Parallel Implementation of Riccati Recursion for Solving Linear-Quadratic Control Problems

Frison, Gianluca; Jørgensen, John Bagterp

Published in:
Proceedings of the 18th Nordic Process Control Workshop

Publication date:
2013

Parallel Implementation of Riccati Recursion for Solving Linear-Quadratic Control Problems

Gianluca Frison * **John Bagterp Jørgensen** *

* *Technical University of Denmark, DTU Compute - Department of Applied Mathematics and Compute Science, DK-2800 Kgs Lyngby, Denmark. (e-mail: {giaf, jbj} at imm.dtu.dk).*

Abstract: In both Active-Set (AS) and Interior-Point (IP) algorithms for Model Predictive Control (MPC), sub-problems in the form of linear-quadratic (LQ) control problems need to be solved at each iteration. The solution of these sub-problems is usually the main computational effort. In this paper an alternative version of the Riccati recursion solver for LQ control problems is presented. The performance of both the classical and the alternative version is analyzed from a theoretical as well as a numerical point of view, and the alternative version is found to be approximately 50% faster than the classical one, for systems with many states. A number of parallel implementations of the alternative version has been proposed and tested.

Keywords: Riccati recursion, LQ control problem, parallel computation

1. INTRODUCTION

The linear-quadratic (LQ) control problem can be considered the core problem in Model Predictive Control (MPC). In its classical formulation, it represents an unconstrained optimal control problem where the controlled system is linear time-invariant and the cost function is quadratic. This problem formulation is especially important because it arises as a sub-problem in Active-Set (AS) and Interior-Point (IP) algorithms for MPC (Wright (1997); Rao et al. (1998); Jørgensen et al. (2004)). The solution of these sub-problems is typically the main computational effort at each iteration, and this explains the need for efficient solvers.

From a mathematical point of view, the LQ control problem is an equality constrained quadratic program, and it can be solved using general solvers for this class of problems. The cost of this approach is $\mathcal{O}(N^3(n_x + n_u)^3)$, where N is the control horizon length, n_x is the number of states and n_u is the number of controls (or inputs).

However, it is well known that the KKT system associated with the LQ control problem is sparse and highly structured, and this structure can be exploited to obtain more efficient solvers. In case of dense controlled systems, the Riccati recursion based solver is known to be the fastest among a large class of solvers (Frison et al. (2013)).

In this paper, we present two versions of the Riccati recursion based solver for an extended formulation of the LQ control problem. For both the classical and the alternative (called 'factorized' in Frison et al. (2013)) version, we state a detailed description of the algorithm, and we suggest and test the use of numerical libraries for their parallel implementation on shared memory machines. The implementation of the classical version scales quite well with the number of threads, since its key routine (the matrix-matrix multiplication routine) is particularly parallel friendly. On the contrary, the key routine of the factorized version (the Cholesky factorization routine) is not so parallel friendly, and this affects the scalability of the factorized version. Therefore, we tested a number of implementations of the factorized version, aiming at improving its scalability.

The paper is organized as follows. In section 2 we present an extended formulation of the LQ control problem, and we state conditions for its solution. In section 3 we present a general formulation of the Riccati recursion based solver for the extended LQ control problem. Efficient implementation of both the classical and the factorized version of this Riccati solver are presented in section 4. In section 5 we present the libraries used in our tests, and the result and the discussion of these tests are reported in section 6. Finally, section 7 contains the conclusion.

2. THE EXTENDED LQ CONTROL PROBLEM

In this paper we consider an extended version of the classical LQ control problem: in this formulation, the cost function has a quadratic, a linear and a constants term, and the constraint (given by the equation describing the dynamic system) is affine. Furthermore, all matrices are time variant. The classical and the extended LQ control problems can be solved by means of Riccati recursion based solvers at the same asymptotic cost: the cubic (dominant) terms in the respective cost functions are identical. The main advantage of the extended formulation is that it is flexible enough to describe a wide range of problems (Jørgensen et al. (2012)): in particular, it can be used as sub-routine in AS and IP methods.

Problem 1. The extended LQ control problem is the equality constrained quadratic program

$$\begin{aligned} \min_{u_n, x_{n+1}} \quad & \phi = \sum_{n=0}^{N-1} l_n(x_n, u_n) + l_N(x_N) \\ s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \end{aligned} \tag{1}$$

where $n \in \{0, 1, \ldots, N-1\}$ and

$$l_n(x_n, u_n) = \frac{1}{2} \begin{bmatrix} x_n' & u_n' \end{bmatrix} \begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \begin{bmatrix} q_n' & s_n' \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \rho_n$$

$$l_N(x_N) = \frac{1}{2} x_N' P x_N + p' x_N + \rho_N$$

The state vector x_n has size n_x, the input vector u_n has size n_u, and N is the control horizon length.

