Math. Program., Ser. A 88: 487–505 (2000)

Digital Object Identifier (DOI) 10.1007/s101070000158

Michael C. Ferris · Todd S. Munson

Modeling languages and Condor: metacomputing for optimization*

Received: October 28, 1998 / Accepted: December 01, 1999
Published online June 8, 2000 –

Abstract. A generic framework is postulated for utilizing the computational resources provided by a metacomputer to concurrently solve a large number of optimization problems generated by a modeling language. An example of the framework using the Condor resource manager and the AMPL and GAMS modeling languages is provided. A mixed integer programming formulation of a feature selection problem from machine learning is used to test the mechanism developed. Due to this application's computational requirements, the ability to perform optimizations in parallel is necessary in order to obtain results within a reasonable amount of time. Details about the simple and easy to use tool and implementation are presented so that other modelers with applications generating many independent mathematical programs can take advantage of it to significantly reduce solution times.

1. Introduction

Metacomputers are large confederations of heterogeneous computing resources connected by a local or wide area network, with components ranging from supercomputers to workstations. One method for the optimization community to exploit metacomputers is to develop specialized codes tailored to this environment [9]. In this paper, we develop an alternative framework that uses modeling languages to generate a large number of optimization problems and metacomputing resources to solve them concurrently.

The programming paradigm envisioned is the master/worker model in which a master program creates a pool of independent subproblems that the workers can solve in parallel. Once the subproblems finish, the master program performs additional computation and creates a new set of subproblems. This cycle repeats.

Two functions are required to implement this simple form of task parallelism. The first is a mechanism to spawn tasks for solution on a metacomputer. This call should be non-blocking so that many tasks can be placed in the pool of available work. A corresponding retrieve function is used to obtain the results for a subproblem. To simplify use, this call should be blocking and return the results in the order spawned.

M.C. Ferris, T.S. Munson: Computer Sciences Department, University of Wisconsin – Madison, 1210 West Dayton Street, Madison, Wisconsin 53706, e-mail: ferris,tmunson@cs.wisc.edu

Mathematics Subject Classification (1991): 90C90, 90C11

* This material is based on research supported by the Air Force Office of Scientific Research under grant F49620-98-1-0147 and National Science Foundation Grants CCR-9619765, CCR-9972372, and CDA-9726385.

We develop implementations for these commands using the Condor resource manager to schedule tasks on the metacomputer and the AMPL and GAMS modeling languages to generate the subproblems.

Condor [12,24] exploits one vast, and largely untapped, part of the metacomputer, a pool of pre-existing, "off-the-shelf" workstations. The basic premise for Condor is that most workstations are severely under-utilized. Typically, the workstation owner does some computation and then the machine idles for long stretches of time. Condor notices these idle machines and schedules tasks on them. When the owner wants to perform additional work, Condor migrates the task to another available computer. In essence, Condor uses resources that would have otherwise been lost.

Modeling languages [4,16] provide 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.

Condor 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.

As an example of the use of our tools, we consider a feature selection problem that arises as follows. One branch of the machine learning community, those researching supervised learning [3,18,19], attempts to construct a process based upon historical data for the purpose of forecasting. The feature selection problem chooses a small number of the data characteristics with the best predictive capability. This problem is applicable in numerous situations [2,6,26] and is becoming increasingly important, especially in data mining. Many approaches to solving the problem have been postulated and used [5–7, 20–23,29]. The method presented in this paper solves the problem by generating a large number of independent mixed integer programs. To make this technique practical, we need to perform the individual optimizations in parallel. Hence, it is an ideal candidate to test our mechanism for exploiting metacomputing resources from within a modeling language. Other applications in machine learning, such as boosting, are also amenable to this approach.

The formulation of the feature selection problem is presented in Sect. 2. The tools developed are simple and easy to use and can be adapted for use in other applications. Section 3 provides simple examples in both the AMPL [17] and GAMS [8] modeling languages of how the tool is used. The implementation of the feature selection problem in this framework is also discussed. We present technical details of the tool in Sect. 4. Results of the feature selection using several representative datasets are found in Sect. 5. They demonstrate that significant improvements in overall solution time can be reliably achieved by running mixed integer programming solvers in parallel. Finally, we draw some conclusions and point out relevant extensions of the work in Sect. 6.

Our implementation of tools and feature selection model are available for download from [14].

2. Feature selection

Researchers in the machine learning community studying supervised learning attempt to construct a scheme based upon known historical data to classify unknown observations. While no guarantee can be made that the forecast is correct, a best possible guess is desired. We consider the two category case, in which the given data consists of measurements taken from elements in each of two categories. In order to improve generalization ability, the feature selection problem chooses a small number of the data characteristics with the best predictive capability. The method presented in this paper generates a large number of independent mixed integer programs and is an ideal candidate for using the utilizing metacomputing resources.

Let $\mathcal{A} \subseteq \Re^F$ and $\mathcal{B} \subseteq \Re^F$ be finite sets of observations taken from elements in category 1 and 2 respectively. Each point measures the same set of F characteristics, with elements in the same category sharing similar properties. By $A \in \Re^{|\mathcal{A}| \times F}$ and $B \in \Re^{|\mathcal{B}| \times F}$ we denote the matrices formed from all elements in $\mathcal{A}$ and $\mathcal{B}$, where $|\cdot|$ is used to denote the cardinality of a set.

The approach considered in this paper attempts to quantify differences between the two categories by constructing a separating hyperplane [1,25], $P := \{x \in \Re^F \mid x^T w = \gamma\}$ with $w \in \Re^F$ and $\gamma \in \Re$ such that for all $a \in \mathcal{A}$, $a^T w > \gamma$ and for all $b \in \mathcal{B}$, $b^T w < \gamma$. If $y \in \Re^F$ is an unknown observation we want to classify, we use the following process to categorize it:

1. If $y^T w > \gamma$ then y likely belongs to category 1.
2. If $y^T w < \gamma$ then y likely belongs to category 2.
3. Otherwise, we cannot make any determination.

