Parallel Exploitation for Tree-Structured Coupled Quadratic Programming in Julia

Shervin Parvini Ahmadi and Anders Hansson

Conference article

Cite this conference article as:

Parvini, S., Hansson, A. Parallel Exploitation for Tree-Structured Coupled Quadratic Programming in Julia, In Proceedings of the 22nd International Conference on System Theory, Control and Computing, IEEE; 2018, pp. 597-602. ISBN: 978-1-5386-4444-7

DOI: https://doi.org/10.1109/ICSTCC.2018.8540646

Copyright: IEEE

The self-archived postprint version of this conference article is available at Linköping University Institutional Repository (DiVA):
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-157150

Tweet

LINKÖPING UNIVERSITY

Parallel Exploitation for Tree-Structured Coupled Quadratic Programming in Julia

Shervin Parvini Ahmadi
Division of Automatic Control
Linkoping University
Linkoping, Sweden
shervin.parvini.ahmadi@liu.se

Anders Hansson
Division of Automatic Control
Linkoping University
Linkoping, Sweden
anders.g.hansson@liu.se

***Abstract*—The main idea in this paper is to implement a distributed primal-dual interior-point algorithm for loosely coupled Quadratic Programming problems. We implement this in Julia and show how can we exploit parallelism in order to increase the computational speed. We investigate the performance of the algorithm on a Model Predictive Control problem.**

***Index Terms*—Distributed optimization; Primal-dual interior-point method; Julia; Model predictive control; Parallelization**

I. Introduction

The study of distributed optimization methods has become important due to some issues and challenges that arise with centralized optimization methods. In particular, centralized optimization methods are difficult to use for the following two types of problems. The first type is big size problems with enormous number of decision variables while the computational unit is limited in terms of power and memory. The second type is problems which cannot be described in a centralized manner. This can be the case when the problem data is stored locally or when there is a requirement on respecting the privacy of data. Distributed optimization methods can be used in order to address these challenges. They are typically based on either first order methods where gradients of the objective and constraint functions are used, or second order methods where both gradients and hessians of the objective and constraint functions are used [9]. The effectiveness of these algorithms differs from each other in terms of sensitivity to the scaling of the problem, the local computational cost and the number of iterations for convergence. One drawback of first order methods is that they normally need many iterations to converge to an accurate solution, and this is one of the reasons why there is a great interest in second order methods which have better convergence properties. Some of the works which fall into the class of second order methods are presented in [1], [7], [10], [12], [13], [15]. In [7], the authors present an algorithm for loosely coupled convex problems which is based on primal-dual interior-point method where the Alternating direction method of multipliers (ADMM) is used for calculating the search directions distributedly. The algorithm presented in [1] is a primal-dual interior-point method based distributed algorithm for solving coupled problems with chordal sparsity which relies on an exact search direction computation unlike the ADMM method. In [10], the authors present a distributed Newton-type algorithm for Network Utility Maximization problems. In the proposed algorithm, they use an iterative scheme in order to compute the primal and dual updates for the Newton step in a decentralized manner. The authors then analyze the convergence properties of the proposed algorithm in [11]. In [12], the authors propose an interior-point and Lagrangian dual decomposition based algorithm in order to solve large-scale separable convex problems. In [13], [14], the authors present a parallel structure exploiting interior-point algorithm for convex quadratic programming problems with a so-called nested block structure. In [15], the authors present a parallel algorithm for solving the Newton step in both interior-point and active-set methods when applied to Model Predictive Control (MPC) problem.

Distributed optimization methods have a variety of applications in different fields such as control, electric power systems, estimation, communication systems and economics. One application is MPC where an optimization problem needs to be solved repeatedly. In [2], the authors explore chordality for MPC problems and show how the banded structure stemming from an MPC problem can be exploited in order to find a computational graph for the distributed primal-dual algorithm which enables efficient parallel computations. In [8], a method is introduced for parallel computation of the Riccati recursion which in turn is used to solve constrained finite-time optimal control problems in parallel. Other applications are, localization problems for sensor networks [17] and the problem of optimal power flow and/or optimal frequency control and/or optimal voltage control in the field of electric power systems [9].