Problem (1) can be rewritten in a more compact form as

$$\begin{aligned} \min_{x} \quad & \phi = \frac{1}{2} x' H x + g' x \\ s.t. \quad & Ax = b \end{aligned} \tag{2}$$

where (in the case of $N = 3$)

$$x = \begin{bmatrix} u_0 \\ x_1 \\ u_1 \\ x_2 \\ u_2 \\ x_3 \end{bmatrix}, H = \begin{bmatrix} R_0 & & & & & \\ & Q_1 & S_1' & & & \\ & S_1 & R_1 & & & \\ & & & Q_2 & S_2' & \\ & & & S_2 & R_2 & \\ & & & & & P \end{bmatrix}, g = \begin{bmatrix} \tilde{s}_0 \\ q_1 \\ s_1 \\ q_2 \\ s_2 \\ p \end{bmatrix}$$

$$A = \begin{bmatrix} -B_0 & I & & & & \\ & -A_1 & -B_1 & I & & \\ & & & -A_2 & -B_2 & I \end{bmatrix}, b = \begin{bmatrix} \tilde{b}_0 \\ b_1 \\ b_2 \end{bmatrix}$$

where $\tilde{s}_0 = S_0 x_0 + s_0$ and $\tilde{b}_0 = A_0 x_0 + b_0$. The matrices H and A are large and sparse; in particular, H is block diagonal.

Theorem 1. (KKT (necessary) conditions). If x^* is a solution of problem (2), then there exists a vector π^* of size $N \cdot n_x$ such that

$$\begin{bmatrix} H & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} x^* \\ \pi^* \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix} \tag{3}$$

System (3) is the KKT system associated with problem (2), and in the case of the extended LQ control problem the KKT matrix is large (of size $(2n_x + n_u)N \times (2n_x + n_u)N$) and sparse.

Sufficient conditions for existence and uniqueness of the solution of problem (2) are given in the following theorem.

Theorem 2. (Sufficient conditions). Let the matrices P and $\begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix}$ be positive semi-definite, and the matrices R_n be positive definite for all $n \in \{0, 1, \ldots, N-1\}$, then problem (2) has one and only one solution, given by the solution of the KKT system (3).

The proof of both theorems can be found in Frison (2012).

If the hypothesis of theorem 2 are satisfied and if the matrices Q_n, R_n and P are symmetric, then the KKT system (3) is a symmetric indefinite system of linear equations. In the following we assume that these hypothesis hold.

3. RICCATI RECURSION FOR SOLVING LQ CONTROL PROBLEMS

As shown in Wright (1997), the KKT system (3) can be rewritten in band diagonal form as

$$\begin{bmatrix} R_0 & B_0' & & & & & & & & \\ B_0 & -I & & & & & & & & \\ & -I & Q_1 & S_1' & A_1' & & & & & \\ & & S_1 & R_1 & B_1' & & & & & \\ & & A_1 & B_1 & -I & & & & & \\ & & & & -I & Q_2 & S_2' & A_2' & & \\ & & & & & S_2 & R_2 & B_2' & & \\ & & & & & A_2 & B_2 & -I & & \\ & & & & & & & -I & P & \end{bmatrix} \begin{bmatrix} u_0 \\ \pi_1 \\ x_1 \\ u_1 \\ \pi_2 \\ x_2 \\ u_2 \\ \pi_3 \\ x_3 \end{bmatrix} = \begin{bmatrix} -\tilde{s}_0 \\ -\tilde{b}_0 \\ -q_1 \\ -s_1 \\ -b_1 \\ -q_2 \\ -s_2 \\ -b_2 \\ -p \end{bmatrix} \tag{4}$$

and solved in time $\mathcal{O}(N(n_x + n_u)^3)$ by using the Riccati recursion to factorize the KKT system. A Riccati recursion based solver for problem (1) is summarized in Algorithm 1 (see Frison (2012); Jørgensen (2005)).