Typically, we cannot construct such a separating hyperplane, so we choose one minimizing a measure of the misclassification on the known data. Furthermore, recent work [5,6,23] has shown that it is typically beneficial to use a subset of the measured features for the classification. If we employ too few features, we will not have enough information to correctly classify future data. However, if we have too many features, we will overclassify the system and perform poorly on the unseen data. We want to choose the optimal number of features, i.e. that with a minimal expected misclassification percentage.

In the next subsection, we discuss a mechanism for constructing a hyperplane minimizing the sum of the average violation for a given number of features. The following subsection describes how 10-fold cross validation [30] is used to determine the effectiveness of the hyperplane by calculating the expected misclassification percentage. The final subsection discusses the issues involved in finding the best number of features to use and the computational aspects that make this application ideal for our parallelization tool.

2.1. Classification

We use linear programming to construct a hyperplane such that the sum of the average violation of the misclassified points is minimized. In the best case, we want to choose w and γ such that:

$$Aw > e\gamma \text{ and } Bw < e\gamma$$

where e is a vector of ones of the appropriate dimension. We can remove the strict inequalities by normalizing the problem [1] to obtain an equivalent system:

$$Aw \geq e(\gamma + 1) \text{ and } Bw \leq e(\gamma - 1).$$

We want to minimize the sum of the mean violations for each condition [1]. Stated as a linear program, we have:

$$\begin{aligned} \min \ & \frac{1}{|\mathcal{A}|} e^T s + \frac{1}{|\mathcal{B}|} e^T t \\ \text{subject to } & -Aw + e(\gamma + 1) \leq s \\ & Bw + e(1 - \gamma) \leq t \\ & s, t \geq 0. \end{aligned}$$

We note that if $w_i = 0$, then feature i is not used in the construction of the hyperplane, and is hence irrelevant to the classification process.

Since we want to vary the number of features selected, we use a mixed integer programming reformulation that forces some of the components of w to zero. If we want to use at most f of the F features when constructing the hyperplane, we can solve the following problem:

$$\begin{aligned} \min \ & \frac{1}{|\mathcal{A}|} e^T s + \frac{1}{|\mathcal{B}|} e^T t \\ \text{subject to } & -Aw + e(\gamma + 1) \leq s \\ & Bw + e(1 - \gamma) \leq t \\ & -\alpha y \leq w \leq \alpha y \\ & e^T y = f \\ & s, t \geq 0, y \in \{0, 1\}^F \end{aligned}$$

where α is a large positive constant. For $y_i = 0$, we have that $w_i = 0$. Since $e^T y = f$ also holds, at most f features can be used. We note that this model does not preclude the selection of fewer than f features. If $y_i = 1$, then we can still set $w_i = 0$, i.e. feature i is not used. From a statistical learning perspective, a smaller value for α may be desirable. However, the choice of α is beyond the scope of this paper.

The w and γ resulting from solving this mixed integer program provide one hyperplane minimizing the sum of the average violation using at most f features. We now want to determine the expected misclassification percentage, which is done using 10-fold cross validation.

2.2. Validation

n-fold cross validation [30] is a statistical technique used to measure generalization ability. We use the technique to determine the expected misclassification percentage when f features are used to construct a hyperplane minimizing violation. The number

of folds to use is dependent upon the number of observations in the dataset. For the datasets used in the numerical tests, 10 folds are an appropriate number and will be used throughout. The procedure starts by splitting the given data, $\mathcal{A}$ and $\mathcal{B}$, into 10 random subsets of approximately equal size, $\mathcal{A}_1, \ldots, \mathcal{A}_{10} \subseteq \mathcal{A}$ and $\mathcal{B}_1, \ldots, \mathcal{B}_{10} \subseteq \mathcal{B}$. Each element of $\mathcal{A}$ is placed into exactly one of the subsets $\mathcal{A}_j$; similarly for each element of $\mathcal{B}$. Let $\mathcal{A}_j^c := \mathcal{A} \setminus \mathcal{A}_j$ and let $A_j^c \in \Re^{|\mathcal{A}_j^c| \times F}$ denote the matrix formed from the elements of $\mathcal{A}_j^c$. Furthermore, let $\mathcal{B}_j^c := \mathcal{B} \setminus \mathcal{B}_j$ and let $B_j^c \in \Re^{|\mathcal{A}_j^c| \times F}$ denote the matrix formed from the elements of $\mathcal{B}_j^c$. For a fixed $j \in \{1, \ldots, 10\}$, the set $\mathcal{A}_j^c \cup \mathcal{B}_j^c$ is known as the training set, while $\mathcal{A}_j \cup \mathcal{B}_j$ is known as the testing set. We note that approximately 90% of the data will be used for training and 10% for testing.

