IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 19, 2018, accepted August 15, 2018, date of publication September 13, 2018, date of current version September 28, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2868222

Parampl: A Simple Tool for Parallel and Distributed Execution of AMPL Programs

ARTUR OLSZAK[1] **AND ANDRZEJ KARBOWSKI**[2]
[1]Institute of Computer Science, Warsaw University of Technology, 00-665 Warsaw, Poland
[2]NASK, Research and Academic Computer Network, 01-045 Warsaw, Poland

Corresponding author: Artur Olszak (aolszak@gmail.com)

ABSTRACT Due to the physical processor frequency scaling constraint, current computer systems are equipped with more and more processing units. Therefore, parallel computing has become an important paradigm in the recent years. This paper presents Parampl, a simple tool for parallel and distributed execution of AMPL programs. AMPL is a comprehensive algebraic modeling language for formulating optimization problems. However, AMPL itself does not support defining tasks to be executed in parallel. Although the parallelism is often provided by solvers, in many cases, it is more efficient to formulate the problem in a decomposed way and apply various problem specific enhancements. Parampl introduces explicit asynchronous execution of AMPL subproblems from within the program code. Such an extension yields a new view on AMPL programs, where a programmer is able to define complex, parallelized optimization tasks and formulate algorithms solving optimization subproblems in parallel or in a distributed manner.

INDEX TERMS AMPL, parallel computing, distributed computing, optimization, algebraic modeling languages.

I. INTRODUCTION

In recent years, due to the physical processor frequency scaling constraint, the processing power of current computer systems is mainly increased by employing more and more processing units. This trend can be observed in the TOP500 list of the fastest supercomputers [1], [2]. The top systems reach as much as millions of cores. Similar trend, on a different scale however, is also present in regular home computers and server-class hardware. As hardware supported parallelism has become a standard nowadays, parallel computing and parallel algorithms are recently much of interest. In this paper, we focus on solving optimization problems and defining such problems using AMPL [3]. AMPL - A Mathematical Programming Language is a comprehensive algebraic modeling language for linear and nonlinear optimization problems with continuous and discrete variables. AMPL allows to express an optimization problem in a declarative way, while constructs like conditional expressions and loops enable the modeler to define a program flow that specifies the problem or solves multiple problems in one run. However, parallel processing of the subproblems is not supported by AMPL. For individual problems, the parallelism is often provided by solvers, which take advantage of multiple hardware processing units and employ multi-threading when solving optimization tasks. However, in many situations, it is more efficient to formulate the problem itself in a decomposed way, taking advantage of the problem structure and apply various problem specific enhancements and heuristics, which may not be obvious for the solvers or impossible to recognize at all, e.g. applying Benders decomposition or Lagrangian relaxation [4].

In this paper, we present Parampl, a simple tool for parallel and distributed execution of AMPL programs. Parampl introduces a mechanism for explicit parallel execution of subproblems from within the AMPL program code. The mechanism allows dispatching subproblems to separate threads of execution, synchronization of the threads and coordination of the results in the AMPL program flow, allowing the modeler to define algorithms solving optimization problems with parallel subtasks.

A preliminary version of Parampl was presented at a conference and published in the conference proceedings [5].

The rest of this paper is organized as follows. Section 2 is a brief introduction to AMPL. Section 3 describes the related work. Section 4 presents the design of Parampl, including the information how Parampl can be used in AMPL programs. Section 5 describes the extension of Parampl which allows parallel problems to be solved in a cluster in a distributed way. The evaluation of Parampl and experimental results are presented in Section 6. Section 7 presents future work, and Section 8 concludes.

II. AMPL

AMPL is an algebraic modeling language that allows to express an optimization problem in a declarative way, very similar to its mathematical form. The AMPL problem definition usually consists of the *model* (which is a general form of the problem) and the *data* (values of the model parameters that make the model specific).

Let us assume that we would like to solve the following optimization problem:

$$\max_x \; 3\,x_1 - 2\,x_2$$

$$-x_1+5\,x_2 \le 7.5;\; -2\,x_1+3\,x_2 \ge -13;\; 6\,x_1-7\,x_2 \le 5$$

$$2 \le x_1 \le 45; \quad 0 \le x_2 \le 21$$

The model for general LP problems may be presented in the following form:

$$\max_{l \le x \le u} \; c^T x$$
$$Ax \le b$$

In AMPL, the generic model would be defined as below (model file `aklp.mod`):

```
param n;
param m;

param c{1..n};

param A{1..m, 1..n};
param b{1..m};

param l{1..n};
param u{1..n};

# declaration of the vector x of decision variables
# and the box constraints on its coordinates
var x{i in 1..n} >= l[i], <= u[i];

# objective function
maximize obj_fun:
     sum {i in 1..n} c[i]*x[i];

# linear constraints
subject to matcon {j in 1..m}:
          sum{i in 1..n} A[j,i]*x[i] <= b[j];
```

And the data making the model specific (data file `aklp.dat`):

```
param n := 2; # number of variables
param m := 3; # number of constraints

# vector of obj. function coefficients
param c :=
        1  3
        2 -2 ;

# matrix of linear constraints coefficients
param A :
           1  2 :=
        1 -1  5
        2  2 -3
        3  6 -7 ;

# vector of right hand coefficients
# for linear constraints
param b :=
        1  7.5
        2 13
        3  5 ;

# left box constraints on variables
param l :=
        1 2
        2 0 ;

# right box constraints on variables
param u :=
        1 45
        2 21 ;
```

The problem defined in AMPL is then passed to an external solver (defined using the *solver* option) by calling the AMPL command *solve*. After the solution is calculated, AMPL will set the values of the decision variables to their optimal values found by the solver. The AMPL program would look as follows (`aklp.run`):

```
option solver ipopt;
model aklp.mod;
data aklp.dat;
solve;
display obj_fun;
display x;
```