There has been enormous interest in the Julia programming language in the scientific community in recent years because of some of its distinguished features in numerical computation. Julia is a high-level dynamic programming language for numerical computing just like Python, Mathematica, MATLAB, etc. The main factor, however, which makes Julia distinguished is the fact that it is designed with the purpose of performance in mind, so that it is comparable to programming languages like C and Fortran. Thus, it is not only fast due to its just-in-time (JIT) compiler, but also it is easy-to-use as is MATLAB or Python. The language achieves this goal by taking advantage of a number of features like Multiple dispatch,

Metaprogramming for code generation, and an expressive type system [3]. One feature, however, that we are particularly interested in is the distributed parallel execution option in Julia, which provides a multiprocessing environment to run a program on multiple processors with separate memory domains simultaneously.

Motivated by this feature in Julia, the idea in this paper is to implement the distributed primal-dual interior-point algorithm in [1] in such a way that we can simultaneously run the algorithm on different processors. Hence we expect to speed up the algorithm run-time, compared to the case where we run the algorithm on a single processor.

The rest of the paper is organized as follows. In Section II we present a simple example in order to clarify the general idea behind the algorithm. In Section III we introduce the type of problems that we consider in the paper. We describe the general idea in [1] briefly in Section IV. In Section V we discuss some technicalities associated with the Julia code. In Section VI we test the implemented algorithm on an MPC problem and discuss different scenarios. Finally, we give some conclusions in Section VII.

II. Simple Example

Consider the following coupled optimization problem[1]:

$$\min_{x_1,x_2,x_3,x_4} F_1(x_1)+F_2(x_1,x_2)+F_3(x_1,x_3)+F_4(x_3,x_4) \quad (1)$$

where

$$F_1(x_1) = \frac{1}{2}x_1^T Q_1 x_1$$
$$F_i(x_n,x_m) = \frac{1}{2}\begin{bmatrix}x_n\\x_m\end{bmatrix}^T Q_i \begin{bmatrix}x_n\\x_m\end{bmatrix},\ i=2,3,4.$$

and Q_is are symmetric matrices. One way of solving this problem is to solve

$$\min_{x_1}\{F_1(x_1)+\min_{x_2}F_2(x_1,x_2)+\min_{x_3}\{F_3(x_1,x_3)+\min_{x_4}F_4(x_3,x_4)\}\} \quad (2)$$

which can be interpreted as follows. Assume that F_1, F_2, F_3 and F_4 are assigned to four different computational nodes as illustrated in Figure 1. Let us also take Node 1 as the root node of the computational graph. Now, in order to solve the problem in (2), in the first step, Node 4 needs to solve its associated subproblem (F_4) with respect to the variable which is not present in Node 3 (x_4) and then send the computed optimal cost function which is a function of x_3, to Node 3. Node 3 is referred to as parent of Node 4. The exchanged function among nodes is referred to as a message. In the second step, Node 3 first adds the received message from Node 4 to its associated subproblem (F_3) and then carries out the same procedure as Node 4, that is, optimizing with respect to the variable which is not present in the parent node and then sending the corresponding message to the parent node. In the third step, Node 2 carries out the same procedure as well. It is worth pointing that the third step can be performed in parallel with the first and second steps. Likewise, in the fourth step, Node 1, which is referred to as the root node, first adds the messages sent from its child nodes (nodes 2 and 3) into its corresponding subproblem, and then solves the obtained problem with respect to x_1. This process is referred to as an upwards pass through the computational graph. Node 1, after finding the optimal x_1, sends it back to its child nodes, so that they can find the optimal x_2, x_3 and x_4, accordingly. This process, in turn, is referred to as a downward pass through the computational graph. Therefore, after one upward and downward pass through the computational graph, the optimal solution is obtained.

As an example for an exchanged message between nodes, let us consider again Node 2. The subproblem to be solved for this node is

$$\min_{x_2}\frac{1}{2}\begin{bmatrix}x_1\\x_2\end{bmatrix}^T Q_2 \begin{bmatrix}x_1\\x_2\end{bmatrix} \quad (3)$$