Algorithm 1 Riccati recursion based solver for the extended LQ control problem (1)

$P_N \leftarrow P$
$p_N \leftarrow p$
for $n = N-1 \rightarrow 0$ **do**
  $R_{e,n} \leftarrow R_n + B_n' P_{n+1} B_n$
  $K_n \leftarrow -R_{e,n}^{-1}(S_n + B_n' P_{n+1} A_n)$
  $P_n \leftarrow Q_n + A_n' P_{n+1} A_n - K_n' R_{e,n} K_n$
  $k_n \leftarrow -R_{e,n}^{-1}(s_n + B_n'(P_{n+1} b_n + p_{n+1}))$
  $p_n \leftarrow q_n + A_n'(P_{n+1} b_n + p_{n+1}) - K_n' R_{e,n} k_n$
end for
$\pi_0 \leftarrow P_0 x_0 + p_0$
for $n = 0 \rightarrow N-1$ **do**
  $u_n \leftarrow K_n x_n + k_n$
  $x_{n+1} \leftarrow A_n x_n + B_n u_n + b_n$
  $\pi_{n+1} \leftarrow P_{n+1} x_{n+1} + p_{n+1}$
end for

4. EFFICIENT IMPLEMENTATION OF THE RICCATI RECURSION BASED SOLVER

In this section we present a detailed description of two different implementations of the Riccati recursion based solver.

We will focus our attention to the case $n_x > n_u$: this means that, at each iteration, the most expensive part is the computation of the term $A_n' P_{n+1} A_n$. This expression has some structure: the left matrix A_n' is equal to the transpose of the right matrix A_n; furthermore, the central matrix P_{n+1} is symmetric and positive semi-definite (proof in Frison (2012)). It is possible to exploit this structure, as shown in the following.

We assume that all matrices are stored in memory in column-major (Fortran-like) order, and we make use of optimized BLAS and LAPACK routines for linear algebra operations. The following discussion can be easily adapted to the case of row-major (C-like) order, using C wrappers to BLAS and LAPACK.

4.1 Classical version

In this version, the term $A_n' P_{n+1} A_n$ is implemented as

$$A'_n \cdot (P'_{n+1} \cdot A_n)$$

by exploiting the symmetry of the P_{n+1} matrix: since the matrices are stored in column-major order, the best performance in the matrix-matrix multiplication is obtained if the left matrix is transposed and the right is not. The two matrix-matrix multiplications are performed using the BLAS general matrix-matrix multiplication routine `dgemm`. The computation of the expression requires roughly $4n_x^3$ flops.

The expressions $B'_n P_{n+1} B_n$ and $B'_n P_{n+1} A_n$ are computed in a similar way, as $B'_n \cdot (P'_{n+1} \cdot B'_n)$ (cost $2n_x^2 n_u + 2n_x n_u^2$ flops) and $(P'_{n+1} B_n)' \cdot A_n$ (cost $2n_x^2 n_u$ flops, re-using the already computed expression $P'_{n+1} B_n$).

The matrix $R_{e,n}$ is symmetric positive definite (since R_n is symmetric positive definite, and $B'_n P_{n+1} B_n$ is symmetric positive semi-definite): it can be factorized using the LAPACK Cholesky factorization routine `dpotrf`, obtaining the lower triangular factor Λ_n. This costs $\frac{1}{3} n_u^3$ flops.

About the computation of the term $K'_n R_{e,n} K_n$, we have

$$K'_n R_{e,n} K_n = M'_n R_{e,n}^{-1} R_{e,n} R_{e,n}^{-1} M_n = M'_n R_{e,n}^{-1} M_n = \\ = M'_n (\Lambda'_n)^{-1} \Lambda_n^{-1} M_n = (\Lambda_n^{-1} M_n)' (\Lambda_n^{-1} M_n) = L'_n L_n$$

where $M_n = S_n + B'_n P_{n+1} A_n$, and $L_n = \Lambda_n^{-1} M_n$. The operation $\Lambda_n^{-1} M_n$ is performed using the BLAS routine `dtrsm`, requiring $n_x n_u^2$ flops.

The equations for updating vectors are implemented in a similar way, even if their contribution to the total computation time is negligible.

In case of unstable systems, numerical evidence shows that the stability of the algorithm is improved by ensuring the symmetry of matrix P_n by means of the term $P_n \leftarrow 0.5(P_n + P'_n)$ (Jørgensen (2005)). There is not a BLAS or LAPACK routine implementing this operation, and we suggest to implement a blocked version in order to reduce cache misses, with block size equal to the cache line size.

The overall algorithm requires

$$N \left(4n_x^3 + 6n_x^2 n_u + 3n_x n_u^2 + \tfrac{1}{3} n_u^3 \right)$$

flops. The algorithm is summarized in Algorithm 2.