The transcript of the session follows:

```
$ ampl aklp.run
Ipopt 3.11.1: Optimal Solution Found
suffix ipopt_zU_out OUT;
suffix ipopt_zL_out OUT;
obj_fun = 5.76087
x [*] :=
1  3.36957
2  2.17391
;
```

Despite being a declarative language, AMPL also allows constructs present in the procedural languages which allow to define the program flow - assignments, conditional expressions, loops:

```
if vect["abc",2] < 1 then {
  let vect2["abc",2] := vect2["abc",2] * 1.1;
}
else {
  ...
}

for {a in V} {
  let p[1] := a;
  solve;
  let profits[a] := obj_f;
}

repeat while x > 0 {
  let x := x * 0.7 - 1;
}
```

Thus, AMPL makes it possible to define a program that solves multiple problems sequentially and calculates the result based on the solutions of the subproblems. However, processing of the subproblems cannot be explicitly parallelized.

An extended overview of AMPL, its interaction with solvers and implementation techniques can be found in [6]. The book [3] contains the full description and documentation.

III. RELATED WORK

There have been several works related to extending algebraic modeling languages with possibility of solving problems in parallel. Kestrel [7] is an application that imitates a solver and submits an optimization job to be executed on a remote machine (NEOS server [8]). In AMPL program, Kestrel is chosen as a solver, but Kestrel also supports non-blocking submission of problems to the NEOS server and solution retrieval using external AMPL commands, which allows multiple problems to be solved in parallel. In our opinion, Parampl is a more convenient alternative to the Kestrel interface, as it allows multiple problems to be solved in parallel on a local machine or in a user defined cluster.

In [9], AMPLX toolkit is presented. AMPLX relies on the Everest platform [10] to expose computing services, called applications as REST services. Everest manages execution of application tasks on external computing resources, which can be easily attached to the platform and bound to individual applications. The resources run special programs called agents, which communicate with Everest, making it possible to run tasks on multiple resources in parallel. To use AMPLX, it is necessary to modify the original AMPL script, calling commands to submit tasks to and retrieve solutions from Everest (in a similar way to Parampl), making it possible to exchange data with remote solvers during regular AMPL code execution. AMPLX is provided as a Platform as a Service (PaaS), and, despite being publicly available online, like Kestrel, it relies on the existence of the external platform. With Parampl, it is possible to solve problems in parallel without dependence on any external services.

Colombo *et al.* [11] present a structure-conveying algebraic modeling language for mathematical and stochastic programming (SML). The language is an extension of AMPL which allows definition of the model from sub-models. Such an extension allows the modeler to express nested structure of the problem in a natural and elegant way. The solution is generic as the block structure is passed to the solvers within the problem definition file, so SML can be used with any structure-exploiting solver.

SET [12] is another approach which allows defining the structure of the problem in a separate structure file. Fourer and Gay [13] prove that AMPL's declared suffixes can be used to define the structure of the problem in many common situations. Furthermore, several approaches targeted at stochastic programming have been proposed, for example sMAGIC [14], SAMPL [15], and StAMPL [16].

It is worth mentioning that AMPL itself already provides APIs to programming languages like C++, Java, Matlab and Python, making it possible to write multithreathed applications calling AMPL instances. However, creating such applications requires some programming skills.

In this paper, we present a different approach (with an interface similar to Kestrel), which enables the modeler to define a fork-join structure of the program flow, allowing processing the results of subtasks by a coordination algorithm. Our solution is solver-independent, parallel solver instances are run either locally (separate processes) or are distributed to multiple machines, and the parallel execution and results retrieval is handled at the AMPL level by the modeler.

Despite the existence of environments like Pyomo [17] or JuMP [18], we decided to to develop an extension to AMPL language because of a higher level of abstraction provided by AMPL and its syntax that is very similar to mathematical notation. Developing code in the mentioned environments, based on programming languages (Python [19] and Julia [20] respectively), is, in fact, developing object-oriented code, which causes considerable programming overhead (creating objects, using their handles, method calls) obfuscating the essence of the problem, i.e. the mathematical formulation, decreasing legibility and delaying obtaining the first results. Developing a model in AMPL does not require object programming knowledge, not even programming knowledge in general, which makes it easy to use not only for software engineers. Fourer [21], [22] discusses the importance of expressing models in a clear, high-level way, without the need of real programming. In our opinion, the fork-join execution flow provided by Parampl is a very intuitive and sufficient approach for introducing parallelism to AMPL, as it does not require the modeler to focus on the parallelizing technology itself. The tool created by us is universal - it may be utilized both, on machines equipped with multi-core processing units with shared or local memory (even processing units providing significant amount of parallelism), as well as in networks of machines.

IV. DESIGN AND USE OF PARAMPL

Let us consider a very simple AMPL program, which solves sequentially the same problem for two sets of parameters p1, p2 and stores the results in one vector *res*:

```
var x{i in 1..3} >= 0;
param res {i in 1..6};
param p1 {iter in 1..2};
param p2 {iter in 1..2};
param iter;

minimize obj:
  p1[iter] - x[1]^2 - 2*x[2]^2 - x[3]^2 - x[1]*x[2]
      - x[1]*x[3];

subject to c1:
  8*x[1] + 14*x[2] + 7*x[3] - p2[iter] = 0;

subject to c2:
  x[1]^2 + x[2]^2 + x[3]^2 -25 >= 0;

let p1[1] := 1000;
let p1[2] := 500;
let p2[1] := 56;
let p2[2] := 98;

for {i in 1..2} {
  # define the initial point
  let {k in 1..3} x[k] := 2;
  # solve
  let iter := i;
  solve;
  for {j in 1..3} {
    # store the solution
    let res[(i-1)*3 + j] := x[j];
  };
};

display res;
```

Individual calls of the *solve* command will block until the solution is calculated.