For this, first we partition Q_2 so that $Q_2 = \begin{bmatrix}R_2 & S_2\\ S_2^T & T_2\end{bmatrix}$. The solution will then be:

$$x_2 = -T_2^{-1}S_2^T x_1$$

and therefore the message to be sent to Node 1 is obtained by substituting x_2 into the cost function, that is

$$m_{21}(x_1) = \frac{1}{2}x_1^T(R_2 - S_2T_2^{-1}S_2^T)x_1$$

As mentioned earlier, Node 2 can work in parallel with nodes 3 and 4. Now let us consider a case where nodes 1 and 2 and nodes 3 and 4 are defined in two different processors. Let us also define the following recursive function

```
function opt_cost = upwardpass(Node)
if  Node has a child
for all children
cost = cost + upwardpass(child)
end for
end if
optimize cost
return cost
end
```

which performs the upward pass stage, on both processors. In particular, when the function is called with the input argument `Node`, first it investigates if `Node` has a child and if so, then for all children of `Node`, it adds the corresponding message to its subproblem. After that it optimizes its subproblem with respect to the variables which does not share with its parent `Node` and then returns it. Now in order to take advantage of parallelism, one can proceeds as follows. Node 1 asks its child which lives in another processor (Node 3), to start performing the upward pass phase, and right after that it

[1]Part of the text is used in an extended abstract which has been submitted to the local Swedish Control Meeting/Reglermotet 2018.

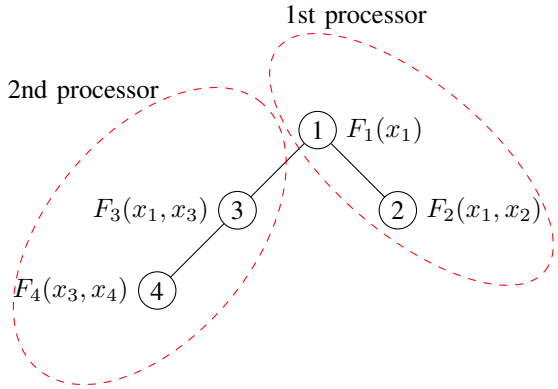

Fig. 1. Computational graph for the example in 2

performs the upward pass phase on its child which lives in the same processor as itself.

III. Problem Formulation

The type of problem that we consider in this paper is a loosely coupled Quadratic Program (QP) formulated as

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \sum_{i=1}^{N} \frac{1}{2} x^T H_i x + f_i^T x \\ \text{subject to} \quad & A_i x + b_i \leq 0, \qquad i = 1, 2, \ldots, N \\ & A_{eq_i} x + b_{eq_i} = 0, \quad i = 1, 2, \ldots, N \end{aligned} \tag{4}$$

where x is an n-dimensions column vector, H_i are $n \times n$ sparse symmetric matrices and f_i, A_i, b_i, A_{eq_i} and b_{eq_i} are matrices with proper dimensions. A_i and A_{eq_i} are sparse as well. Here $\leq$ denotes component-wise inequality. We assume that the problem is sufficiently sparse. Next we show how to find an optimal solution to this problem using a distributed primal-dual interior-point algorithm.

IV. Method

The distributed primal-dual interior-point algorithm in [1] which is a second-order optimization method, solves convex optimization problems with inherent coupling structure. The general idea in that paper is as follows. First a so-called Sparsity graph of the problem which express the coupling structure within the problem is defined. This graph is used in order to compute a so-called chordal embedding of the problem which is needed to define the clique tree of the problem. The obtained tree is then used as the computational graph for the algorithm. Once the computational graph of the problem is found, different terms of the optimization problem are assigned to nodes of the computational graph using a specific rule. The primal-dual search directions are then computed by performing message-passing upwards and downwards through the computational graph at each iteration. The corresponding step size and the termination criterion at each iteration are also computed by going upwards and downwards through the computational graph. For details regarding the algorithm, see [1]. As it is reported in the paper, one of the main advantages of the algorithm over first order methods is its convergence within a finite number of iterations.

V. Julia Implementation

The code for the algorithm is available at the GitHub repository [5]. Next we briefly address some technicalities of the distributed primal-dual interior-point method implementation in Julia. A node in the computational graph is implemented as a Composite Type[2] in Julia which comprises information regarding the processor in which it is defined, the child nodes together with the processors in which they are defined, the parent node and also information regarding the indices of variables assigned to the node and the data matrices. From now on we denote the defined Composite Type by Node.

Concerning the different stages of the presented distributed primal-dual interior-point algorithm in [1], a number of recursive functions are defined on all processors which are referred to as "workers" in Julia. In particular, there are functions for calculating the search direction which is carried out by an upward and downward pass over the computational graph at each iteration; one function implements the upward pass stage and one implements the downward pass stage. Also there are functions for step size calculation and also a number of functions for checking whether the algorithm should be terminated or the perturbation parameter should be updated [1].

