn(m,x,c)`.

We have mentioned nonlinear optimization solvers, but NEOS contains solvers in other areas. A complete listing is available at the NEOS Server homepage:

`http://www.mcs.anl.gov/otc/Server/`

The addition of solvers is not difficult. Indeed, as discussed in [8], NEOS was designed so that solvers in a wide variety of optimization areas can be added easily.

We provide Internet users the choice of three interfaces for submitting problems: e-mail, the NEOS Submission Tool, and the NEOS Web interface. These interfaces are designed so that problem submission is intuitive and requires the minimal amount of information.

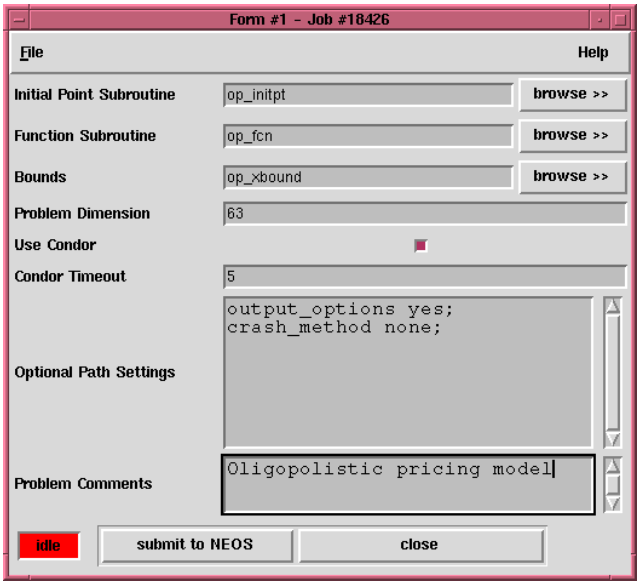

Figure 2.1: The NEOS submission form for PATH

The interfaces differ only in the way that information is specified and passed to the NEOS Server.

The e-mail interface is relatively primitive, but useful because most users have easy access to e-mail. Information on the available solvers and on the format used to submit problems via e-mail can be obtained by sending the mail message help to

neos@mcs.anl.gov

Users interested in the Web interface should visit the homepage for the NEOS Server, which has links to all the solvers in the library, as well as pointers to other NEOS information, in particular, the NEOS Guide. In the remainder of this section we examine the NEOS Submission Tool.

The NEOS Submission Tool provides a high-speed link to the NEOS Server via TCP/IP sockets. Once this tool is installed (only Perl [17] is required), the user has access to all solvers offered by NEOS. Additional information on the NEOS Submission Tool, including installation instructions, can be obtained from the NEOS Server homepage.

Submission of problems via the NEOS Submission Tool is simple. The user must first choose the type of optimization problem and then select the desired solver. Once the solver is selected, the user is given a submission form specific to the solver.

The PATH submission form, shown in Figure 2.1, requires that the user specify the number of variables, the files for the initial point, bounds on the variables, and function evaluation subroutines.

Figure 2.1 shows the NEOS Submission form for a model of oligopolistic pricing [16]

with 63 variables. The model is defined by the file `op_fcn`, while the initial point and the bounds on the problem are defined by the files `op_initpt` and `op_xbound`. Note that in this case two PATH options are set and that we have requested the use of Condor for the solution of this problem.

The user has the option of using a Condor pool of workstations for solving the submitted problem. Since Condor is essentially a batch processing mechanism, the user is also allowed to specify a timeout for Condor. After this period of time, the Web browser or Submission Tool is released from its busy state and returned to the user. The job, however, continues to process, and the results are returned to the user at a later date via e-mail. The default timeout is 5 minutes; Condor is used by default on all problems that are larger than 500 variables.

The PATH submission form allows the user to provide a specification file that can be used to set tolerances and other parameters that govern the algorithm. For most problems the defaults provided are adequate. Figure 2.1 shows two options in use for this submission. The first provides a listing of the current settings of all the available options for this run; the second just turns off the default crash technique. The form also has room for comments, which can be used to identify the problem submission.

Once specified, the problem is submitted to NEOS where it is then scheduled for execution. A variety of computers, even a massively parallel processor, could be used to solve the problem. At present these computers are workstations that reside at Argonne National Laboratory, Northwestern University, the University of Wisconsin, Lawrence Berkeley National Laboratory, the Technical University of Ilmenau in Germany, and Arizona State University.