Using Parampl, it is possible to solve multiple problems in parallel. Parampl is a program written in Python programming language [19], which is accessed from AMPL programs by calling two AMPL commands:

- *paramplsub* - submits the current problem to be processed in a separate thread of execution and returns:

```
write ("bparampl_problem_" & $parampl_queue_id);
shell 'python parampl.py submit';
```

- *paramplret* - retrieves the solution (blocking operation) of the first submitted task, not yet retrieved:

```
shell 'python parampl.py retrieve';
if shell_exitcode == 0 then {
  solution ("parampl_problem_"
     & $parampl_queue_id & ".sol");
  remove ("parampl_problem_"
     & $parampl_queue_id & ".sol");
}
```

Before calling these scripts, Parampl must be configured within the AMPL program - the solver to be used and the queueId should be set. The queueId is a unique identifier of the task queue, which is a part of the names of temporary files created by Parampl, which allows executing Parampl in the same working directory for different problems and ensures that the temporary problem, solution and jobs files are not overwritten. The options for the chosen solver should be set in the standard way:

```
option parampl_options 'solver=ipopt';
option parampl_queue_id 'powelltest';
option ipopt_options 'mu_init=1e-6 max_iter=10000';
```

The *paramplsub* script saves the current problem to a *.nl* file (using AMPL command *write*) and executes Parampl with the parameter *submit*. When executed with the parameter *submit*, Parampl generates a unique identifier for the task, the generated *.nl* file is renamed to a temporary file, and the solver is executed in a separate process with the *.nl* file passed to it. Thus, the tasks submitted in this way are executed in parallel (separate processes). After calculating the solution, the solver creates a solution (*.sol*) file with the file name corresponding to the temporary problem file passed to the solver upon execution.

Information about the tasks being currently processed by Parampl is stored in the *jobs* file - new tasks are appended to this file upon submission. The file simply stores the list of the task identifiers being currently executed (jobs that have been submitted and have not yet been retrieved), e.g.:

```
2 3 4 5
```

The temporary problem (.nl) and solution (.sol) file names are composed of the queueId and the task identifier. The *jobs* file name also contains the queueId, so multiple problems may be solved using Parampl at the same time, as long as the queueId value is different for each of them.

The *paramplret* script executes Parampl with the parameter *retrieve*, which is a blocking call, waiting for the first submitted task from the *jobs* file (not yet retrieved) to finish. The solution file is then renamed to the *.sol* file known by the *paramplret* script and is then passed to AMPL using AMPL command *solution*. At this point, the temporary *.nl* file is deleted, and the task id is removed from the *jobs* file. After calling the script *paramplret*, the solution is loaded to the main AMPL program flow as if the *solve* was called. Figure 1 presents the execution flow of an AMPL program using Parampl calls.

When the solver generates the solution file, the *paramplret* command does not immediately load the solution back to AMPL. Depending on the solver and the problem size, the file generation may take more time, and the file might be incomplete if the AMPL *solution* command is called immediately after the solution file is created. That is why the blocking call *parampl retrieve* does not wait for the solution file itself to be generated, but for the notification (.not) file which is created by Parampl after the solver process terminates - calling Parampl with parameter *submit* does not directly run the solver process, but creates another instance of Parampl (executed in the background) which executes the solver and generates the notification (*.not*) file after the solver process returns. This ensures that writing the solution file is finished before reading it back to AMPL.

The logic of Parampl is presented in the listing below using pseudocode:

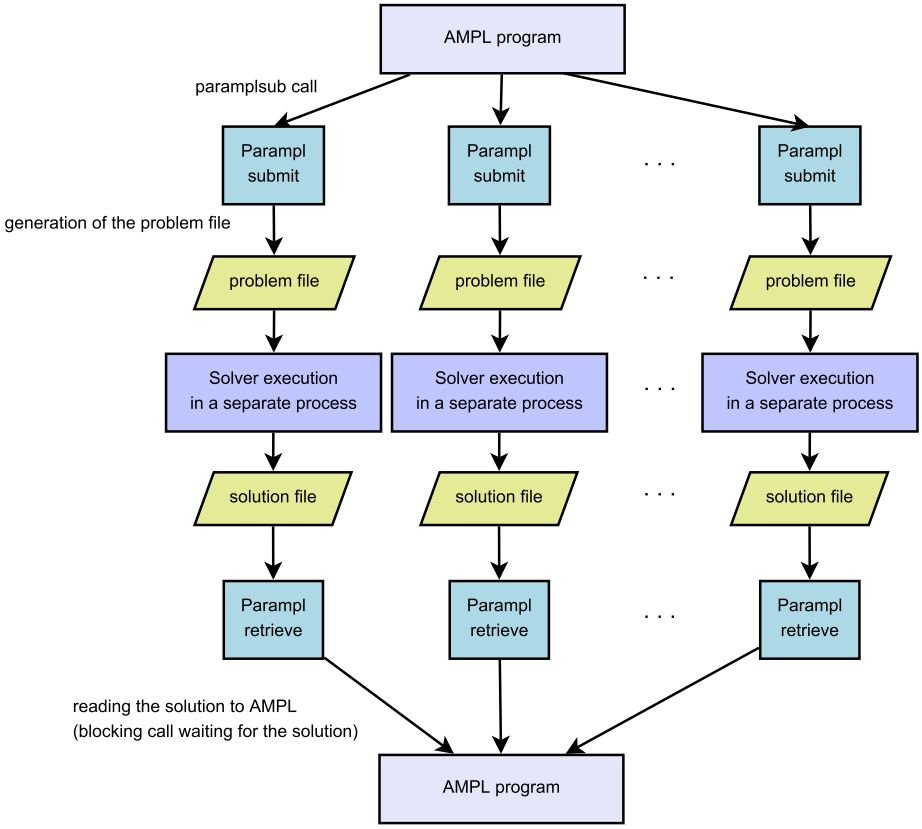

FIGURE 1. Execution of an exemplary AMPL program using Parampl calls.

```
parampl submit:
  jobs = loadJobsFromFile("parampl_jobfile_queueId")
  taskId = generateNextTaskId(jobs)
  jobs.append(taskId)
  saveJobsToFile("parampl_jobfile_queueId", jobs)
  rename("parampl_problem_queueId.nl",
     "parampl_job_queueId_taskId.nl")
  # start the solver in a separate process:
  executeInBackground(
     "parampl executeSolver queueId taskId")
  # "submit" terminates, but "parampl executeSolver"
  # is still being executed in the background
```

```
parampl retrieve:
  jobs = loadJobsFromFile("parampl_jobfile_queueId")
  taskId = jobs.getAndRemoveFirst()
  saveJobsToFile("parampl_jobfile_queueId", jobs)
  # wait for the notification file (blocking):
  waitForFile("parampl_job_queueId_taskId.not")
  # rename the sol file so that AMPL can read it:
  rename("parampl_job_queueId_taskId.sol",
     "parampl_problem_queueId.sol")
  # delete the problem and notification files:
  delete("parampl_job_queueId_taskId.nl")
  delete("parampl_job_queueId_taskId.not")
```

```
parampl executeSolver:
  # execute the solver and wait
  # until the process terminates
  execBlocking(
     "solver parampl_job_queueId_taskId.nl -AMPL")
  # generate notification file
  # after the solver returns:
  generateNotfiticationFile(
     "parampl_job_queueId_taskId.not")
```

The sequential problem presented before may now be run in parallel as follows:

```
for {i in 1..2} {
   # define the initial point.
   let {k in 1..3} x[k] := 2;
   let iter := i;

   # execute solver (non blocking execution):
   commands paramplsub;
};

# the tasks are now being executed in parallel...

for {i in 1..2} {
   # retrieve solution from the solver:
   commands paramplret;

   # store the solution
   for {j in 1..3} {
      let res[(i-1)*3 + j] := x[j];
   };
};
```

In the above scenario, both problems are first submitted to Parampl, which creates a separate process for solving each of them (parallel execution of the solvers), while in the second loop, the solutions for both subtasks are retrieved for further processing by the AMPL code.

V. DISTRIBUTED PARAMPL

Because a single machine might be not sufficient for solving very complex problems due to a limited number of processing cores, and, in practice, memory is often the limiting resource,

Parampl is also able to execute solvers for subproblems on remote machines. However, the modeler should be aware that the subproblems executed on remote machines should be relatively long compared to the time of sending the tasks and retrieving the solution to/from the remote machines. Even within a local network, the overhead is significantly increased compared to running Parampl locally if the problem solving time is short. The remote executions were implemented only for Unix-based operating systems - SSH protocol [23] is used to reach the remote machines - we use OpenSSH implementations of ssh [24] and scp [25] programs. It is required that the users are authenticated using keys (the password does not have to be provided upon every remote execution and copy operation). To distribute the execution to remote machines, two additional AMPL scripts are provided:

- *paramplrsub* - submits the current problem for execution on a remote machine and returns:

```
write ("bparampl_problem_" & $parampl_queueId);
shell 'python parampl.py rsubmit';
```

- *paramplrret* - retrieves the solution (blocking operation) of the first submitted task from a remote machine, not yet retrieved:

```
shell 'python parampl.py rretrieve';
if shell_exitcode == 0 then {
  solution ("parampl_problem_"
     & $parampl_queueId & ".sol");
  remove ("parampl_problem_"
     & $parampl_queueId & ".sol");
}
```

Both of the above scripts perform the same operations as the corresponding *paramplsub* and *paramplret* commands, but Parampl is executed with parameters *rsubmit* and *rretrieve* respectively. Calling Parampl with parameter *rsubmit* moves the problem file to the remote machine and calls Parampl with parameter *submit* on the remote machine, whereas parameter *rretrieve* causes execution of *parampl retrieve* on the remote machine and moves the retrieved solution file to the local machine, which is then ready to be loaded to AMPL.

The *parampl_options*, *parampl_queue_id* options, as well as the solver options should be set in AMPL, and, additionally, three configuration files should be prepared for distributed execution:

- *parampl_cluster_queueId* (queueId is the identifier of the task queue) - specifies the machines of the cluster - the machines are defined in separate lines. Each line contains three values separated by a white character: the hostname or the IP address of the remote machine, the user name on the remote machine, on which behalf Parampl will be executed and the working directory on the remote machine - this should be the path where the Parampl program is located. New tasks are submitted to the machines according to the round-robin strategy (it is possible to submit multiple tasks to each machine). The exemplary cluster definition is presented below:

```
#hostname username dir
ux1       aolszak  /home/aolszak/parampl
ux2       aolszak  /home/aolszak/parampl
ux3       aolszak  /home/aolszak/parampl
ux4       aolszak  /home/aolszak/parampl
ux5       aolszak  /home/aolszak/parampl
ux6       aolszak  /home/aolszak/parampl
ux7       aolszak  /home/aolszak/parampl
ux8       aolszak  /home/aolszak/parampl
```

- *parampl_remote_queueId* - contains three numbers separated by the space character: the index of the machine to which the next task will be submitted, the index of the machine from which the next *parampl_rret* command should retrieve the solution, and the number of tasks being currently executed. When absent, a file with default values will be created automatically:

```
0 0 0
```

- *parampl_envs_queueId* - this file should contain the list of all environment variables (one variable name per line), which should be copied to the remote machines before Parampl execution. Copying certain environment variables to the remote machines is essential because the environment variables mechanism is used for passing configuration to solvers, as well as options of Parampl. That is why this file should always contain the variables *parampl_options*, *parampl_queue_id* and all options required by the solver. Exemplary file content:

```
parampl_options
parampl_queue_id
ipopt_options
```