All functions take a Node as the input argument. The output of the functions are obviously different, however they can be interpreted as a message from a child Node to its parent Node or vise versa. For example, calling the upward pass function in the search direction calculation step with Node `N`, will output a data matrix which represents a quadratic function for the parent of Node `N`.

As mentioned, the functions are recursive and in general, they work as follows. Whenever a function is called with a Node argument, say `N`, it first investigates if there exist any child Node of `N` which is defined in any other processor, and if so then for all those child Nodes, the same function is invoked from the corresponding processor with the provided corresponding child Node of `N` as the input argument. For invoking a function from another processor, the `@spawnat`[3] Macro is used. Once the functions are invoked from the proper processors, the calling function continues performing the remaining tasks instantly on child Nodes of `N` which are defined on the same processor, in a recursive manner, and at the end it fetches messages from the functions running on the other processes as soon as their computations are finished. This is how we benefit from parallelism in Julia.

VI. Numerical Experiments

One application of the QP problem is the optimization problem that is solved at each sample in linear MPC. This problem can be cast in the form

[2]For details, see https://docs.julialang.org/en/latest/manual/types/

[3]For details, see https://docs.julialang.org/en/latest/manual/parallel-computing/#Distributed-Memory-Parallelism-1

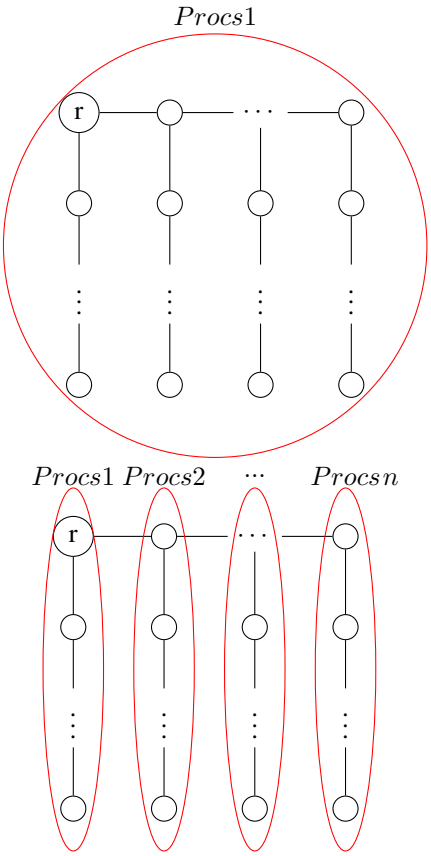

Fig. 2. Computational graph for the problem in (5). In the first setup we only use one processor (the top figure) whereas in the second setup we make use of a number of processors such that each parallel branch performs on a separate processor (the bottom figure).

$$\begin{aligned} \min_{u} \frac{1}{2} \sum_{k=0}^{N-1} \begin{bmatrix} x_k \\ u_k \end{bmatrix}^T Q \begin{bmatrix} x_k \\ u_k \end{bmatrix} + \frac{1}{2} x_N^T S x_N \qquad (5) \\ \text{s.t. } x_{k+1} = Ax_k + Bu_k, \quad x_0 = \bar{x} \\ Cx_k + Du_k \leq e_k \end{aligned}$$

where $u = (u_0, u_1, \ldots, u_{N-1})$ is the optimization variable, x_k is the state vector and A, B, C, D, Q, S, $\bar{x}$ and e_k are given [16].

In order to solve this problem using the distributed interior-point algorithm in [1], a computational graph is needed. A simple computational graph which follows from the banded structure of the problem, is a graph with a chain of nodes. This, in fact, is the well-known backward dynamic programming formulation [2]. With this graph, however, we cannot benefit from parallelism since the computations in the nodes should be carried out sequentially. The type of computational graphs which we are interested in, are the ones with a number of parallel branches, so that we can take advantage of parallelism. To this end, we use the approach proposed in [2], in which we can define computational graphs with arbitrary number of branches for the problem in (5). For details, see [2]. The idea in that paper is similar to what is presented in [8]. The length of each branch is roughly the time horizon N divided by the number of parallel branches.