3 Solving Complementarity Problems: PATH

The process used to solve a nonlinear complementarity problem by NEOS is illustrated in Figure 3.1. This process includes the generation of derivative and sparsity information with the ADIFOR [5, 4] automatic differentiation tool and the solution of the problem with the Condor [15, 11] distributed resource management system. We discuss ADIFOR further in Section 4, while Condor is discussed in Section 5. In this section we discuss issues in the solution process that are pertinent to the development of optimization software and problem-solving environments. Although the discussion is specific to PATH [10, 12], most of the issues are applicable to all the solvers of nonlinear optimization problems in NEOS.

Submitting a problem to the NEOS Server does not guarantee success, but NEOS users are able to solve difficult optimization problems without worrying about many of the details that are typical in a conventional computing environment. Even if the user has suitable optimization software, the user would need to read the documentation, write code to interface his problem with the optimization software, and then debug this code. The user would also have to write code to evaluate the Jacobian matrix and sparsity pattern, and debug that

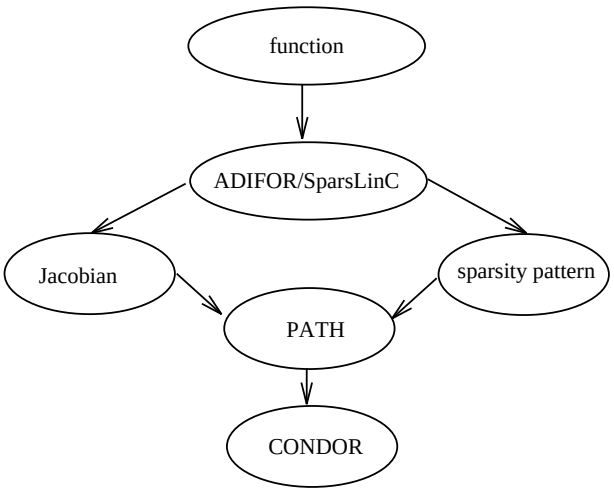

Figure 3.1: PATH and Condor

code—a nontrivial undertaking in most cases.

For a typical submission, the user receives information on the progress, and the solution. Figure 3.2 shows part of the output received when the problem in Figure 2.1 is submitted to NEOS via the NEOS Submission Tool. In particular, we see the NEOS server selecting an available workstation, transferring all user data to the workstation, and then invoking the solver remotely. The solver (in this case PATH) checks the data and compiles the user's code. If any errors are found at this stage, the compiler error messages are returned to the user, and execution terminates.

If the user's code compiles correctly, the automatic differentiation tool ADIFOR [5, 4] is used to generate the Jacobian matrix and the sparsity pattern. Additional details on this part of the process are discussed in Section 4. Once the Jacobian matrix and sparsity pattern are obtained, the user's code is linked with the optimization libraries, and execution begins. Results are returned in the window generated by the NEOS Submission Tool.

The solver script that handles the solution process must check the input data to make sure that the job submission is valid. A typical error at this stage of the solution process is for the user to interchange files and to send, for example, subroutine `initpt` where NEOS is expecting subroutine `xbound`. A similar error is to neglect to send one of the files required for the job submission. These errors are detected by the solver script by checking that the files that specify `fcn`, `initpt`, and `xbound` exist and that they reference the appropriate subroutine. The solver script also checks that the problem dimensions are positive.

Even if the supplied subroutines compile correctly, ADIFOR may find an error during the generation of the `fcn` subroutine. The most common error here is for the submission to contain an improper calling sequence. For example, if the calling sequence of the submitted `fcn` is `fcn(n,x,y)`, ADIFOR generates an error because it is assuming that the independent