For each $j \in \{1, \ldots, 10\}$, we solve the following classification problem:

$$\begin{aligned} \min \ & \frac{1}{|\mathcal{A}_j^c|} e^T s + \frac{1}{|\mathcal{B}_j^c|} e^T t \\ \text{subject to } & -A_j^c w + e(\gamma + 1) \leq s \\ & B_j^c w + e(1 - \gamma) \leq t \\ & -\alpha y \leq w \leq \alpha y \\ & e^T y = f \\ & s, t \geq 0, y \in \{0, 1\}^F \end{aligned}$$

for a fixed f giving (w^j, γ^j, y^j). Using the hyperplanes generated for the training sets, we calculate the percentage of misclassified elements on the corresponding testing data. We define the following error functions:

$$\text{error}_{\mathcal{A}}(x, w, \gamma) := \begin{cases} 1 & \text{if } x^T w \leq \gamma, \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{error}_{\mathcal{B}}(x, w, \gamma) := \begin{cases} 1 & \text{if } x^T w \geq \gamma, \\ 0 & \text{otherwise.} \end{cases}$$

These functions indicate if a point has been misclassified with regard to the hyperplane defined by w and γ and the sets $\mathcal{A}$ and $\mathcal{B}$ respectively. We then determine the percent misclassified as follows:

$$p^j := \frac{1}{|\mathcal{A}_j| + |\mathcal{B}_j|} \left(\sum_{a \in \mathcal{A}_j} \text{error}_{\mathcal{A}}(a, w^j, \gamma^j) + \sum_{b \in \mathcal{B}_j} \text{error}_{\mathcal{B}}(b, w^j, \gamma^j) \right).$$

The final number generated is the average percentage of misclassified elements over all 10 folds, $p := \frac{1}{10} \sum_{j=1}^{10} p^j$. This number indicates how well we can be expected to perform on unseen data, assuming that $\mathcal{A}$ and $\mathcal{B}$ accurately represent the real world. Since we use random subsets, we need to run the validation several times to truly obtain a good estimate of expected performance.

2.3. Selection

In order to select the best number of features to use, we vary f from $1, \ldots, F$ and perform the 10-fold cross validation procedure L times for each selected f, tracking the expected misclassification percentage for each validation run. We calculate the mean and standard deviation of the measure over all L validation runs. We choose the number of features producing the smallest mean value as our optimal solution.

From a computational perspective, each of the independent tasks spawned is a mixed integer program. These problems can be very difficult to solve and can require a large amount of computational resources. For a fixed f, we can run $10L$ independent tasks concurrently; each run of 10-fold cross validation generates 10 mixed integer programs and we perform L such validations at a time. The feature selection problem therefore satisfies our candidate criteria and is ideal for the parallelization envisioned. We now turn our attention towards how our tool is used and the implementation for the feature selection problem.

3. Examples

Our tool for solving multiple models in parallel is flexible and powerful, yet can be easily incorporated into pre-existing applications written using the AMPL or GAMS modeling languages. New versions of these packages are not required in order to use our software. Two procedures were developed for our implementation; one to spawn jobs and the other to retrieve the results. Modelers do not have to entirely rewrite their code to exploit inherent parallelism, they simply replace the solve with a spawn command and later issue a retrieve. The commands are implemented within the modeling language framework. The spawn is non-blocking, which means we immediately resume processing the application code after the command is issued, thus enabling us to put multiple models into a work pool before waiting for results. The retrieve is a blocking call that waits until the solution to a problem is available. Users are able to interleave solves performed under Condor and those run on the host machine. For example, problems that contain confidential data can be solved locally, while less sensitive subproblems can be solved elsewhere. Furthermore, Condor allows us to utilize the full resources of a heterogeneous computing environment and allows the state of long running jobs to be saved and restored at a later time.

In the rest of this section, we provide some simple examples illustrating the basic features of our mechanism. We then discuss the issues encountered when coding the feature selection problem.

3.1. Simple examples

The simplest way to use our tool is to replace a single solve command with a spawn/retrieve combination. This technique illustrates the basic method, but fails to exploit the full capabilities of the mechanism. Two simple examples, one within the AMPL modeling language and the other from GAMS, highlight how we can use the spawn and retrieve to solve multiple models in parallel.

3.1.1. AMPL. The AMPL implementation consists of two routines called via the `commands` function, `condor_spawn` and `condor_retrieve`. Instead of issuing a `solve`, the modeler simply replaces it with a `condor_spawn`, and later issues the corresponding `condor_retrieve`.

The example selected for exposition in AMPL is an implementation of Dantzig-Wolfe decomposition for a multi-commodity transportation problem. The original code is available at:

`http://www.ampl.com/cm/cs/what/ampl/NEW/LOOP2/multi2.run`

We only show the portion of the source code modified to run under Condor.

The original code uses a looping construct to solve several subproblems serially:

```
for {p in PROD} { printf "\nPRODUCT %s\n\n", p;
   solve SubI[p];
   display Artif_Reduced_Cost[p];
   ...
};
```

We can solve each individual subproblem via Condor by replacing the `solve` with the following spawn/retrieve combination:

```
for {p in PROD} { printf "\nPRODUCT %s\n\n", p;
   problem SubI[p];
   commands condor_spawn; commands condor_retrieve;
   display Artif_Reduced_Cost[p];
   ...
};
```

In this case, one problem is submitted to Condor and then we immediately wait for the result. This approach is likely to give worse performance than the serial code because of the additional scheduling overhead. A better technique is to repeat the loop, once to spawn all the jobs and the second time to retrieve all the results:

```
for {p in PROD} {
   problem SubI[p]; commands condor_spawn;
};

for {p in PROD} { printf "\nPRODUCT %s\n\n", p;
   problem SubI[p]; commands condor_retrieve;
   display Artif_Reduced_Cost[p];
   ...
};
```

We note that the first loop just spawns the jobs, while the second does all of the output relevant to the application. In this case, all the subproblems are spawned out to separate processors (i.e. machines) before any results are collected into AMPL.