For the problem in (5), we generate 20 random linear systems using the `drss` function in MATLAB together with two inequality constraints such that x and u have dimensions 5 and 3, respectively. We discard unstable systems and systems for which the condition number of the Controllability Gramian is greater than 100, in order to exclude the systems which are difficult to control, and then we consider different time horizons (N), starting from 50 up to 4000. The weight matrices Q and S are also generated randomly. We run the algorithm on a Linux computational server with 24 cores and clock frequency of 3.07 GHz. The parameters α and β for the step size calculation are chosen to be 0.01 and 0.7, respectively, see [1] and [6] for details. The algorithm is terminated when the surrogate duality gap [1] is less than 10^{-5} and, primal and dual variables are feasible within a tolerance of 10^{-5}. The initial iterates for dual variables $\lambda^{(0)}$ and $v^{(0)}$ are set to $\mathbf{1}$, and for primal variable $x^{(0)}$ is chosen such that it is feasible with respect to the inequality constraints. The number of iterations required for convergence ranges from 6 to 8. In the worst case, the algorithm converges after 10 iterations. It is often observed that for convergence, big size problems, which are the ones with large time horizons, require more iterations than small size problems, which are the ones with short time horizons. Next we compare the computational time for different cases using various number of processors.

A. Scenario 1

Here we run the algorithm for different computational graphs with 2, 4, 8 and 16 parallel branches in two separate setups. As illustrated in Figure 2, in the first setup we only use one processor (the top figure) whereas in the second setup we make use of a number of processors such that each parallel branch performs on a separate processor (the bottom figure). Therefore, for example, in the second setup for a computational graph with 16 parallel branches we use 16 CPUs. The average total computational time for different time horizons is illustrated in Figure 3. Note that the initialization phase where we define the problem is not considered in the recorded computational time. As can be observed, the computational time is smaller when we make use of multiple processors except for the case with short time horizon and several number of processors, in particular, the case with $N = 50$ and 16 processors. This is expected because the communication overhead dominates the computations.

B. Scenario 2

Here we compare the speed-up factor for the computational graphs with 4, 8 and 16 parallel branches with respect to the case with 2 parallel branches, when we use one processor for each parallel branch. The results are illustrated in Figure 4. As can be seen, for big size problems as we increase the number of parallel branches in the computational graph and assign a

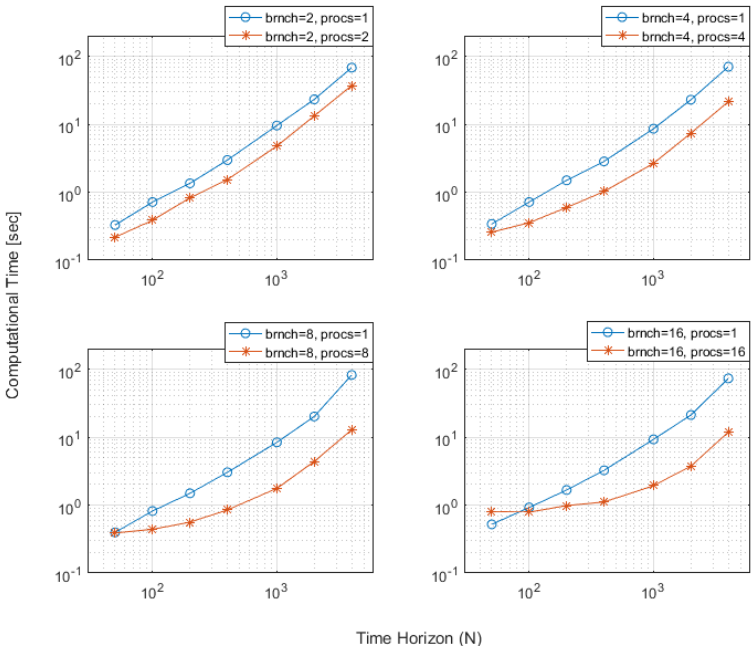

Fig. 3. The average total computational time for cases with 2, 4, 8 and 16 parallel branches over 20 random examples using two setups. In the first setup, one processor is used for the whole computational graph whereas in the second setup, one processor is used for each parallel branch.