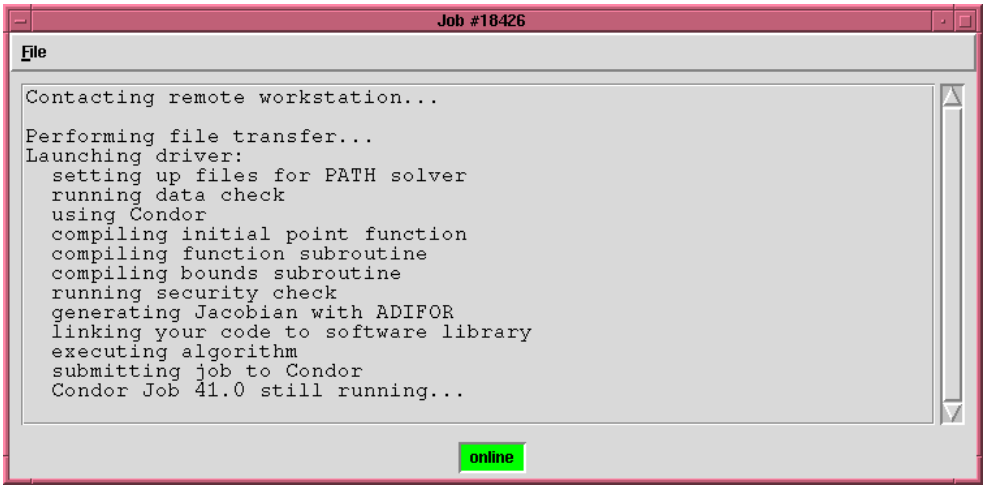

Figure 3.2: Output from the NEOS Submission tool

variables are `f`. On the other hand, ADIFOR does not generate an error if the calling sequence is `fcn(n,x,f,w)` because now there is a dependence on `f`. All error messages generated by ADIFOR are sent back to the user.

If the derivative and sparsity information is generated, this information is sent to PATH. Errors may also occur during this part of the solution process, and it is again important to send appropriate messages back to the user. At present, we check only that the user function does not create any system exceptions during the evaluation of the function at the starting point or at any of the iterates. Although simple, this test catches many user errors. In particular, this test does not allow a calling sequence of the form `fcn(n,x,f,w)`.

Additional checks on the function would be desirable, but seem to be difficult to implement. For example, we would like to check that the function provided is indeed differentiable. If the user provides a function that is discontinuous, automatic differentiation tools will generate the Jacobian matrix but will not be able to detect this situation.

4 ADIFOR

Figure 3.1 shows that given the function F that defines the mixed complementarity problem, the automatic differentiator tool ADIFOR/SparsLinc [3, 5] is used to produce the Jacobian matrix of F and the sparsity structure of the Jacobian matrix. This information is then fed to PATH. In this section we describe the process for generating a representation of the sparse Jacobian matrix of the function F that is submitted to the PATH solver. This process is of interest to any researcher who wishes to use automatic differentiation tools.

The first step in using ADIFOR is to create a *script* file that defines the dependent and independent variables, the name of the subroutine that needs to be differentiated, and a *composition* file.

```
AD_PROG        =  fcn.comp
AD_TOP         =  fcn
AD_IVARS       =  x
AD_DVARS       =  f
AD_SEP         =  _
AD_OUTPUT_DIR  =  .
AD_FLAVOR      =  sparse
```

Figure 4.1: Script file `fcn.script` for ADIFOR

Figure 4.1 shows the script file that is used with PATH. This file tells ADIFOR that the subroutine that needs to be differentiated is called `fcn`, that the independent variables are `x`, and that the dependent variables are `f`. The composition file is specified with `AD_PROG`, so in this case the composition file is `fcn.comp`.

The composition file contains a list of all the files that are required to compute the function, and also a sample program that specifies the calling sequence for the subroutine `fcn`. The composition file used with PATH is shown below:

```
fcn.f
fcn_sample.f
```

This file tells ADIFOR that file `fcn.f` contains the subroutine `fcn` and that the sample program is contained in file `fcn_sample.f`.

All subroutines that are required to evaluate the function must be in the file `fcn.f`. If other subroutines are needed (for example, some `blas` subroutines), they also must be included in `fcn.f`.

The sample file `fcn_sample.f` is not strictly needed because we already know that the subroutine to be differentiated is called `fcn`. However, in older versions of ADIFOR, this file is needed. Figure 4.2 shows the sample file that is used with PATH. The only information specified by this file is the calling sequence used by `fcn`.

```
program fcn_sample
integer n
double precision x(n), f(n)
call fcn(n,x,f)
end
```

Figure 4.2: Sample file `fcn_comp` for ADIFOR

Given the information in the script and composition files, ADIFOR can be used to generate a subroutine that computes the Jacobian matrix. Since we are interested in computing *sparse* Jacobian matrices, the subroutine that ADIFOR generates uses special data structures (called *objects*) to define the Jacobian matrix.

The command `Adifor2.0 AD_SCRIPT=fcn.script` instructs ADIFOR to generate a subroutine of the form

```
g_fcn(n,x,g_x,f,g_f)
```

where `g_x` is a gradient object for the independent variable `x` and `g_f` is a gradient object for the function F. These objects are manipulated and accessed by PATH as described below. Further details on how to invoke ADIFOR can be found in [3, 5].