Algorithm 2 Efficient implementation of Riccati recursion based solver solver, classical version

$P_N \leftarrow P$
$p_N \leftarrow p$
for $n = N-1 \rightarrow 0$ **do**
  $R_{e,n} \leftarrow R_n + B'_n \cdot (P'_{n+1} \cdot B_n)$
  $\Lambda_n \leftarrow \mathrm{chol}(R_{e,n}, \text{'lower'})$
  $L_n \leftarrow \Lambda_n^{-1} (S_n + (P'_{n+1} B_n)' \cdot A_n)$
  $P_n \leftarrow Q_n + A'_n \cdot (P'_{n+1} \cdot A_n) - L'_n \cdot L_n$
  $P_n \leftarrow 0.5(P_n + P'_n)$
  $l_n \leftarrow \Lambda_n^{-1} (s_n + B'_n \cdot (P_{n+1} \cdot b_n + p_{n+1}))$
  $p_n \leftarrow q_n + A'_n \cdot (P_{n+1} b_n + p_{n+1}) - L'_n \cdot l_n$
end for
$\pi_0 \leftarrow P_0 \cdot x_0 + p_0$
for $n = 0 \rightarrow N-1$ **do**
  $u_n \leftarrow -(\lambda'_n)^{-1} (L_n \cdot x_n + l_n)$
  $x_{n+1} \leftarrow A_n \cdot x_n + B_n \cdot u_n + b_n$
  $\pi_{n+1} \leftarrow P_{n+1} \cdot x_{n+1} + p_{n+1}$
end for

4.2 Factorized version

This version requires all matrices P_n to be positive definite: a sufficient condition for this is the further hypothesis that all matrices Q_n and P are positive definite Frison (2012).

The term $A'_n P_{n+1} A_n$ is implemented as

$$(\mathcal{L}'_n \cdot A_n)' \cdot (\mathcal{L}'_n A_n)$$

where $\mathcal{L}$ is the lower triangular factor of the Cholesky factorization of P_{n+1}. The advantage of this implementation is that the product $\mathcal{L}'_n \cdot A_n$ can be computed using the BLAS routine `dtrmm`, requiring n_x^3 flops, and the product $(\mathcal{L}'_n A_n)' \cdot (\mathcal{L}'_n A_n)$ of a matrix and its transposed can be computed using the BLAS routine `dsyrk`, requiring n_x^3 flops. Since the cost of the Cholesky factorization is roughly $\frac{1}{3} n_x^3$ flops, the total complexity is roughly $\frac{7}{3} n_x^3$ flops.

Using the LAPACK routine `dpotrf`, the computation of the lower factor is slightly less efficient than the computation of the upper factor; on the other hand, the lower factor gives the advantage that in each matrix-matrix multiplication the left matrix factor is transposed and the right matrix factor is not, exploiting the data order in memory.

In a similar way, the term $B'_n P_{n+1} B_n$ is computed as $(\mathcal{L}'_n \cdot B_n)' \cdot (\mathcal{L}'_n B_n)$, at the cost of $n_x^2 n_u + n_x n_u^2$ (re-using the factorization of P_{n+1}), and the term $B'_n P_{n+1} A_n$ is computed as $(\mathcal{L}'_n B_n)' \cdot (\mathcal{L}'_n A_n)$, at the cost of $2n_x^2 n_u$ flops (re-using the products $(\mathcal{L}'_n A_n)$ and $(\mathcal{L}'_n B_n)$).

The term $K'_n R_{e,n} K_n$ is computed again as in the classical version, except that the term $L'_n \cdot L_n$ is computed using the BLAS routine `dsyrk` instead of `dgemm`. The use of `dsyrk` implies that only the lower triangular part of P_{n+1} can be referenced: the terms $P_{n+1} \cdot b_n$ and $P_{n+1} \cdot x_{n+1}$ are then computed using the BLAS routine `dsymv` instead of `dgemv`.

The total cost of the algorithm is

$$N \left(\tfrac{7}{3} n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \tfrac{1}{3} n_u^3 \right)$$

flops, lower than the cost of the classical version. In the case of n_x large and $n_x \gg n_u$, the theoretical cost of the classical version is approximately $\frac{12}{7} = 1.71$ times the cost of the factorized version. The algorithm is summarized in Algorithm 3.

5. LIBRARIES

In this section we want to briefly describe the libraries used in the code to perform linear algebra operations.