The listing below presents the logic of the distributed Parampl:

```
parampl rsubmit:
  machines = loadMachinesFromFile(
     "parampl_cluster_queueId")
  (nextSub, nextRet, numTasks) =
     readTaskInfoFromFile("parampl_remote_queueId")
  # default values (0, 0, 0),
  # if the file does not exist
  envs = getEnvVarNamesFromFile(
     "parampl_envs_queueId)
  scp("parampl_problem_queueId.nl",
     machines[nextSub] + dir)
  setEnvs(envs, machines[nextSub])
  ssh(machines[nextSub], "parampl submit")
  nextSub = mod(nextSub + 1, machines.count)
  numTasks = numTasks + 1
  saveTaskInfoToFile(nextSub, nextRet, numTasks,
     "parampl_remote_queueId")
```

```
parampl rretrieve:
  machines = loadMachinesFromFile(
     "parampl_cluster_queueId")
  (nextSub, nextRet, numTasks) =
     readTaskInfoFromFile("parampl_remote_queueId")
  envs = getEnvVarNamesFromFile(
     "parampl_envs_queueId)
  setEnvs(envs, machines[nextRet])
  # retrieve the solution remotely (blocking)
  ssh(machines[nextRet], "parampl retrieve")
  scp(machines[nextRet] + dir
     + "/parampl_problem_queueId.sol", ".")
  nextRet = mod(nextRet + 1, machines.count)
  numTasks = numTasks - 1;
  saveTaskInfoToFile(nextSub, nextRet, numTasks,
     "parampl_remote_queueId")
```

It is also possible to distribute subproblems to multiple machines sharing a common file system. In such cases, addi-

tional time may be saved as the problem file and the solution file do not have to be copied over the network. Even if the machines are located in the same local network, and the physical connection between them is very fast, copying larger files might cause significant delays, especially if the main problem is split into many small (in terms of calculation time) subproblems. In case of running Parampl in a directory shared between the machines, the *shared_fs* parameter should be set to *true* (if not specified, the default value is *false*).

```
option parampl_options=
   'solver=ipopt shared_fs=true';
```

In case the value of *shared_fs* is true, Parampl will not copy the problem and solution files to the remote machines, assuming that all the file system operations are immediately propagated on the remote machines and will only perform the remote execution of Parampl on the cluster nodes. Synchronized file systems which introduce delays between the file system operations and propagating them to other machines, may cause problems when using Parampl with *shared_fs=true*, because the instances of Parampl operate on the same files.

Remote Non-Blocking Solver Execution - Technical Details:

By default, the Unix shell (run in the non-interactive mode) waits for all commands in the pipeline to terminate before returning [26] and, with no tty, ssh connects to stdin/stdout/stderr of the shell process via pipes (which are then inherited by the child processes). Depending on the implementation, ssh may wait for them to close before exiting, which would result in remote task submission returning no sooner than the solver process terminates. In such a case, the tasks on the remote machines would be executed sequentially, not in parallel. Ssh may force a pseudo-tty allocation (ssh -t) [24], in which case the shell will return immediately after the remote call "parampl submit" returns. However, the solver process, which is supposed to continue running in the background, would then receive the SIGHUP signal and terminate (when the parent shell process exits, the kernel sends the SIGHUP signal to its process group [26]). Parampl may detach the solver process from the shell (not terminating *"parampl executeSolver ..."* process running in the background) in several ways:

- Detaching *"parampl executeSolver ..."* using the *screen* program [27]. When *screen* is used to detach the process from the shell, it does not matter whether the pseudo-tty allocation is forced or not.
- Forcing pseudo-tty allocation (ssh -t) and enabling the monitor mode by calling *set −m* shell built-in [26]. If the monitor mode is enabled, all processes are run in their own process groups.
- Forcing pseudo-tty allocation (ssh -t) and wrapping "parampl submit" execution in the *nohup* command call (*nohup parampl submit*). The *nohup* command [26] will prevent sending the SIGHUP signal to the solver process.
- Redirecting pipes (of the remote call to "parampl submit") to /dev/null, so that the shell process returns immediately after "parampl submit" returns.

The method of creating the background process (*"parampl executeSolver ..."*) is determined by the value of the *unix_bkg_method* Parampl option (*option parampl_options*), which may take one of the following values:

- *spawn_nowait* - creates the subprocess in the background by calling the Python *spawn* function [28] with the parameter *os.P_NOWAIT*
- *screen* - executes "screen -dm parampl executeSolver ...", and the *screen* process is created by the blocking *spawn* function call (with the parameter *os.P_WAIT*)

The remote call behavior is configurable by setting the values of the following Parampl options:

- *remote_submit_ssh_allocate_pseudo_tty*
- *remote_submit_shell_monitor_mode_enabled*
- *remote_submit_run_with_nohup*
- *remote_submit_redirect_pipes_to_devnull*

By default, the solver process is created without the use of the *screen* program, the pseudo-tty is not allocated by ssh, the monitor mode is not forced, nohup call is not used and stdin/strout/stderr of the remote call "parampl submit" are redirected to /dev/null.