We compute the Jacobian of F by manipulating the gradient object with subroutines provided by ADIFOR in the SparsLinc [4] library. We first set the gradient object `g_x` for the independent variable `x` to the identity matrix with the code segment

```
do j = 1, n
   call dspsd(g_x(j),j,1.d0,1)
end do
```

Once `g_x` is defined, we use the ADIFOR-supplied subroutine `dspxsq` to compute the Jacobian matrix. The call

```
call dspxsq(ind_col,val,n,g_f(i),lenrow,info)
```

extracts the ith row of the Jacobian matrix. On exit from this call to `dspxsq`, the array `ind_col` contains the column indices of the ith row of the Jacobian matrix, the array `val` contains the values of the ith row, and the variable `lenrow` is the number of nonzeros.

Two key difficulties arise when using the derivative information provided by ADIFOR in PATH. The first difficulty is that PATH, like most optimization software for sparse problems, assumes that the sparsity structure is known for all values of the independent variables `x`. This information is needed in order to preallocate enough storage for the Jacobian matrix and to minimize the cost of preprocessing the Jacobian matrix. For example, orderings that reduce the fill in an elimination algorithm use the sparsity structure, so if the sparsity structure changes at each iteration, then the ordering will have to be recomputed at each iteration. Dynamic storage allocation schemes could be used, but these schemes tend to increase the overall computing time significantly.

We determine a sparsity structure that is valid for all values of the independent variables `x` by evaluating the Jacobian matrix at a random perturbation of the initial point provided by the user. We cannot use the initial point provided by the user to determine a sparsity structure that is valid for all values of the independent variables `x` because the initial starting point tends to be special (for example, the vector of all zeros or all ones), and thus the resulting sparsity structure is not representative. This heuristic was also used by Bouaricha and Moré [7] in a similar situation.

If the sparsity pattern changes as the iteration proceeds then the heuristic that we are using may fail. However, this situation seems to be rare. Heuristics that detect changes in sparsity patterns are the subject of current research.

The second difficulty is that PATH uses a pivotal method to compute the step between iterates, and this method requires that the Jacobian matrix be stored by columns. On the other hand, ADIFOR computes the Jacobian matrix by rows. Storing the Jacobian matrix in a compressed column format specified by an array `ind_row` of row indices and an array `col_ptr` that points to the start of each column is not difficult. We use an additional array `col_start` that initially agrees with `col_ptr`. As we run through the rows of the Jacobian matrix generated by using `dspxsq`, the entries from `val` are immediately put into their correct location (as determined by `ind_row`), and the corresponding entry of `col_start` is incremented. Note that the resulting column-wise storage is sorted by row indices, even though this is not required by PATH. A final check to determine whether the allocated storage is sufficient is carried out after all rows have been processed.

5 Condor

Condor [11, 15] is a distributed resource management system, developed at the University of Wisconsin, that manages large heterogeneous clusters of workstations. Due to the ever decreasing cost of low-end workstations, such resources are becoming prevalent in many workplaces. The Condor design was motivated by the needs of users who would like to take advantage of the underutilized capacity of these clusters for their long-running, computationally-intensive jobs. Condor has been ported to most UNIX platforms and has been used in production mode for more than eight years in the Computer Sciences Department of the University of Wisconsin and many other sites. A version that runs under Windows NT is under development. The system is publicly available under the GNU copyleft restrictions and can be downloaded from

`http://www.cs.wisc.edu/condor/`

In order to generate vast amounts of computational resources, such a system must use any kind of resource whenever it is made available. Condor acts like a matchmaker, pairing these computational resources with jobs that require processing. The job executes on the allocated machine until it completes or the resource disappears. In the latter case, the job is checkpointed, the machine is returned to the owner, and Condor is contacted once again for pairing. Checkpointing a job is the process of saving the current state of the job in a way that allows restarting from precisely the same point of execution.

Condor preserves a large measure of the originating machine's environment on the execution machine, even if the originating and execution machines do not share a common file system. Condor jobs that consist of a single process (it is possible to run PVM on a Condor cluster) are automatically checkpointed and migrated between workstations as needed to ensure eventual completion. Condor is flexible and fault-tolerant: the design features ensure the eventual completion of the job. This feature is important for our application.

A key design feature of the Condor system is that the owner of the resource should have as little interference from the resource allocation server as possible; in this way, more owners will make their resources available to the pool. Condor pays special attention to the needs of the interactive user of the workstation by allowing the user to define the conditions under which the workstation can be allocated by Condor to a batch user. As a consequence, there are currently over 400 workstations in the Wisconsin pool.