5.1 OpenBLAS

The BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage) libraries are provided by OpenBLAS [1], version 0.2.6. OpenBLAS is an open-source project (BSD license) that aims to extend GotoBLAS to the most recent architectures (e.g. Intel SandyBridge with AVX instruction set). It provides an optimized implementation of all BLAS and part of LAPACK routines: in particular, it provides an optimized implementation of the Cholesky factorization routine `dpotrf`. The

[1] see http://xianyi.github.com/OpenBLAS/

Algorithm 3 Efficient implementation of Riccati recursion based solver, factorized version

$P_N \leftarrow P$
$p_N \leftarrow p$
for $n = N-1 \rightarrow 0$ **do**
  $\mathcal{L} \leftarrow \text{chol}(P_{n+1}, '\text{lower}')$
  $R_{e,n} \leftarrow R_n + (\mathcal{L}' \cdot B_n)' \cdot (\mathcal{L}' B_n)$
  $\Lambda_n \leftarrow \text{chol}(R_{e,n}, '\text{lower}')$
  $L_n \leftarrow \Lambda_n^{-1}(S_n + (\mathcal{L}' B_n)' \cdot (\mathcal{L}' \cdot A_n))$
  $P_n \leftarrow Q_n + (\mathcal{L}' A_n)' \cdot (\mathcal{L}' A_n) - L_n' \cdot L_n$
  $l_n \leftarrow \Lambda_n^{-1}(s_n + B_n' \cdot (P_{n+1} \cdot b_n + p_{n+1}))$
  $p_n \leftarrow q_n + A_n' \cdot (P_{n+1} b_n + p_{n+1}) - L_n' \cdot l_n$
end for
$\pi_0 \leftarrow P_0 \cdot x_0 + p_0$
for $n = 0 \rightarrow N-1$ **do**
  $u_n \leftarrow -(\lambda_n')^{-1}(L_n \cdot x_n + l_n)$
  $x_{n+1} \leftarrow A_n \cdot x_n + B_n \cdot u_n + b_n$
  $\pi_{n+1} \leftarrow P_{n+1} \cdot x_{n+1} + p_{n+1}$
end for

remaining part of LAPACK is the library version 3.4.2 build using OpenBLAS as BLAS library.

OpenBLAS provides a parallel implementation of BLAS for shared memory machines, and makes use of Pthreads by default. The number of threads can be chosen by means of the environment variable `OPENBLAS_NUM_THREADS`, or at run time by using the function `openblas_set_num_threads()` in the code. This second option has the advantage to allow different number of threads in different parts of the code. Alternatively, it is possible to directly build a sequential library (without support for multi-threading): if possible, this second option should be preferred, since it avoids the overhead associated with the creation and destruction of threads at run-time.

LAPACK relies upon BLAS for parallelization: in fact, the LAPACK libraries has been written having in mind sequential machines, and can exploit parallelism only by calling parallel implementations of BLAS. This approach, however, limits the scalability of the code with the number of threads, especially for medium size problems.

5.2 PLASMA

PLASMA [2] (Parallel Linear Algebra for Scalable Multicore Architectures) is a project that aims to provide efficient parallel implementation of linear algebra routines on shared memory machines. It is released with BSD license. We tested the version 2.5.0 of the library. The approach is completely different compared to LAPACK's one: the parallelization is not hidden in the BLAS, but it is performed to an higher level. PLASMA needs a sequential implementation of BLAS, and explicitly takes care of parallelization, making use of Pthreads.

The main features are: tile matrix layout (the matrices are stored in memory in sub-matrices of contiguous elements), tile algorithms (exploiting the tale matrix layout, reducing the cache and TLB misses, and optimizing reuse of data in cache), dynamic scheduling (the assignment of the parallel tasks to the processors is made at run time) and asynchronous algorithms (returning before completion, and then allowing a routine to start on the idle processors even if the previous routine has not completed yet).

[2] see http://icl.cs.utk.edu/plasma/

PLASMA is under active development and currently provides many important LAPACK routines (and in particular the Cholesky factorization routine `dpotrf`) together with a tile and asynchronous version of all level 3 BLAS: this allows us to write the entire Riccati recursion algorithm in tile format.