3.1.2. GAMS. Performing the spawn and retrieve in GAMS involves “solving” the model twice. The first time, we replace the solver with one that submits the job to

Condor. The second uses a "solver" that retrieves the result from Condor. A simple example of the process is carried out within the GAMS code given below. In this application, we have two trivial problems, `probA` and `probB`. The models are spawned to Condor to run independently and their results are retrieved after performing a solve task locally. Although seemingly trivial, this example shows how a master/worker program can be formulated and solved in parallel using Condor. Note in particular, that some (sub)problems can be spawned and solved elsewhere, during which time other optimizations (for example, master or synchronization problems) can be carried out on the host machine.

```
variables obj; equation f, g;

f.. (obj - 1) =g= 0.;
g.. sqr( obj ) =l= 2;

model probA / f /; model probB / g /;

* Spawn jobs
option nlp = mns5con;
solve probA using nlp minimizing obj;
solve probB using nlp minimizing obj;

* Perform other tasks
option nlp = minos5;
solve probA using nlp minimizing obj;

* Retrieve results
option nlp = rescon;
solve probA using nlp minimizing obj;
solve probB using nlp minimizing obj;
```

In this case, the modeler just needs to know which solvers are available on the substituting machine (e.g. `minos5`) and what the Condor solvers are called (e.g. `mns5con`). The special `rescon` solver is provided by us to retrieve the results.

3.2. The feature selection problem

The implementation of the feature selection problem presented in Sect. 2 is quite sophisticated. We chose the GAMS modeling language in which to implement it. Before running the application, we need to generate data readable by GAMS. To accomplish this task, we wrote a simple program that converts the data set given into one ready for importation by GAMS. At the same time we also calculate random permutations which are used to split the data for validation. We note that the permutations can be created within GAMS, but we found it easier to calculate them outside of the modeling language.

Initially, the code declares appropriate sets and imports the observation data. We give a small section of this portion of the code below:

```
set a / 1 * 358 /;
set o / 1 * 30 /;

parameter a_data(a, o) /
$include "a_data.inc"
/;
```

We read data for the second category in a similar fashion.

We then declare dynamic subsets for the testing and training data and define some equations:

```
set a_test(a);
set a_trai(a);

positive variable s(a);
variables w(o), gamma;

equations a_def(a);

a_def(a_trai)..
    -sum(o, a_data(a_trai, o) * w(o)) + gamma
      + l = l = s(a_trai);
```

In looping statements, we dynamically determine the elements in `a_test` and `a_trai` and submit a mixed integer program to Condor. Since the equations are just defined over the training set, only the relevant ones are generated.

To make the times for each task more reasonable, we only perform the validations for a fixed f at a time. This technique allows us to use information obtained for f features when we solve the mixed integer program for $f + 1$ features. In particular, the objective value for the f features problems is an upper bound on the objective value for $f + 1$ features. Therefore, we have two groups of looping statements. The first set is simple and generates all of the problems needed for the one feature case and submits them to Condor using the aforementioned syntax. The second set is slightly more complex because it needs to retrieve the results, check for errors and then generate the problem for $f + 1$ features. The inner part of the loop consists of the following:

```
option mip = rescon;
solve train using mip minimizing c;

if (execerror or train.solvestat eq 7
     or train.modelstat eq 11,
        execerror = 0;
        option mip = fatmip;
        solve train using mip minimizing c;
   );
```

```
* Generate report data here.

if (ord(f) lt 8,
        features = ord(f) + 1;
        option mip = fatcon;
        solve train using mip minimizing c;
   );
```

The first solve retrieves the solution to the problem from Condor. The first if-statement checks for execution errors and licensing problems and solves (using fatmip, a serial version of FATCOP [9]) the same problem locally if such a condition exists. An execution error is generated for example if the solver fails to write a solution file. Because of some limitations in the way Condor migrates tasks and the nature of some of the codes used, this error condition is sometimes encountered. The problem arises primarily from a solver simultaneously reading and writing to the same file. Another potential difficulty is that we can run out of licenses for a particular solver. We check this condition in GAMS by looking at the solver and model status reported. The last if-statement spawns the next problem in the sequence if required. This problem will be solved on a remote machine using fatmip. Between the two if-statements, we calculate and store the percent misclassification on the testing data. At the end of all the loops, we calculate the average misclassification percentage and report our findings to the user.

The complete listing of the GAMS program and all relevant data can be downloaded by anonymous ftp [14]. This example demonstrates many of the features incorporated into our tool. Clearly, by adding statements to the AMPL or GAMS languages, we could implement the process more efficiently. However, we believe the strength of our approach derives from its simplicity and the fact that operations researchers can easily and in many practical instances effectively exploit parallelism using it.

4. Technical details

The spawn procedure is a non-blocking call used to submit a task for solution. Before delving into how the mechanism is implemented, we need to first examine how AMPL and GAMS behave when a normal model is solved. The solve statement first writes an internal representation of the model to disk. We note that both modeling languages considered can write a machine independent representation, thus allowing heterogeneous solves. The particular solver executable is then run with appropriate command line arguments. The implementation of the spawn command replaces the solver in this two-step process with a more complex series of statements that implement the following steps:

1. Create a new temporary directory.
2. Copy the internal model representation to the temporary directory.
3. Submit the task to Condor and obtain the associated Condor job number.
4. Place the user job number and Condor job number on a job queue.
5. Return control to the modeling language.

The job queue plays a critical role when we want to retrieve the results.

To submit a job to Condor, we create a job description file. One such file for a GAMS model is provided below:

```
Executable   = gamsfatcon.out
Initialdir   = condor_home/temporary_directory
Arguments    = gamscntr.scr
Log          = condor_home/log
Notification = never
Queue
```

We then submit the job using the command `condor_submit -v`. This gives "verbose" output which we process to obtain the Condor job number. If for some reason the submission fails, we wait a small amount of time and then reattempt the submission.

The retrieve is a blocking call used to obtain the results. When the solver executable finishes, a solution file is written. Essentially, the retrieve places the solution file in the correct location for the modeling language to read. The problem whose solution is desired needs to be in scope before issuing the retrieve command, otherwise problems can occur when the modeling language tries to import the solution. Therefore, we require that results for the models be retrieved in precisely the order they were spawned. While this may be inefficient in terms of load balancing, it is simple to implement and ensures the modeler knows precisely the ordering of jobs. To retrieve a job, we perform the following steps:

1. Take a user job number and Condor job number off the job queue.
2. Wait for the Condor job to finish.
3. Copy the solution file into the correct location.
4. Remove the temporary directory.
5. Return control to the modeling language.

