Parampl: A simple approach for parallel execution of AMPL programs

Artur Olszak[1] and Andrzej Karbowski[2,3]

[1] Institute of Computer Science, Warsaw University of Technology
A.Olszak@ii.pw.edu.pl
[2] Institute of Control and Computation Engineering, Warsaw University of Technology
A.Karbowski@elka.pw.edu.pl
[3] NASK, Research and Academic Computer Network, Warsaw, Poland

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. 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 in last years the parallelism is often provided by solvers, which take advantage of multiple processing units, in many cases it is more efficient to formulate the problem in a decomposed way and apply various problem specific enhancements. Moreover, when the number of cores is permanently growing, it is possible to use both types of parallelism.
This paper presents the design of *Parampl* - a simple tool for parallel execution of AMPL programs. *Parampl* introduces explicit asynchronous execution of AMPL subproblems from within the program code. Such an extension implies 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.

Keywords: AMPL, parallel, optimization, modeling languages

1 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. 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 [1]. AMPL - A Modeling Language for Mathematical Programming 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, very similar to its mathematical form. However, despite being a declarative language,

Appears in the Proceedings of the 10th International Conference On Parallel Processing and Applied Mathematics (PPAM 2013), Part II, LNCS 8385, pp. 86-94, 2014.
The original publication is available at www.springerlink.com. DOI: 10.1007/978-3-642-55195-6_8

AMPL also allows the constructions present in procedural languages which allow to define the program flow - assignments, conditional expressions and loops. 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. 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. Parallel solvers can greatly improve the performance of solution calculation, utilizing opportunities for parallelism to make use of all available hardware processing units. 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 Lagrangean relaxation [2].

In this paper, we present *Parampl*, a simple tool for parallel 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 a modeler to define complex parallel algorithms solving optimization problems as subtasks.

The rest of this paper is organized as follows. Section 2 describes the related work. Section 3 presents the design of *Parampl*, including a brief introduction to the usage of *Parampl* in AMPL programs. The evaluation of *Parampl* and experimental results are presented in Section 4. Section 5 concludes the study.

2 Related work

There have been a few works related to extending algebraic modeling languages with constructs allowing defining optimization problems in a decomposed way. One of the most important solutions is Kestrel [3]. Kestrel is an application that imitates a solver and submits an optimization job to be executed on a remote machine (NEOS server [4]). In AMPL program, Kestrel is chosen as a solver (instead of the solver name). The remote solver and other Kestrel parameters are specified within the *kestrel_options* option, while the NEOS server IP address and port are specified by the *neos_server* option:

```
# kestrel instead of the solver name
option solver kestrel;

# configuration of kestrel:
option kestrel_options 'solver=<solverName>';
option neos_server 'www.neos-server.org:3332';
```

The optimization task is then submitted for remote execution by a regular call of *solve* command. Upon receiving a task, the NEOS server returns the job number and the job password:

```
Job has been submitted to Kestrel Kestrel/NEOS Job number: 6893
Kestrel/NEOS Job password: FaahsrIh
```

By default, Kestrel waits for the NEOS Server to send back the solver's results (blocking *solve* command call), but it is possible to interrupt Kestrel, in which case the returned job number and password allow to retrieve the solution for a previously submitted job:

```
option kestrel_options 'job=6893 password=FaahsrIh';
solve;
```

Kestrel also supports non-blocking submission of problems to the NEOS server and solution retrieval using external AMPL commands. This allows multiple problems to be solved in parallel.

Although problem submission and solution retrieval through the Kestrel interface is very flexible, in many situations, such a solution might not be convenient enough as it depends on the existence and the load of the NEOS server. In our opinion, Parampl is a simpler and more convenient alternative to the Kestrel interface, which allows multiple problems to be solved in parallel on a local machine.