VI. EVALUATION AND EXPERIMENTS

The evaluation of efficiency of Parampl (local execution) was performed on a machine equipped with a 4-core Intel Core i7-2760QM processor (Intel SpeedStep, C-States Control, Intel TurboBoost and HyperThreading technologies disabled to make sure all logical cores used operate at the same speed) with AMPL ver. 20130704, IPOPT [29] solver ver. 3.11.1 and Python ver. 3.3.2 on Windows 7 64-bit operating system. The application tested was a decomposed version of the generalized problem 20 presented in [30], formulated below:

$$\min_{y\in\mathbb{R}^n} 0.5\cdot(y_1^2+y_2^2+\ldots+y_n^2)$$
$$y_{k+1}-y_k \geq -0.5+(-1)^k\cdot k, \quad k=1,\ldots,n-1$$
$$y_1-y_n \geq n-0.5$$

The decomposed algorithm divides the vector $y\in\mathbb{R}^n$ into p equal parts (assuming p is a divisor of even n). Let us denote:

$$x_{i,j}=y_{(i-1)\cdot n_i+j}, \quad i=1,\ldots,p,\ j=1,\ldots,n_i$$
$$x_i=\left[x_{i,1},x_{i,2},\ldots,x_{i,n_i}\right]^T$$
$$\left[x_1^T,x_2^T,\ldots,x_p^T\right]^T=y$$

where $n_1=n_2=\ldots=n_p=\frac{n}{p}$. We may then divide all n constraints into $p+1$ groups: constraints dependent only on

TABLE 1. Experimental results (decomposed problem 20): 4 cores, $n = 6720$.

p	sequential *solve* [s]	sequential *Parampl* [s]	parallel *Parampl* [s]	speedup	overall speedup	max theoretical speedup
1	1126.9	1143.4	—	—	—	—
2	873.7	890.9	525.3	1.66	2.15	2
3	580.7	580.9	225.8	2.57	4.99	3
4	793.9	801.4	252.3	3.15	4.47	4
5	707.9	709.0	228.6	3.10	4.93	5

x_i subvector for every $i = 1, \ldots, p$:

$$y_{k+1} - y_k \geq -0.5 + (-1)^k \cdot k,$$

where:

$$k = (i-1) \cdot n_i + j,\ j = 1, \ldots, n_i - 1$$

that is

$$x_{i,j+1} - x_{i,j} \geq -0.5 + (-1)^{k(i,j)} \cdot k(i,j)$$

where $k(i,j) = (i-1) \cdot n_i + j$, and p constraints involving different subvectors x_i and x_{i+1}:

$$y_{k+1} - y_k \geq -0.5 + (-1)^k \cdot k,\quad k = i \cdot n_i,\ i = 1, \ldots, p-1$$
$$y_1 - y_n \geq n - 0.5$$

The latter constraints may be written as:

$$x_{mod(i,p)+1,1} - x_{i,n_i} \geq c_i,\quad i = 1, \ldots, p$$

where

$$c_i = -0.5 + (-1)^{(i \cdot n_i)} \cdot (i \cdot n_i)$$

We define the dual problem as:

$$\max_{\lambda \geq 0} \min_{x_i \in X,\ i=1,\ldots,p} L(x, \lambda) \tag{1}$$

where

$$\begin{aligned} L(x,\lambda) &= \sum_{i=1}^{p} \sum_{j=1}^{n_i} 0.5 \cdot x_{i,j}^2 \\ &\quad + \sum_{i=1}^{p} \lambda_i \cdot \left(c_i + x_{i,n_i} - x_{mod(i,p)+1,1}\right) \\ &= \sum_{i=1}^{p} \left[\left(\sum_{j=1}^{n_i} 0.5 \cdot x_{i,j}^2 \right.\right. \\ &\quad \left.\left. + \lambda_i \cdot x_{i,n_i} - \lambda_{mod(p-2+i,p)+1} \cdot x_{i,1} \right) + \lambda_i \cdot c_i \right] \end{aligned} \tag{2}$$

The inner optimization in (1) decomposes into p local problems:

$$\min_{x_i \in X} \sum_{j=1}^{n_i} 0.5 \cdot x_{i,j}^2 + \lambda_i \cdot x_{i,n_i} - \lambda_{mod(p-2+i,p)+1} \cdot x_{i,1} \tag{3}$$

which may be solved independently, if possible, in parallel. The external, dual problem (the coordination problem), may be solved in the simplest case by the steepest ascent gradient algorithm (iterative):

$$\lambda_i := \lambda_i + \alpha \cdot \left(c_i + \hat{x}_{i,n_i}(\lambda) - \hat{x}_{mod(i,p)+1,1}(\lambda)\right),\quad i = 1, 2, \ldots, p$$

where α is a suitably chosen step coefficient and $\hat{x}(\lambda)$ is the optimal vector built of solutions of local problems (3). The algorithm terminates when no significant change of the result vector is achieved.

During the experiments, three variants of the algorithm were tested - sequential (solving p subproblems by calling the blocking *solve* command), sequential Parampl (using *paramplsub* and *paramplret* calls), and parallel (in every iteration, *paramplsub* was first called for the subproblems, after which the results were retrieved by calling *paramplret* - p execution threads). The results of the experiment for $n = 6720$ are presented in Table 1. The column "speedup" presents the speedup achieved compared to the sequential execution of the decomposed algorithm, while "overall speedup" is the speedup in comparison to the calculation time for the original problem. The values presented are the averages for 5 runs.

In the presented results, the effect of employing the parallelism is clearly visible. The larger the number of subtasks, the larger speedup was achieved. The values of speedup are lower than their upper limit (Amdahl's law[1]), which is caused by the differences of solving times for individual subproblems (which is a typical load balancing problem). It is, however, worth mentioning that the overall speedup reached (compared to the original problem calculation time) is even greater than the number of cores, which was achieved by applying a problem specific heuristic.

To verify Parampl running in the distributed mode, we tested a simple AMPL program finding the minimum m of the Griewank function [31]:

$$f(x) = \frac{1}{4000} \cdot \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

in the area restricted to a hypercube $-600 \leq x_i \leq 600, i = 1, \ldots, n$. The function has a complex structure

[1] If β is the proportion of execution time that the part benefiting from parallel execution originally occupied (max. 1), the speedup S with respect to the sequential execution time equals $\frac{1}{(1-\beta)+\frac{\beta}{p}} = p$, where p is the minimum of two values: number of threads and the number of computing cores.