We check to see whether the Condor job has finished by periodically checking the log specified in the job description file. Whenever the Condor job changes state, an appropriate event is noted in this log. We look for the "job completed" event which tell us that the Condor job has finished.

The basic mechanism is the same for the procedures in both AMPL and GAMS. However, differences in the modeling language lead to different implementations. In the next two subsections we discuss the relevant features of the implementation for each of the modeling languages considered. We finish the section by mentioning some Condor related issues.

4.1. AMPL details

AMPL allows us to execute a script via the `commands` feature. The `condor_spawn` is implemented as the following script:

```
write bcondor;
shell 'condor_sub minos x86';
```

The first command writes out a (binary) file called condor.nl which is a representation for the problem currently in scope. The second command executes a perl script called condor_sub that submits the model to Condor for execution. In this particular case, the solver used is MINOS and we run it on the x86 machines in the pool.

Similarly, the `condor_retrieve` is implemented as:

```
shell 'condor_results .';
solution condor.sol;
```

Here, the condor_results script waits for the model to finish processing before returning back to the modeling language. The solution command reads in the condor.sol file generated by the solver.

4.2. GAMS details

The solver GAMS uses can be any executable. Therefore, for the submission call, we created a perl script that is invoked instead of an actual solver. However, GAMS presents some unique issues that needed to be resolved.

The first is that when we copy the model, we also need to copy the options file if one exists. Otherwise, we can run into some file permission problems with Condor. At the same time, we update the gamscntr.scr file so that it looks in the appropriate temporary directory for the model and options.

Another detail is that if we fail to provide a solution to GAMS from any solve statement, we would get an execution error. This is undesirable since the user would have to perform additional checks to remove the error. Therefore, we have implemented a very simple solver that writes a solution file containing the initial, user-specified point and the function evaluation at that point. This simple solver is called at the end of the submission script. In this way, we avoid the problem.

The retrieve is implemented as a special solver that waits for the results and then copies the solution to the correct location, essentially as in Sect. 4.1.

4.3. Condor issues

The Condor tool is designed for high-throughput computing in a dynamic environment. Machines which are currently idle are available for our use. When the owner of the resource returns and wants to work, the machine is removed from the pool and any job currently running on it is stopped and migrated to another available computer. The same mechanisms are used to provide fault tolerance when machines unexpectedly become unavailable. Condor provides two modes of operation for such cases. The 'vanilla' universe is primarily used for jobs that do not require a long time to complete. When 'vanilla' jobs are migrated, the computational effort already expended is lost and the job is restarted from the beginning. All that is required in this case is an executable. For example, existing executables for solver links to AMPL and GAMS could be used in this 'vanilla' universe without any modification whatsoever.

The 'standard' environment is preferred for long running programs and enables a job to be checkpointed and restarted without losing all of the time spent in the code. The

checkpoint appears as a normal executable but contains a "snapshot" of the job at its current state. When the checkpoint is run, it resumes the computation from the exact state at which the checkpoint was made. In order to submit a 'standard' job, the program called need only be relinked using the 'condor_compile' command. This command incorporates the appropriate Condor libraries into the code to perform checkpointing.

Some limitations are placed upon the jobs that we want to checkpoint. The main one we encountered is that the executable is not allowed to simultaneously read and write to the same file. The results are unpredicatable if a code does this type of i/o. Another problem has to do with the use of sockets, which are not allowed under Condor. The could cause potential problems with some license managers. Other limitations have to do with signal handlers and forking processes; these are detailed in [10].

There are several advantages that accrue to a Condor user. First of all, we note that we are not required to have an account on all of the machines in the Condor pool. Condor can start jobs on machines for which we do not have permission to directly use. Condor makes the processor and memory on these machines available to the job submitter. Data is read from and results are written to the submitter's disk either through a "shared file system" or by using remote callbacks. These mechanisms make Condor an ideal "intranet" tool to facilitate large scale computations over a local network of workstations. Internet usage is also possible as documented in [13]; however, in this environment other security issues also arise.

A second key feature is that Condor enables us to run our solvers in a heterogeneous environment. This power comes with one cavéat. Since the checkpoints use system dependent information, once we start a job on a particular architecture, we cannot migrate the job to a different architecture without restarting the job from the beginning. For 'standard' universe jobs, we need to specify the exact platform on which to run the jobs because of this limitation.

Finally, we note that all machines in the pool are not equal. Some of them may have fast processors with a large amount of memory, while others may be slower. The classified ad mechanism provided by Condor allows us to prioritize the available machines. We consequently rank the machines based upon processor speed and available memory. Documentation on the Condor supplied commands, job universes, checkpointing, and classified ads can be obtained from the Condor website [10].

5. Results

Two standard datasets were used for our tests with different characteristics. The galaxy dataset [27] contains a large number of observations, but has a small number of features. There are 1505 and 957 elements in the two categories, dim and bright, respectively. Each has 14 measured features. The WDBC database [26] contains observations of malignant and benign breast tumors. The set consists of observations for 358 benign tumors and 212 malignant tumors. Each observation measures 30 attributes.