The use of Condor for solving complementarity problems generated from NEOS is intended to be an example, showing how a wide variety of software tools can be interfaced and used in a practical operations research environment. The development of software tools is extremely important, but frequently there are few examples demonstrating how the developer envisioned these tools being used. Such examples serve as a prototype for new applications and show potential users how to develop their applications and problems for network solution.

Figure 3.1 shows all the steps used by NEOS to solve a mixed nonlinear complementarity problem. We have already discussed the generation of derivative and sparsity patterns in Section 4. We now outline how Condor schedules job submissions on an appropriate workstation from the Wisconsin pool.

Using Condor-managed resources is easy. Fortran or C code that runs under one of the supported systems can be relinked by using libraries from the Condor system without any changes to the source code. The solver script schedules only large jobs for this facility, since there may be a delay in execution while waiting for an appropriate idle workstation. Small jobs are executed directly on a non-clustered machine at Wisconsin.

The first step in preparing a code for solution using Condor is to ensure that the code compiles and runs on the native machine of that class. Since the PATH solver is already tested and available in library form, this amounts to checking that the submitted routines and ADIFOR-generated routines can be compiled, exactly as is done for a submission that is to run on a local machine. The second step links these objects to the PATH objects, replacing some of the standard libraries with Condor-supplied replacements. Our interface replaces a single line in the standalone makefile, namely,

```
f77 -o pathsol $(OBJECTS) $(LIBS)
```

with the following line

```
condor_compile f77 -o pathsol $(OBJECTS) $(LIBS)
```

Both of these makefiles use precisely the same library of routines that implement PATH as described in [12]. The only difference is that different system libraries are linked into the executable in order to facilitate the checkpointing that was mentioned above.

Instead of executing `pathsol` on the machine where it was compiled, the solver script generates a *job description file* that details the location of the executable, the requirements of the job, and all input and output files. For NEOS, the job description file `jdf` is

```
Executable = pathsol
Log = condor_dir/condor.log
Coresize = 0
Notification = never
Queue
```

This file specifies, in particular, that `condor.log`, in directory `condor_dir`, is the Condor log file. The purpose of the log file is discussed below.

The job is submitted to Condor by using the command `condor_submit -v jdf`. At this stage, Condor takes over control of the job. For security purposes, the job runs as user `nobody`, thereby limiting the access of the submitted job to files that it owns.

In the remainder of this section, we outline how we allow NEOS to timeout from the Condor job and how we guarantee that the job results are returned to the NEOS user - even if the machine that submitted the job to Condor dies during the execution of the job, or communication between NEOS and Condor dies. Note that the Condor job is detached from the submitting machine and is guaranteed to continue to execute to completion.

We first create a persistent directory, `condor_dir`, on the machine that submits the Condor job. All files related to Condor jobs are located in `condor_dir`. When a job is submitted, we create a symbolic link into `condor_dir` the job directory created by the NEOS Communications Package - the facility enabling communication between a solver and the NEOS Server. This job directory serves as the repository for both incoming job submissions and outgoing results.

The Condor log file, `condor.log`, resides in `condor_dir` and shows where each job from NEOS was submitted and executed. We have created a program for monitoring this log with the UserLogAPI that is part of the Condor distribution. The monitoring program, `watchlog`, is regularly invoked by the system utility cron and immediately exits if it finds another version of `watchlog` running.

The purpose of `watchlog` is to ensure that the results of any job submitted to Condor are written to the appropriate job directory and to signal that the Condor job is finished. Condor writes to `condor.log` every time the status of the job changes. The `watchlog` monitoring program acts upon two events that are written into `condor.log`, namely, EXECUTE and TERMINATE. If the job in question starts executing on a machine, `watchlog` adds the IP address of this machine to the list of machines that have been used for this job. At the end of processing, this list is appended to the job results; an example of such a list is given below:

```
<128.105.40.8:32776>
<128.105.41.104:32839>
<128.105.76.12:36651>
<128.105.5.11:56558>
```

The second event that triggers the `watchlog` program is the fact that the job in question is terminating. In this case, `watchlog` writes the status of the Condor job to a file called `CONDOR_DONE` in the job directory. Once the job completes, the results are returned to the NEOS user by the mechanism we now describe.

First note that even if a job was executed under Condor, the solver creates exactly the same solution files and writes these files in the job directory as before. To guarantee that the results are returned to the NEOS user we have to deal with two cases. Firstly, the submitting machine may die while the job is being executed under Condor and secondly, the NEOS user may not be willing to wait more than 5 minutes for the job results to be returned. To deal with both these cases, we have created another cron program, `job-checker`, to ensure that the job results are returned either to NEOS or directly to the user via e-mail.