TABLE 2. Experimental results (Griewank): local execution (8 cores).

p	sequential *solve* [s]	parallel *Parampl* [s]	speedup	max theoretical speedup
2	133.07	69.48	1.92	2
4	271.99	72.81	3.74	4
6	418.05	126.18	3.31	6
8	418.53	84.15	4.97	8
10	626.71	102.06	6.26	10
12	760.70	150.30	5.06	12
14	784.19	129.57	6.05	14
16	819.29	141.83	5.78	16
32	1790.47	2862.34	0.63	32

TABLE 3. Experimental results (Griewank): cluster of 8 machines.

p	sequential *solve* [s]	distributed *Parampl* [s]	speedup	max theoretical speedup
2	95.86	50.83	1.89	2
4	198.21	56.17	3.53	4
6	298.87	93.89	3.18	6
8	297.72	60.82	4.90	8
10	460.17	72.73	6.33	10
12	543.38	102.32	5.31	12
14	559.38	70.27	7.96	14
16	587.72	75.10	7.83	16
32	1270.80	244.28	5.20	32

with numerous regularly distributed local minima. For our needs, the domain was partitioned into p regions (split in one dimension k into p equal intervals), and, for each region, the local minimum was calculated:

$$m = \min_{\substack{-600\le x_i\le 600\\ i=1,\ldots,n}} f(x) = \min_{j=1,\ldots,p} \left[\min_{\substack{-600\le x_i\le 600,\ i=1,\ldots,n, i\neq k\\ -600+\frac{1200}{p}(j-1)\le x_k\le -600+\frac{1200}{p}j}} f(x) \right] \quad (4)$$

The inner optimization problems were solved in parallel (p subtasks), after which the results were gathered and the minimum of the results was calculated.

In the experiments, we tested parallel job execution (run locally on a single machine with an 8-core processor and 64-bit Lunux operating system) and in a distributed mode (in a cluster of 8 machines equipped with Intel Core 2 Duo E8400 processors running 64-bit Linux operating system). For both experiments, we used AMPL ver. 20160325, KNITRO [32] solver ver. 10.0.1 and Python ver. 2.7.3. p subproblems were solved sequentially using *solve* command, as well as using Parampl to sumbit the subproblems for parallel/distributed processing (*paramplsub*/*paramplrsub* and *paramplret*/*paramplrret* commands). The results of the experiment for $n = 4000$ and $p \in \{2, 4, 6, 8, 10, 12, 14, 16, 32\}$ are presented in Table 2 (local execution) and Table 3 (distributed execution).

The results of the experiments verified that we benefit from running tasks in multiple threads and distributing tasks to multiple machines. For certain values of p, the speedup values are slightly higher for the local execution. This is because in the distributed mode, a greater portion of time is spent on transferring the problem and solution files to/from the remote machines. The difference is, however, not significant for larger problems, where the calculation time for individual subproblems is long. Moreover, our test program requires large amount of memory, and, when run in parallel locally, 20GB memory limit causes extensive page swapping, resulting in significantly longer calculation times (which can be seen in the results presented in Table 2 for $p = 16$ and $p = 32$). By using distributed Parampl, we are able to overcome the memory limitation, dispatching subproblems to multiple physical machines.

VII. FUTURE WORK

While implementing Parampl, we did not focus on the load balancing, as in our solution, we are relying on the operating system scheduler to distribute the load (processes created by Parampl) to the hardware cores. However, when Parampl is used in the distributed mode, it might happen that a problem is scheduled to a machine still solving the previous problem, while there is another machine that already finished

performing its tasks. Although in many use cases, the tasks are scheduled at the same time (at which point there is no knowledge of the expected execution time), after which the results are collected, this could be a potential area of future development of Parampl, including a possibility of taking advantage of the information about computing power of individual nodes.

VIII. CONCLUSION

In this paper, we presented Parampl, a parallel and distributed task submission extension for AMPL. The experimental results obtained proved that Parampl equips AMPL with a possibility of defining complex parallel algorithms solving optimization problems. It is able to take advantage of multiple processing units while computing the solutions, as well as to distribute the calculation load to multiple machines. Parampl is very easy to use and deploy into existing AMPL programs, and its implementation in Python programming language makes it platform independent.