Instead of performing the validation for all of the features in each dataset, we varied the number of features selected from 1 to 8. The difficulty of the mixed integer programs solved increases as the number of features selected increases. A model is solved for f features before moving on to selecting $f + 1$ features. The objective function value for

f features is an upper bound on the objective function value for $f + 1$ features. The branch and bound code used, in this case a serial version of FATCOP [9], can use this information to limit the size of the tree generated, making the problem more tractable. For each of the runs, we prove the optimality of our solution. We plot the average number of nodes explored and the average time per solve in Figs. 1 and 2. While in practice it

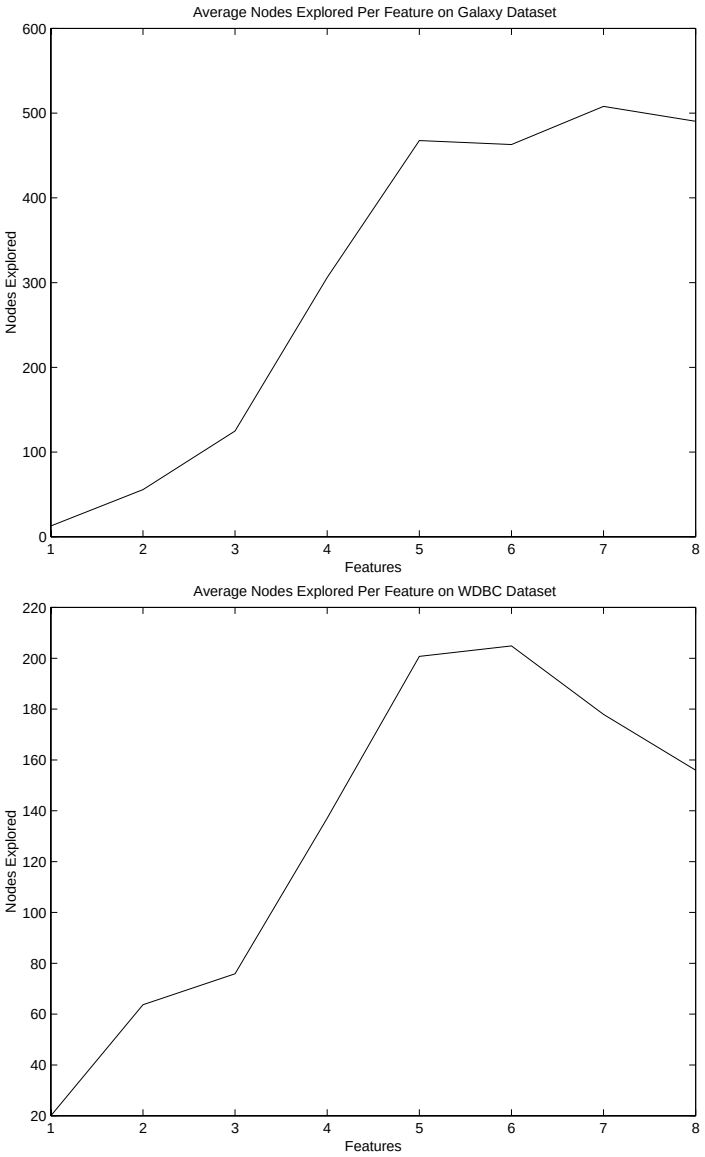

Fig. 1. Plots of average number of nodes explored

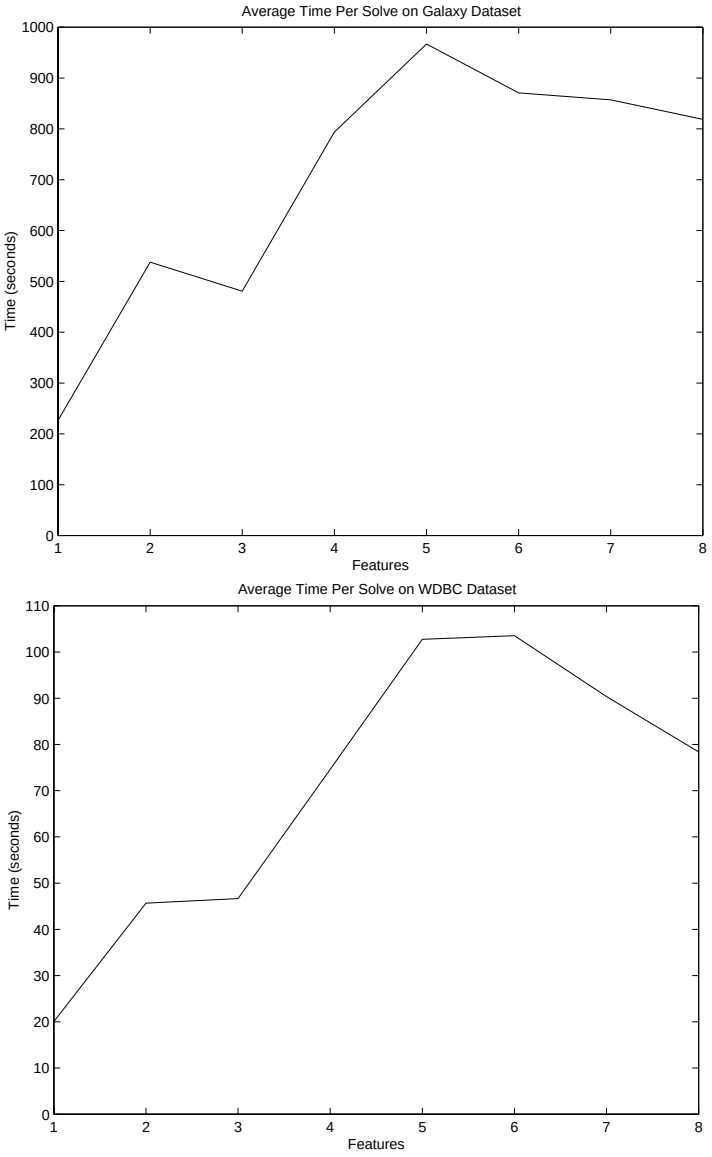

Fig. 2. Plots of average time per problem

may be sufficient to solve the problems to within a prespecified error tolerance, this was not carried out here since both the quality of the solution and utility of our tool were deemed more important.