A very interesting approach was presented in [5]. The authors 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. The main extension over AMPL is the introduction of the *block* keyword used to define sub-models. The *block* sections group together sub-model entities and allow them to be repeated over an indexing set:

```
block nameofblock {j in nameofset} : {
    ...
}
```

The blocks may contain any number of *set*, *param*, *subject to*, *var*, *minimize* or nested *block* definitions. Such an extension allows the modeler to express the 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 [6] is another approach which allows defining the structure of the problem in a separate structure file. In [7] the authors 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 [8], SAMPL [9], and StAMPL [10].

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

3 Design 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;

   let iter := i;

   solve;

  #store the solution
   for {j in 1..3} {
      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, 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");
}
```

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* creates a unique identifier for the task, renames the generated *.nl* file to a temporary file and executes a solver in a separate process passing the problem file to it. Information about the tasks being currently processed by *Parampl* is stored in the *jobs* file - new tasks are appended to this file. The tasks submitted in this way are executed in parallel in separate processes. After calculating the solution, the solver creates a *.sol* file with the file name corresponding to the temporary problem file passed to the solver upon execution. The solution may be afterwards passed back to AMPL by calling the *paramplret* script.

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 - a notification file is generated. 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 job 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* command was called.

The problem presented above may be run in parallel using *Parampl* in the following way:

```
for {i in 1..2} {
    # Define the initial point.
    let x[1] := 2;    let x[2] := 2;    let x[3] := 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). In the second loop, the solutions for both subtasks are retrieved back to AMPL and may be then processed.

Before calling the *Parampl* scripts, *Parampl* must be configured within the AMPL program - the solver to be used and the queue_id should be set. The queue_id is the 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, e.g.:

```
option  parampl_options  'solver=ipopt';
option  parampl_queue_id  'powelltest';

option  ipopt_options  'mu_init=1e-6 max_iter=10000';
```

4 Evaluation and experiments

The efficiency of *Parampl* was evaluated on a machine equipped with Intel Core i7-2760QM processor with all 4 cores enabled and Intel SpeedStep, C-States Control, Intel TurboBoost and HyperThreading technologies disabled. The machine was running Windows 7 64-bit operating system, AMPL ver. 20130704 and Python ver. 3.3.2.

The application tested was a decomposed version of generalized problem 20 presented in [11], 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 that p is a divisor of even n). Let us denote:

$$x_{i,j} = y_{(i-1)\cdot n_i + j},\ i = 1, \ldots, p,\ j = 1, \ldots, n_i,$$

$$x_i = \left[x_{i,1}, x_{i,2}, \ldots, x_{i,n_i}\right]^T$$

$$\left[x_1^T, x_2^T, \ldots, x_p^T\right]^T = y$$

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

$$y_{k+1} - y_k \geq -0.5 + (-1)^k \cdot k,\ \ k = (i-1)\cdot n_i + j,\ j = 1, \ldots, n_i - 1$$

that is

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

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

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

$$y_1 - y_n >= n - 0.5$$

The latter constraints may be written as:

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

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

$$L(x,\lambda) = \sum_{i=1}^{p} \sum_{j=1}^{n_i} 0.5 \cdot x_{i,j}^2 + \sum_{i=1}^{p} \lambda_i \cdot \left(c_i + x_{i,n_i} - x_{mod(i,p)+1,1}\right)$$

$$= \sum_{i=1}^{p} \left(\left(\sum_{j=1}^{n_i} 0.5 \cdot x_{i,j}^2 + \lambda_i \cdot x_{i,n_i} - \lambda_{mod(p-2+i,p)+1} \cdot x_{i,1} \right) + \lambda_i \cdot c_i \right)$$

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

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

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 (2). The algorithm terminates when no significant change of the result vector is achieved.

For the simulations, we used the solver *IPOPT* [12] ver. 3.11.1 with problem scaling disabled (nlp_scaling_method=none), initial value for the barrier parameter mu_init=1e-6 and the maximum number of iterations max_iter=10e6. Three variants of the algorithm were tested - sequential (solving the subproblems by calling the blocking *solve* command), sequential *Parampl* (using *paramplsub* and *paramplret* calls) and parallel (in every iteration, *paramplsub* was first called for all the subproblems, after which the results were retrieved by calling *paramplret*). The results of simulations for $n = 6720$ and various values of p are presented in Table 1. The column "speedup" presents the speedup achieved when 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.