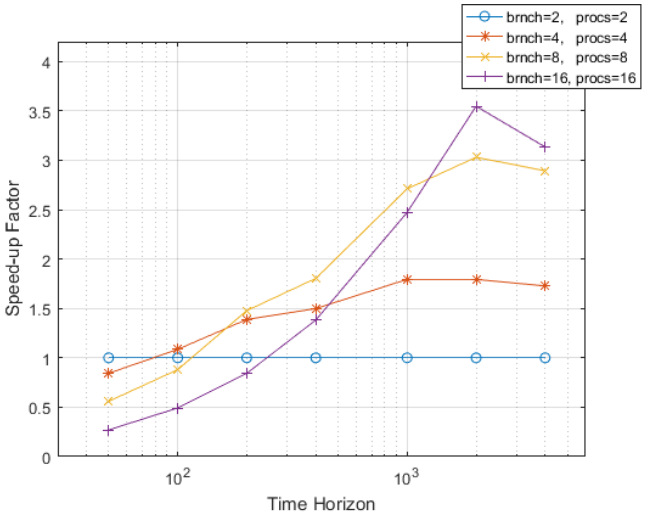

Fig. 4. Speed-up factor for the computational graphs with 4, 8 and 16 parallel branches with respect to the computational graph with 2 parallel branches, over 20 random examples. One processor is used for each parallel branch.

separate processor for each branch, we get a decrease of run-time. In the ideal case, we would expect to get a speed-up factor of 2, as we double up the number of processors in use. However this does not happen in practice. One reason is that the linear algebra operations that we use in Julia are "implicitly parallelized", since the Basic Linear-Algebra Building-Blocks (BLAS) which is underlying the default library for linear algebra operations in Julia (LAPCK), is multithreaded, [3]. Therefore, two kind of parallelizations happen at the same time as we run the program: 1. Implicit parallelization imposed by the programming language, 2. Parallelization imposed by the algorithm. This, in turn, slows down the algorithm compared to the ideal case. For small size problems, however, we obtain larger run-time as we increase the number of processors in use. This happens because the communication overhead for small size problems is considerable, which is not the case for big size problems where the computations on processors dominate over the communication overhead.

C. Scenario 3

In the implemented code which is used for scenarios 1 and 2, all the information required for the algorithm is stored locally in the nodes. However, because of the coupling structure in (4), one variable which can be defined as a global variable and be shared between nodes, is the optimization variable, x and the corresponding search direction, Δx. For this purpose, the Shared Array[4] type can be used in Julia. Here we compare the impact of two types of implementations, one with a distributed x and Δx, and the other with a global and shared x and Δx. The speed-up factor for the case with distributed x and Δx with respect to the case with shared x and Δx is illustrated in Figure 5. As we see, for small size problems with a few number of processors, although the difference between two cases is not very significant, it is reasonable to use shared variables. However, for big size problems it is obvious from the figure that using distributed and local variables accelerates the algorithm.

[4]For details, see https://docs.julialang.org/en/latest/manual/parallel-computing/#Distributed-Memory-Parallelism-1

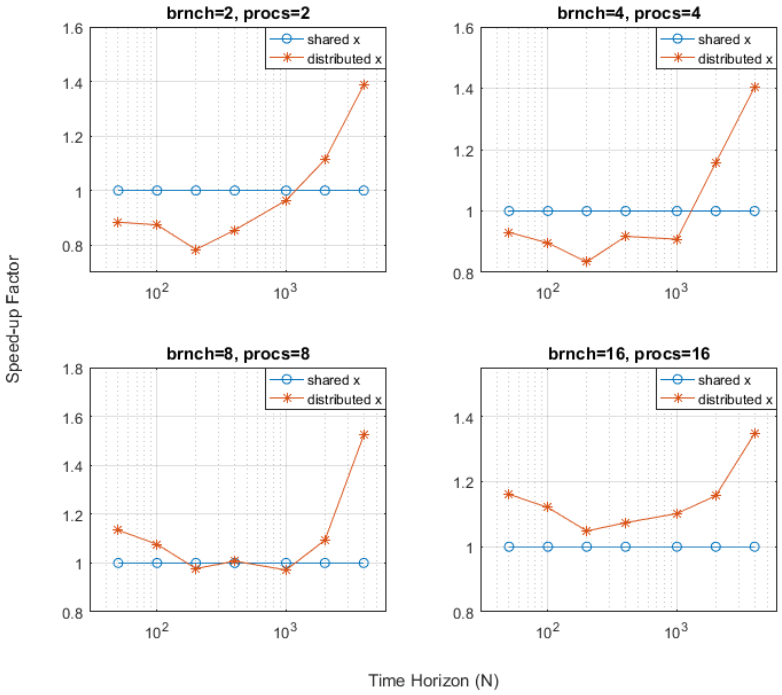

Fig. 5. Speed-up factor for the case with distributed x and Δx with respect to the case with shared x and Δx, over 20 random examples. One processor is used for each parallel branch.