To maintain efficiency, we attempt to keep as many tasks in the job queue as possible. To do this we immediately spawn $10L$ tasks. Whenever we retrieve results for a model, we immediately add another job to the queue. We decided to perform 10-fold cross

validation 20 times, leading to a maximum of 200 jobs in the job queue at a time. We use up to 8 features, so a total of 1600 jobs were created and solved. Each of the jobs can take from several seconds up to several minutes to complete.

The Condor pool at the University of Wisconsin-Madison, containing around 300 workstations, was used as our testbed. Most faculty and students are active during the daytime hours. Since Condor can only use idle machines, relatively few processors are available at this time. During the nighttime hours and weekends, more resources are usable.

A serial version of the cross validation ran in 14 days for the Galaxy set and 1.4 days for the WDBC dataset. Using Condor we were able to get the results back in 0.37, and 0.21 days respectively. The total speed up is then 37.9 and 6.5 times. The speedup on the WDBC dataset is disappointing but expected since each mixed integer problem solves in a short amount of time. The costs of scheduling each task becomes significant leading to a modest speedup. Furthermore, while the serial code was run on a fast workstation, many of the solves under Condor used significantly inferior machines.

A plot of the misclassification error versus the number of features is found in Fig. 3. We can see from the plot that the optimal number of features to choose for the Galaxy dataset is 5 and the best number of features to select for the WDBC database is 3. These results are consistent with results reported in the literature [5].

6. Conclusion

In this paper, we have described an easy to use tool that provides enormous computational resources to a mathematical modeler. The computational power is harnessed from an existing network of workstations using the Condor resource manager to detect and deliver idle cycles within this network.

To facilitate high throughput within this metacomputing environment, we have built an interface to the two most popular modeling languages in optimization, namely AMPL and GAMS. Our system requires no changes to these languages, but is provided as an add-on feature, built within the confines of the existing language structures. Once our system is installed, only very minor and simple changes are required to an optimizer's model file.

The tool developed is suitable for solving multiple optimization models in parallel from within AMPL and GAMS. We were able to obtain significant speed-up for a particular application, namely the feature selection problem. Other applications, such as multimedia architecture design [11] and radiation therapy [28], can also use the mechanisms provided to improve performance. The only limitation to the number of concurrent jobs is the availability of computational resources.

Our facility can be used in other situations where we can exploit a larger metacomputer concept. For example, we can have one machine dedicated to visualization of results and another pool of workstations for solving problems. Furthermore, due to the tool's generality, we can also exploit other resources, such as supercomputers. The only facilities required are a mechanism to spawn an optimization task on a remote machine and a way to retrieve the results. A key contribution of our work is that we provide a template for these mechanisms in the Condor implementation. Other utilities, such as

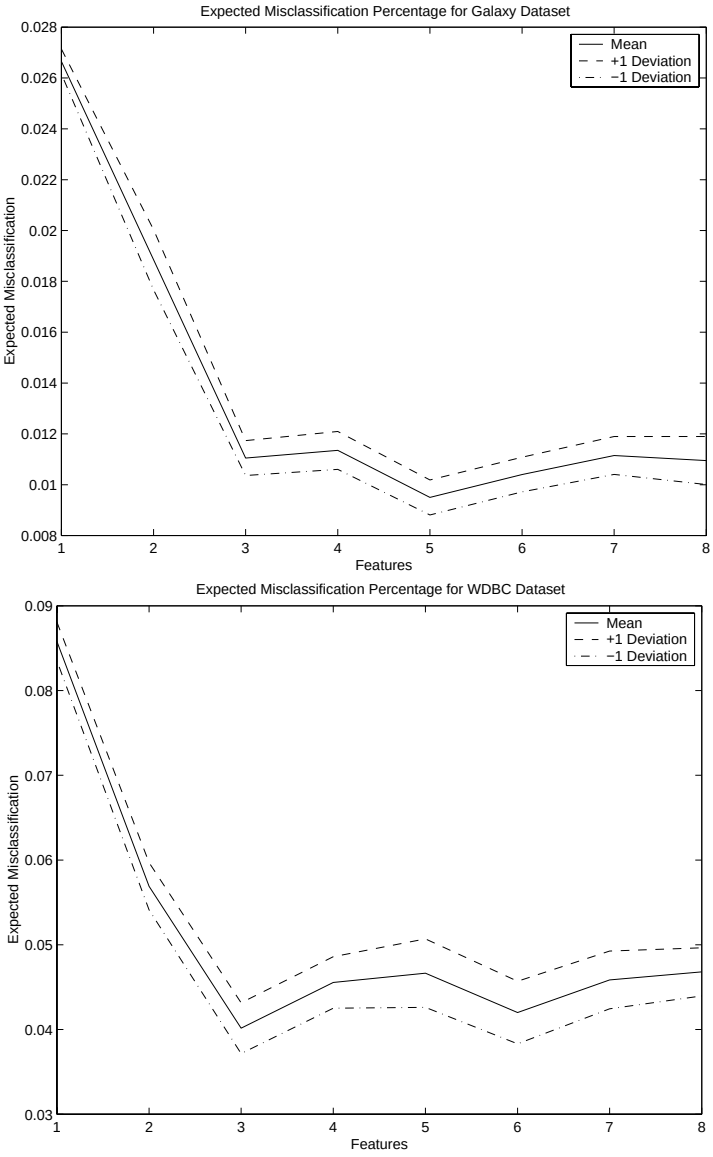

Fig. 3. Plots of average misclassification percentage

GLOBUS [15], could be used for a similar implementation in a broader metacomputing environment.

There are outstanding research issues in this computing environment, some of which have been alluded to in the preceding text. For example, a key issue for metacomputing in data mining and machine learning applications is the treatment of enormous datasets.