REFERENCES

[1] J. Dongarra and P. Luszczek, "Top500," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA, USA: Springer, 2011, pp. 2055–2057.
[2] H. W. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon, *The TOP500: History, Trends, and Future Directions in High Performance Computing*, 1st ed. London, U.K.: Chapman & Hall, 2014.
[3] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, 2nd ed. Scituate, MA, USA: Duxbury Press, 2002.
[4] M. Boschetti and V. Maniezzo, "Benders decomposition, Lagrangean relaxation and metaheuristic design," *J. Heuristics*, vol. 15, no. 3, pp. 283–312, 2009.
[5] A. Olszak and A. Karbowski, "Parampl: A simple approach for parallel execution of AMPL programs," in *Proc. 10th Int. Conf. Parallel Process. Appl. Math. (PPAM)* (Lecture Notes in Computer Science), vol. 8385. 2014, pp. 86–94, doi: 10.1007/978-3-642-55195-6_8.
[6] D. M. Gay, "The AMPL modeling language: An aid to formulating and solving optimization problems," in *Numerical Analysis and Optimization* (Springer Proceedings in Mathematics & Statistics), vol. 134. Cham, Switzerland: Springer, 2015, pp. 95–116.
[7] E. D. Dolan, R. Fourer, J.-P. Goux, T. S. Munson, and J. Sarich, "Kestrel: An interface from optimization modeling systems to the NEOS server," *INFORMS J. Comput.*, vol. 20, no. 4, pp. 525–538, 2008.
[8] J. Czyzyk, M. P. Mesnier, and J. J. Moré, "The NEOS server," *IEEE Comput. Sci. Eng.*, vol. 5, no. 3, pp. 68–75, Jul. 1998.
[9] S. Smirnov, V. Voloshinov, and O. Sukhoroslov, "Distributed optimization on the base of AMPL modeling language and everest platform," *Procedia Comput. Sci.*, vol. 101, pp. 313–322, Jan. 2016.
[10] O. Sukhoroslov, S. Volkov, and A. Afanasiev, "A Web-based platform for publication and distributed execution of computing applications," in *Proc. 14th Int. Symp. Parallel Distrib. Comput.*, Jun. 2015, pp. 175–184.
[11] M. Colombo, A. Grothey, J. Hogg, K. Woodsend, and J. Gondzio, "A structure-conveying modelling language for mathematical and stochastic programming," *Math. Program. Comput.*, vol. 1, no. 4, pp. 223–247, Dec. 2009, doi: 10.1007/s12532-009-0008-2.
[12] E. Fragnière, J. Gondzio, R. Sarkissian, and J.-P. Vial, "A structure-exploiting tool in algebraic modeling languages," *Manage. Sci.*, vol. 46, no. 8, pp. 1145–1158, 2000.
[13] R. Fourer and D. M. Gay, "Conveying problem structure from an algebraic modeling language to optimization algorithms," in *Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research*, vol. 12. Boston, MA, USA: Springer, 2000, pp. 75–89.
[14] C. S. Buchanan, K. I. M. McKinnon, and G. K. Skondras, "The recursive definition of stochastic linear programming problems within an algebraic modeling language," *Ann. Oper. Res.*, vol. 104, nos. 1–4, pp. 15–32, Apr. 2001.
[15] C. Valente, G. Mitra, M. Sadki, and R. Fourer, "Extending algebraic modelling languages for stochastic programming," *INFORMS J. Comput.*, vol. 21, no. 1, pp. 107–122, 2009.
[16] R. Fourer and L. Lopes, "StAMPL: A filtration-oriented modeling tool for multistage stochastic recourse problems," *INFORMS J. Comput.*, vol. 21, no. 2, pp. 242–256, 2009.
[17] W. E. Hart, J.-P. Watson, and D. L. Woodruff, "Pyomo: Modeling and solving mathematical programs in python," *Math. Program. Comput.*, vol. 3, no. 3, pp. 219–260, 2011.
[18] M. Lubin and I. Dunning, "Computing in operations research using julia," *INFORMS J. Comput.*, vol. 27, no. 2, pp. 238–248, 2015, doi: 10.1287/ijoc.2014.0623.
[19] G. van Rossum and F. L. Drake, Jr., *The Python Language Reference. Release 3.3.3*, document, Python Software Foundation, 2013.
[20] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *CoRR*, vol. abs/1209.5145, Sep. 2012.
[21] R. Fourer, "On the evolution of optimization modeling systems," *Optim. Stories*, pp. 377–388, 2012.
[22] R. Fourer, "Algebraic modeling languages for optimization," in *Encyclopedia of Operations Research and Management Science*. Boston, MA, USA: Springer, 2013, pp. 43–51.
[23] T. Ylonen and C. Lonvick, *The Secure Shell (SSH) Protocol Architecture*. document RFC 4251, ISOC, IETF, Jan. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4251.txt
[24] *OpenBSD 5.4 Reference Manual (OpenSSH 6.4p1). SSH(1)—Openssh 632 SSH Client (Remote Login Program)*, Dec. 2013.
[25] *OpenBSD 5.4 Reference Manual (OpenSSH 6.4p1). SCP(1)—Secure Copy (Remote File Copy Program)*, Oct. 2013.
[26] *The IEEE and The Open Group, Single UNIX Specification, Version 4. The Open Group Base Specifications Issue 7*, IEEE Standard 1003.1, 2013.
[27] *Screen User's Manual. Version 4.1.0*, Free Softw. Found., Boston, MA, USA, Aug. 2013.
[28] G. van Rossum and F. L. Drake, Jr., *The Python Library Reference. Release 3.3.3*, document, Python Software Foundation, 2013.
[29] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Math. Program.*, vol. 106, no. 1, pp. 25–57, 2006.
[30] M. J. D. Powell, "On the quadratic programming algorithm of Goldfarb and Idnani," in *Mathematical Programming Studies*, vol. 25. Berlin, Germany: Springer, 1985, pp. 46–61.
[31] A. O. Griewank, "Generalized descent for global optimization," *J. Optim. Theory Appl.*, vol. 34, no. 1, pp. 11–39, 1981.
[32] R. H. Byrd, J. Nocedal, and R. A. Waltz, "KNITRO: An integrated package for nonlinear optimization," in *Large-Scale Nonlinear Optimization*. New York, NY, USA: Plenum, 2006, pp. 35–59.

ARTUR OLSZAK received the M.Sc. degree in computer science from the Faculty of Electronics and Information Technology, Warsaw University of Technology, in 2008, and the Ph.D. degree in computer science from the Institute of Computer Science, Warsaw University of Technology, in 2015. His research interests include parallel and distributed computing architectures, environments, tools and algorithms, especially large-scale distributed architectures.

ANDRZEJ KARBOWSKI received the M.Sc. degree in electronic engineering (specialization in automatic control) from the Faculty of Electronics, Warsaw University of Technology, in 1983, and the Ph.D. and D.Sc. degrees in automatic control and robotics in 1990 and 2012, respectively. He is an Adjunct both at the Faculty of Electronics and Information Technology, Institute of Control and Computation Engineering, Warsaw University of Technology, and at Research and Academic Computer Network (NASK). His research interests include data network management, optimal control in risk conditions, and decomposition and parallel implementation of optimization algorithms.