6. NUMERICAL RESULTS

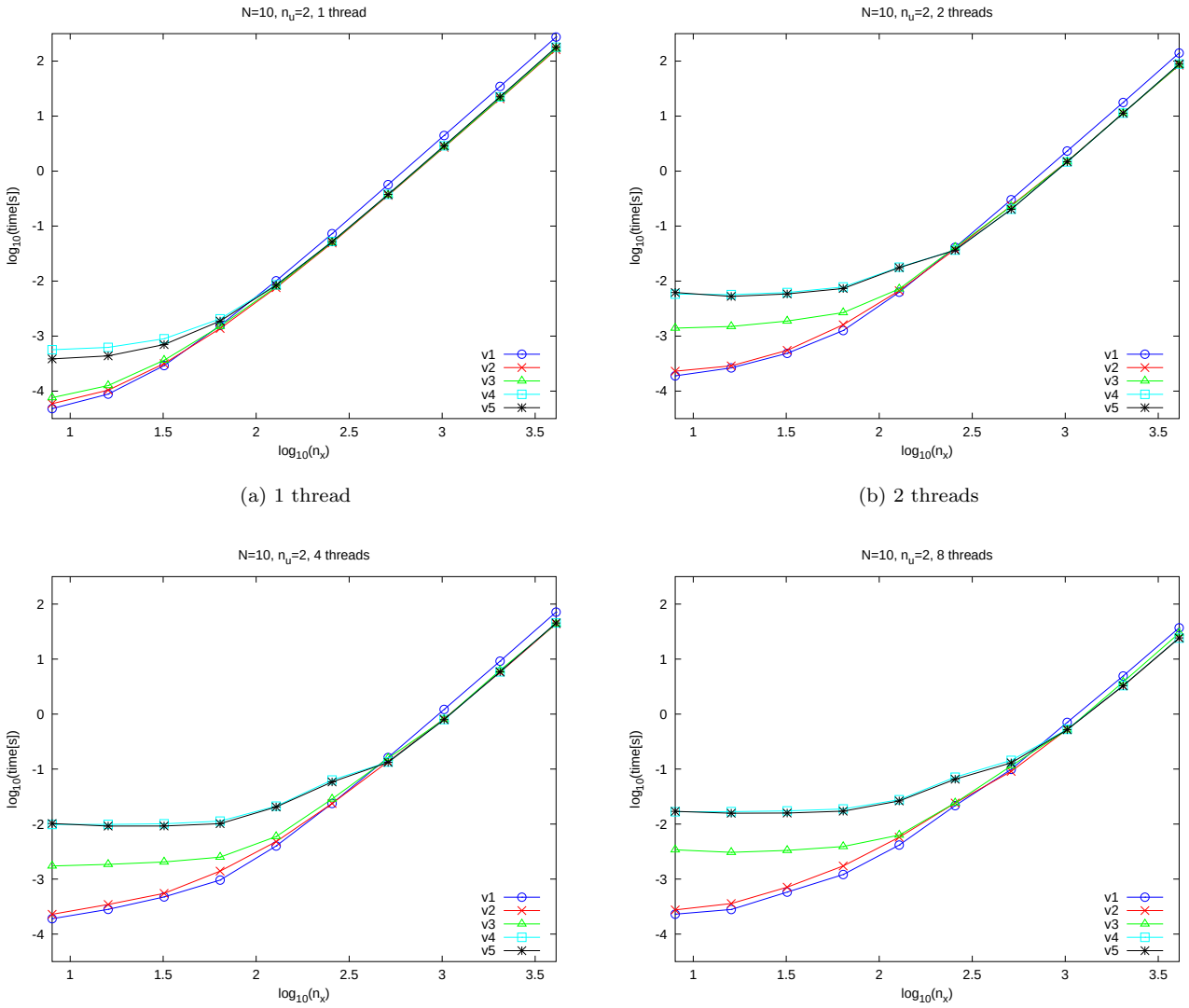
In this section we consider a number of parallel implementations of algorithms (2) and (3) on shared memory machines. We decided to test the following algorithms, that for simplicity we call `v1` to `v5`:

- **v1** implementation of algorithm (2), with BLAS and `dpotrf` provided by parallel OpenBLAS.
- **v2** implementation of algorithm (3), with BLAS and `dpotrf` provided by parallel OpenBLAS.
- **v3** implementation of algorithm (3), with BLAS provided by parallel OpenBLAS and `dpotrf` provided by PLASMA (that makes use of sequential OpenBLAS; in this case sequential and parallel OpenBLAS are given by the same library, and the switch between the two is made at run-time by means of `openblas_set_num_threads()`).
- **v4** implementation of algorithm (3), with level 3 BLAS and `dpotrf` provided by tile version of PLASMA (that makes use of sequential OpenBLAS).
- **v5** implementation of algorithm (3), with level 3 BLAS and `dpotrf` provided by tile and asynchronous version of PLASMA (that makes use of sequential OpenBLAS); routines working on independent data are gathered together into sets, and explicit barrier is used among sets.