Table 1. Simulation results: 4 cores, $n = 6720$

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

The decomposed algorithm that we tested appeared to be very sensitive to the problem size (the efficiency varies significantly for various values of n and p - for some problems, more iterations are needed to achieve the same accuracy of the results). It is however a very good example to demonstrate the effect of running programs in parallel on many cores. In the presented simulation results, the effect of employing the parallelism is clearly visible. The larger the number of subtasks, the larger speedup is achieved. The values of speedup are however lower than their upper limit (Amdahl's law), which is caused by the differences of solving times for individual subproblems[4]. Thus, if the differences between

[4] The time of calculations of the AMPL mathematical instructions in the sequential part, i.e. the time of execution of the coordination algorithm is rather negligible. However, for much smaller problems and large numbers of subproblems and iterations, we noticed that significant portion of the execution time is the startup time of the Python virtual machine, and thus the speedup drops sharply.

computation times for individual subproblems might be significant, the number of parallel subproblems should be greater than the number of physical processor cores available to minimize the relative time when some cores are idle while waiting for the remaining tasks to complete. It is 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 (although the accuracy might be slightly worse). Such a speedup could not be achieved just by employing a parallel solver nor any universal automated tool detecting the problem structure.

5 Conclusion

In this paper, the design and usage of *Parampl* was presented, a parallel task submission extension for AMPL. Our experimental results prove 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. *Parampl* is very easy to deploy and use in AMPL programs and its implementation in Python programming language makes it platform independent.

References

1. Fourer, R., Gay, D.M. and Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. Duxbury Press, Brooks/Cole Publishing Company, 2002, Second edition
2. Boschetti, M. and Maniezzo, V.: Benders decomposition, Lagrangean relaxation and metaheuristic design. Journal of Heuristics, vol. 15 (2009), pp. 283-312
3. Dolan, E.D., Fourer, R., Goux, J.-P., Munson T.S. and Sarich, J.: Kestrel: An Interface from Optimization Modeling Systems to the NEOS Server. INFORMS Journal on Computing, vol. 20 (2008), pp. 525-538
4. Czyzyk J., Mesnier M.P., Moré J.J.: The NEOS Server. IEEE Computational Science & Engineering, vol. 5, issue 3 (July 1998), pp. 68-75
5. Colombo M., Grothey A., Hogg J., Woodsend K., Gondzio J.: A structure-conveying modelling language for mathematical and stochastic programming. Mathematical Programming Computation, vol. 1, issue 4 (December 2009), pp. 223-247. DOI 10.1007/s12532-009-0008-2
6. Fragnière, E., Gondzio, J., Sarkissian, R., Vial, J.-P.: Structure exploiting tool in algebraic modeling languages. Management Science, vol. 46 (2000), pp. 1145-1158
7. Fourer R., Gay D.M.: Conveying problem structure from an algebraic modeling language to optimization algorithms. Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research, vol. 12 (2000), pp. 75-89
8. Buchanan, C.S., McKinnon, K.I.M., Skondras, G.K.: The recursive definition of stochastic linear programming problems within an algebraic modeling language. Annals of Operations Research, vol. 104, issue 1-4 (April 2001), pp. 15-32
9. Valente, C., Mitra, G., Sadki, M., Fourer, R.: Extending algebraic modelling languages for stochastic programming. INFORMS Journal on Computing, vol. 21, issue 1 (2009), pp. 107-122

10. Fourer, R., Lopes, L.: StAMPL: A Filtration-Oriented Modeling Tool for Multistage Stochastic Recourse Problems. INFORMS Journal on Computing, vol. 21 (2009), pp. 242-256
11. Powell M.J.D.: On the quadratic programming algorithm of Goldfarb and Idnani. Mathematical Programming Studies, vol. 25 (1985), pp. 46-61
12. Wächter A., Biegler L.T.: On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming. Mathematical Programming, vol. 106, issue 1 (2006), pp. 25-57