Furthermore, concurrent reading and writing of files under Condor is currently not supported. These issues remain the subject of future research.

References

1. Bennett, K.P., Mangasarian, O.L. (1992): Robust linear programming discrimination of two linearly inseparable sets. Optim. Methods Softw. **1**, 23–34
2. Bennett, K.P., Wu, D., Auslender, L. (1998): On support vector decision trees for database marketing. Technical Report, Department of Mathematical Sciences, Rensselaer Polytechnic Institute
3. Bishop, C.M. (1995): Neural Networks for Pattern Recognition. Clarendon Press, Oxford, England
4. Bisschop, J., Meeraus, A. (1982): On the development of a general algebraic modeling system in a strategic planning environment. Math. Program. Study **20**, 1–29
5. Bradley, P.S., Mangasarian, O.L. (1998): Feature selection via concave minimization and support vector machines. In: Shavlik, J., ed., Machine Learning Proceedings of the Fifteenth International Conference (ICML '98), pp. 82–90, San Francisco, California, 1998. Morgan Kaufmann. ftp://ftp.cs.wisc.edu/math-prog/tech-reports/98-03.ps
6. Bradley, P.S., Mangasarian, O.L., Street, W.N. (1998): Feature selection via mathematical programming. INFORMS J. Comput. **10**, 209–217, ftp://ftp.cs.wisc.edu/math-prog/tech-reports/95-21.ps
7. Bredensteiner, E., Bennett, K. (1998): Feature minimization within decision trees. Comput. Optim. Appl. **10**, 111–126
8. Brooke, A., Kendrick, D., Meeraus, A. (1988): GAMS: A User's Guide. The Scientific Press, South San Francisco, CA
9. Chen, Q., Ferris, M.C. (1999): FATCOP: A fault tolerant Condor-PVM mixed integer program solver. Mathematical Programming Technical Report 99-05, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin
10. Computer Sciences Department, University of Wisconsin, Madison, Wisconsin. Condor homepage. http://www.cs.wisc.edu/condor
11. Eager, D.L., Ferris, M.C., Vernon, M.K. (1999): Optimized regional caching for on-demand data delivery. In: Multimedia Computing and Networking, Proceedings of SPIE, **3654**, 301–316. Bellingham, Washington
12. Epema, D.H.J., Livny, M., van Dantzig, R., Evers, X., Pruyne, J. (1996): A worldwide flock of condors: Load sharing among workstation clusters. J. Future Generations Computer Systems **12**, 53–65
13. Ferris, M.C., Mesnier, M.P., Moré, J. (2000): NEOS and Condor: Solving nonlinear optimization problems over the Internet. ACM Trans. Math. Softw. , forthcoming
14. Ferris, M.C., Munson, T.S.: Condor modeling language interface. ftp://ftp.cs.wisc.edu/math-prog/condor
15. Foster, I., Kesselman, C. (1997): Globus: A metacomputing infrastructure toolkit. Int. J. Supercomput. Appl. **11**, 115–128
16. Fourer, R., Gay, D.M., Kernighan, B.W. (1990): A modeling language for mathematical programming. Manage. Sci. **36**, 519–554
17. Fourer, R., Gay, D.M., Kernighan, B.W. (1993): AMPL: A Modeling Language for Mathematical Programming. Duxbury Press
18. Gallant, S.I. (1993): Neural Network Learning and Expert Systems. MIT Press, Cambridge, Massachusetts
19. Hertz, J., Krogh, A., Palmer, R.G. (1991): Introduction to the Theory of Neural Computation. Addison-Wesley, Redwood City, California
20. John, G.H., Kohavi, R., Pfleger, K. (1994): Irrelevant features and the subset selection problem. In: Proceedings of the 11th International Conference on Machine Learning, pp. 121–129, San Mateo, CA. Morgan Kaufmann
21. Kira, K., Rendell, L. (1992): The feature selection problem: Traditional methods and a new algorithm. In: Proceedings of the Tenth National Conference on Artificial Intelligence, pp. 129–134, San Mateo, CA. Morgan Kaufmann
22. Kittler, J. (1986): Feature selection and extraction. In: Young, T.Y., Fu, K.-S., eds., Handbook of Pattern Recognition and Image Processing. Academic Press, New York
23. le Cun, Y., Denker, J.S., Solla, S.A. (1990): Optimal brain damage. In: Touretzky, D.S., ed., Advances in Neural Information Processing Systems II (Denver 1989), pp. 598–605, San Mateo, CA. Morgan Kaufmann
24. Litzkow, M.J., Livny, M., Mutka, M.W. (1988): Condor: A hunter of idle workstations. In: Proceedings of the 8th International Conference on Distributed Computing Systems, pp. 104–111, June 1988

25. Mangasarian, O.L. (1965): Linear and nonlinear separation of patterns by linear programming. Oper. Res. **13**, 444–452
26. Murphy, P.M., Aha, D.W. (1992): UCI repository of machine learning databases. Department of Information and Computer Science, University of California, Irvine, California, http://www.ics.uci.edu/AI/ML/MLDBRepository.html
27. Odewahn, S., Stockwell, E., Pennington, R., Hummphreys, R., Zumach, W. (1992): Automated star/galaxy discrimination with neural networks. Astronom. J. **103**, 318–331
28. Shepard, D.M., Ferris, M.C., Olivera, G., Mackie, T.R. (1999): Optimizing the delivery of radiation to cancer patients. SIAM Rev. **41**, 721–744
29. Siedlecki, W., Sklansky, J. (1988): On automatic feature selection. International Journal of Pattern Recognition and Artificial Intelligence **2**, 197–220
30. Stone, M. (1974): Cross-validatory choice and assessment of statistical predictions. J. R. Statist. Soc. **36**, 111–147