The test machine is a HPC node equipped with dual Intel Xeon X5550 processor (in total 8 cores running at 2.66 GHz, 8 MB level 3 cache per socket) running Scientific Linux version 6.1. The processor supports the SSE, SSE2, SSSE3, SSE4_1, SSE4_2 instruction sets.

In figure (1) there are results of numerical tests. About the test problem, the linear system is a randomly-generated time-invariant asymptotically-stable one, while the cost function is strictly quadratic with identity as Hessian: anyway, the special structure of this test problem has not been exploited. In all tests only the number of states has been varied: we investigated the behavior of the proposed algorithms for $n_x \in \{4, 8, 16, 32, 64, 128, 256, 1024, 2048, 4096\}$. The number of inputs was fixed to $n_u = 2$ (its actual value does not influence the performance, as long as $n_u \ll n_x$), and the horizon length to $N = 10$ (its actual value does not influence the results of tests since Riccati recursion is linear in N).

The block size for the tile matrix layout in PLASMA has been chosen equal to $NB = 128$: this is a good trade off between fine-grid parallelism and performance of the sequential BLAS on matrices of size NB. For values of $n_x \leq NB$ the PLASMA routines clearly will reduce to a call to the sequential BLAS, with some overhead. Anyway the largest matrices, of size 4096, are decomposed into $16 \cdot 16 = 256$ blocks, enough to have a fine-grid parallelism.



(a) 1 thread

(b) 2 threads

(c) 4 threads

(d) 8 threads

Fig. 1. Comparison of the different implementations of the Riccati recursion based solver, for the solution of problem (1), for 1,2,4 or 8 threads. Problem size: $N = 10$, n_x varied, $n_u = 2$.

In the test in figure (1a) 1 thread was used. As expected, the implementations making use of PLASMA (i.e. `v3, v4, v5`) suffer a certain overhead for small matrices. For large matrices, all implementations of algorithm 3 (i.e. `v2, v3, v4, v5`) behave in a very similar way, and are faster than the implementation of algorithm 2 (i.e. `v1`), as expected from the theoretical complexity. Anyway, for very small problems, the latter is the fastest, due to the better performance of `dgemm` on small matrices compared to the others level 3 BLAS and `dpotrf` routines. The tile asynchronous implementation `v5` is always slightly faster than the tile synchronous one `v4`, and this is true also for a larger number of threads.

As the number of threads increases to 2, in figure (1b), the overhead associated with implementations making use of PLASMA (i.e. `v3, v4, v5`) increases of an order of magnitude, and it seems proportional to the number of PLASMA routines used per iteration (1 for `v3`, 9 for `v4` and `v5`). For $n_x \in \{256, 512\}$ the tile implementations `v4` and `v5` are slightly faster than the implementation `v2` making use of parallel OpenBLAS. Anyway for larger systems their performance is almost identical.

As the number of threads further increases to 4 (figure (1c)) and 8 (figure (1d)), the trend remains unchanged. In fact, the overhead associated with the use of PLASMA routines increases, and then they become competitive with respect to parallel OpenBLAS only for increasingly larger systems. For large n_x the performance of `v4` and `v5` is almost identical to the one of `v2`, while `v3` is slightly slower. Also the cross-over point between the parallel OpenBLAS implementations of algorithm 2 and algorithm 3 (respectively `v1` and `v2`) moves toward larger values of n_x, since `dgemm` (the key routine in `v1`) is particularly parallel friendly, while `dpotrf` (the key routine in `v2`) is not.

As a result, on the tested machine implementation `v2` making use of OpenBLAS and implementations `v4, v5` making use of PLASMA shows an almost identical per-

	number of threads			
n_x	1	2	4	8
4	0.89	0.58	0.58	0.56
8	0.81	0.82	0.83	0.84
16	0.85	0.92	0.81	0.78
32	0.92	0.88	0.86	0.81
64	1.13	0.78	0.69	0.70
128	1.34	0.94	0.83	0.72
256	1.48	1.08	1.00	0.90
512	1.58	1.28	1.18	1.09
1024	1.64	1.55	1.48	1.34
2048	1.68	1.55	1.60	1.52
4096	1.69	1.67	1.64	1.54

Fig. 2. Speed-up of `v2` with respect to `v1`, computed as $time_{v1}/time_{v2}$. Problem size: $N = 10$, $n_u = 2$.

formance. Anyway the result can be different on shared memory machines with more cores (e.g. PLASMA documentation reports test on machines with 16 or 32 cores). We also notice that, in case of loaded machine, PLASMA shows a smaller decrease in performance compared to OpenBLAS.