VII. Conclusion

We implemented a distributed primal-dual interior-point algorithm for loosely coupled Quadratic Programming problems in Julia. We discussed how we can take advantage of parallelism in order to accelerate the execution time of the algorithm. We evaluated the algorithm on an MPC problem. For large scale problems we benefit from parallelism just as in [8].

Acknowledgment

This work was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The authors are grateful for discussions with Sina Khoshfetrat Pakazad and Erik Hedberg.

References

[1] Khoshfetrat Pakazad, Sina, Anders Hansson, Martin S. Andersen, and Isak Nielsen. "Distributed primaldual interior-point methods for solving tree-structured coupled convex problems using message-passing." Optimization Methods and Software 32, no. 3 (2017): 401-435.

[2] Hansson, Anders, and Sina Khoshfetrat Pakazad. "Exploiting Chordality in Optimization Algorithms for Model Predictive Control." arXiv preprint arXiv:1711.10254 (2017).

[3] Bezanson, Jeff, Alan Edelman, Stefan Karpinski, and Viral B. Shah. "Julia: A fresh approach to numerical computing." SIAM review 59, no. 1 (2017): 65-98.

[4] Pakazad, Sina Khoshfetrat. Divide and Conquer: Distributed Optimization and Robustness Analysis. Department of Electrical Engineering, Linkping University, 2015.

[5] Parvini Ahmadi, Shervin. DistPrimDualIntPoint: https://github.com/ShervinParvini/DistPrimDualIntPoint. Accessed: 2018-04-09.

[6] Boyd, Stephen, and Lieven Vandenberghe. Convex optimization. Cambridge university press, 2004.

[7] Annergren, Mariette, Sina Khoshfetrat Pakazad, Anders Hansson, and Bo Wahlberg. "A distributed primal-dual interior-point method for loosely coupled problems using ADMM." arXiv preprint arXiv:1406.2192 (2014).

[8] Nielsen, Isak, and Daniel Axehill. "A parallel structure exploiting factorization algorithm with applications to model predictive control." In Decision and Control (CDC), 2015 IEEE 54th Annual Conference on, pp. 3932-3938. IEEE, 2015.

[9] Molzahn, Daniel K., Florian Drfler, Henrik Sandberg, Steven H. Low, Sambuddha Chakrabarti, Ross Baldick, and Javad Lavaei. "A survey of distributed optimization and control algorithms for electric power systems." IEEE Transactions on Smart Grid 8, no. 6 (2017): 2941-2962.

[10] Wei, Ermin, Asuman Ozdaglar, and Ali Jadbabaie. "A distributed Newton method for network utility maximizationI: Algorithm." IEEE Transactions on Automatic Control 58, no. 9 (2013): 2162-2175.

[11] Wei, Ermin, Asuman Ozdaglar, and Ali Jadbabaie. "A distributed newton method for network utility maximizationPart II: Convergence." IEEE Transactions on Automatic Control 58, no. 9 (2013): 2176-2188.

[12] Necoara, I., and J. A. K. Suykens. "Interior-point lagrangian decomposition method for separable convex optimization." Journal of Optimization Theory and Applications 143, no. 3 (2009): 567.

[13] Gondzio, Jacek, and Andreas Grothey. "Parallel interior-point solver for structured quadratic programs: Application to financial planning problems." Annals of Operations Research 152, no. 1 (2007): 319-339.

[14] Gondzio, Jacek, and Andreas Grothey. "Exploiting structure in parallel implementation of interior point methods for optimization." Computational Management Science 6, no. 2 (2009): 135-160.

[15] Nielsen, Isak, and Daniel Axehill. "An O (log N) parallel algorithm for Newton step computation in model predictive control." IFAC Proceedings Volumes 47, no. 3 (2014): 10505-10511.

[16] Nielsen, Isak. Structure-exploiting numerical algorithms for optimal control. Vol. 1848. Linkping University Electronic Press, 2017.

[17] Pakazad, Sina Khoshfetrat, Emre zkan, Carsten Fritsche, Anders Hansson, and Fredrik Gustafsson. "Distributed Localization of Tree-structured Scattered Sensor Networks." arXiv preprint arXiv:1607.04798 (2016).