This program simply monitors the job directory, checking for the existence of the files `CONDOR` and `CONDOR_DONE` (signifying that Condor was used and that Condor had completed the job) and that the file `DONE` does not exist. The solver script creates the file `DONE` once the user has been notified of the job results. Thus if this file is not present, and the job was a Condor job that had timed-out, we return the job results to the user via e-mail. There is a slight race condition here: it is possible that a (Condor) timeout can occur while the job is being returned to NEOS. In this case, the user may get notified of the results both in the NEOS submission tool or WEB browser and via email. We believe this strategy is preferable to the possibility of losing some results.

Acknowledgments

We thank the developers of Condor and ADIFOR for sharing their expertise with us. Todd Munson deserves special thanks for his contributions to the development of the current version of PATH.

References

[1] B. M. Averick, J. J. Moré, C. H. Bischof, A. Carle, and A. Griewank, *Computing large sparse Jacobian matrices using automatic differentiation*, SIAM J. Sci. Statist. Comput., 15 (1994), pp. 285–294.

[2] C. Bischof, A. Bouaricha, P. Khademi, and J. J. Moré, *Computing gradients in large-scale optimization using automatic differentiation*, INFORMS J. Computing, 9 (1997), pp. 185–194.

[3] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, *ADIFOR: Generating derivative codes from Fortran programs*, Scientific Programming, 1 (1992), pp. 1–29.

[4] C. Bischof, A. Carle, and P. Khademi, *Fortran 77 interface specification to the SparsLinC library*, Technical Report ANL/MCS-TM-196, Argonne National Laboratory, Argonne, Illinois, 1994.

[5] C. Bischof, A. Carle, P. Khademi, and A. Mauer, *The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs*, Preprint MCS-P381-1194, Argonne National Laboratory, Argonne, Illinois, 1994. Also available as CRPC-TR94491, Center for Research on Parallel Computation, Rice University.

[6] C. Bischof, L. Roh, and A. Mauer, *ADIC: An extensible automatic differentiation tool for ANSI-C*, Software — Practice and Experience, 27 (1997), pp. 1427–1456.

[7] A. Bouaricha and J. J. Moré, *Impact of partial separability on large-scale optimization*, Comp. Optim. Appl., 7 (1997), pp. 27–40.

[8] J. Czyzyk, M. P. Mesnier, and J. J. Moré, *The Network-Enabled Optimization System (NEOS) Server*, Preprint MCS-P615-0996, Argonne National Laboratory, Argonne, Illinois, 1996. To appear in IEEE Computational Science & Engineering.

[9] S. P. Dirkse and M. C. Ferris, *MCPLIB: A collection of nonlinear mixed complementarity problems*, Optim. Methods Software, 5 (1995), pp. 319–345.

[10] ——, *The PATH solver: A non-monotone stabilization scheme for mixed complementarity problems*, Optim. Methods Software, 5 (1995), pp. 123–156.

[11] D. H. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne, *A worldwide flock of condors: Load sharing among workstation clusters*, Journal on Future Generations of Computer Systems, (1996).

[12] M. C. Ferris and T. S. Munson, *Interfaces to PATH 3.0: Design, implementation and usage*, Mathematical Programming Technical Report 97-12, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1997.

[13] M. C. Ferris and J.-S. Pang, *Engineering and economic applications of complementarity problems*, SIAM Rev., 39 (1997), pp. 669–713.

[14] W. Gropp and J. J. Moré, *Optimization environments and the NEOS server*, in Approximation Theory and Optimization, M. D. Buhmann and A. Iserles, eds., Cambridge University Press, 1997, pp. 167–182.

[15] M. J. Litzkow, M. Livny, and M. W. Mutka, *Condor - A hunter of idle workstations*, in Proceedings of the 8th International Conference on Distributed Computing Systems, Washington, District of Columbia, 1988, IEEE Computer Society Press, pp. 108–111.

[16] E. Simantiraki and D. F. Shanno, *An infeasible-interior-point algorithm for solving mixed complementarity problems*, in Complementarity and Variational Problems: State of the Art, M. C. Ferris and J. S. Pang, eds., Philadelphia, Pennsylvania, 1997, SIAM Publications, pp. 386–404.

[17] L. Wall, T. Christiansen, and R. L. Schwartz, *Programming Perl*, O'Reilly & Associates, Inc., second ed., 1996.