In the following we thus analyze more deeply the performance of implementations `v1` (implementing the classical version in algorithm 2) and `v2` (implementing the factorized version in algorithm 3), both making use of OpenBLAS.

In figure 2 there is a table showing the relative speed-up of implementation `v2` compared to `v1`, as function of the number of states n_x and the number of threads. For a given number of threads, implementation `v1` is more efficient for small n_x, while `v2` is more efficient for large n_x. The cross-over points moves toward lager values of n_x as the number of threads increases: this means that `v1` scales better with the number of threads compared to `v2`. Looking at the rows of the table, we can arrive at the same conclusion. In particular it is interesting to notice as, for $n_x = \{64, 128, 256\}$, implementation `v2` is faster in case of 1 thread, but slower in case of 8.

In figure 3 there is a table showing, for both `v1` and `v2`, the speedup obtained using more threads, with respect to the sequential code. The parallel code is faster than the sequential one for $n_x \geq 64$ for `v1`, and $n_x \geq 128$ for `v2`. The efficiency in the use of all available cores increases with the problem size, and again we notice as `v1` has a better scalability than `v2`.

7. CONCLUSION

In this paper we presented two version of Riccati recursion based solver for an extended formulation of the LQ control problems. Algorithm 2 has a worst theoretical complexity but it performs better for small instances; algorithm 3 has a better theoretical complexity, that gives it an advantage for large instances. As the number of threads increases, implementations of algorithm 2 scale better than implementations of algorithm 3. This is due to the fact that the key routine in algorithm 3, the Cholesky factorization, is not parallel friendly.

We tested a number of implementations of algorithm 3, one making use of OpenBLAS, 3 making use of PLASMA

	v1 number of threads			v2 number of threads		
n_x	2	4	8	2	4	8
4	0.61	0.59	0.61	0.40	0.39	0.39
8	0.25	0.25	0.21	0.26	0.26	0.21
16	0.33	0.31	0.31	0.36	0.30	0.29
32	0.60	0.61	0.50	0.57	0.57	0.44
64	1.24	1.63	1.29	0.86	0.99	0.80
128	1.61	2.53	2.43	1.13	1.58	1.31
256	1.77	3.07	3.36	1.30	2.08	2.04
512	1.88	3.50	5.75	1.53	2.63	3.97
1024	1.91	3.65	6.29	1.81	3.28	5.13
2048	1.96	3.78	7.03	1.81	3.60	6.38
4096	1.95	3.83	7.41	1.92	3.70	6.76

Fig. 3. Speed-up obtained using multiple threads, compared to sequential code. Problem size: $N = 10$, $n_u = 2$.

(in the combinations synchronous/asynchronous tile algorithms, and making use or not of the parallel level 3 BLAS provided by PLASMA). On the test machine (with 8 cores), the use of PLASMA does not give significant advantages with respect to OpenBLAS.

As future work, further tests may be performed on machines with a larger number of cores.

REFERENCES

Wright, S.J. (1997). Applying new optimization algorithms to model predictive control. *Fifth International Conference on Chemical Process Control CPC V*, 147-155. CACHE, Tahoe City, California.

Rao, C.V., Wright, S.J., and Rawlings, J.B. (1998). Application of interior-point methods to model predictive control. *Journal of Optimization Theory and Applications*, 99(3), 723-757.

Jørgensen, J.B., Rawlings, J.B., and Jørgensen, S.B. (2004). Numerical methods for large scale moving horizon estimation and control. In *DYCOPS 7*. IFAC, Cambridge, MA.

Frison, G. (2012). *Numerical Methods for Model Predictive Control*. M.Sc. thesis, Department of Informatics and Mathematical Modelling, Technical University of Denmark, Kgs. Lyngby, Denmark.

Jørgensen, J.B., Frison, G., Gade-Nielsen, N.F., Damman, B. (2012). Numerical Methods for Solution of the Extended Linear Quadratic Control Problem. *Proc. IFAC Conf. Nonlinear Model Predictive Control (NMPC'12)*. Noordwijkerhout, The Netherlands, 2012, pp. 187-193.

Jørgensen, J.B. (2005). *Moving Horizon Estimation and Control*. Ph.D. thesis, Department of Chemical Engineering, Technical University of Denmark, Kgs. Lyngby, Denmark.

Frison, G., Jørgensen, J.B. (2013). Efficient Implementation of the Riccati Recursion for Solving Linear-Quadratic Control Problems. Submitted to IEEE MSC 2013.