Cyqlone: A Parallel, High-Performance Linear Solver for Optimal Control

Pieter Pas Panagiotis Patrinos

Abstract

We present Cyqlone, a solver for linear systems with a stage-wise optimal control structure that fully exploits the various levels of parallelism available in modern hardware. Cyqlone unifies algorithms based on the sequential Riccati recursion, parallel Schur complement methods, and cyclic reduction methods, thereby minimizing the required number of floating-point operations, while allowing parallelization across a user-configurable number of processors. Given sufficient parallelism, the solver run time scales with the logarithm of the horizon length (in contrast to the linear scaling of sequential Riccati-based methods), enabling real-time solution of long-horizon problems. Beyond multithreading on multi-core processors, implementations of Cyqlone can also leverage vectorization using batched linear algebra routines. Such batched routines exploit data parallelism using single instruction, multiple data (SIMD) operations, and expose a higher degree of instruction-level parallelism than their non-batched counterparts. This enables them to significantly outperform BLAS and BLASFEO for the small matrices that arise in optimal control. Building on this high-performance linear solver, we develop CyQPALM, a parallel and optimal-control-specific variant of the QPALM quadratic programming solver. It combines the parallel and vectorized linear algebra operations from Cyqlone with a parallel line search and parallel factorization updates, resulting in order-of-magnitude speedups compared to the state-of-the-art HPIPM solver. Open-source C++ implementations of Cyqlone and CyQPALM are available at `https://github.com/kul-optec/cyqlone`.

1 Introduction

The need for the efficient solution of linear-quadratic optimal control problems (OCPs) arises in popular engineering applications such as linear model predictive control (MPC), moving horizon estimation (MHE) and data assimilation (DA). Moreover, numerical solvers that target nonlinear MPC, MHE or DA problems are often based on iterative procedures that solve linearized subproblems, which can in turn be reformulated as linear-quadratic OCPs [37]. As these subproblems account for the majority of the computational effort of the overall solver, improving the performance of linear-quadratic OCP solvers can greatly reduce the solver run time for a wide class of problems. Furthermore, applications like MPC and MHE impose hard limits on the solver run time, while often running on constrained embedded devices. Although general-purpose QP solvers can be used in such settings, OCP-specific solvers such as HPIPM [11] and PIQP multi-stage [41] take advantage of the particular OCP structure, typically resulting in faster run times, lower power consumption, and a smaller memory footprint.

With these requirements in mind, the goal of the present work is the development of a performant numerical linear-quadratic OCP solver that exploits both the inherent parallel structure of OCPs and the various levels of parallelism available in modern processors to minimize solver run time. Specifically, we consider the parallelization and vectorization of the factorization and solution of linear systems with optimal control structure originating from the optimality conditions of linear-quadratic OCPs. This parallel linear solver, which we call Cyqlone, is then used to solve Newton systems in a fully parallel implementation of the augmented Lagrangian-based QPALM method [19, 18, 26]. The resulting CyQPALM solver, a highly performant quadratic programming solver for optimal control problems with inequality constraints, achieves speedups of an order of magnitude or higher compared to the state-of-the-art HPIPM solver, particularly for problems with long horizons.

This work was supported by FWO PhD grant 11M9523N; FWO projects G081222N, G0A0920N, G033822N; KU Leuven internal funding C14/24/103, 12Y7622N;

The authors are with the Department of Electrical Engineering (ESAT), STADIUS Center for Dynamical Systems, Signal Processing and Data Analytics, KU Leuven, Kasteelpark Arenberg 10, 3001 Leuven, Belgium.
e-mail: {pieter.pas,panos.patrinos}@esat.kuleuven.be

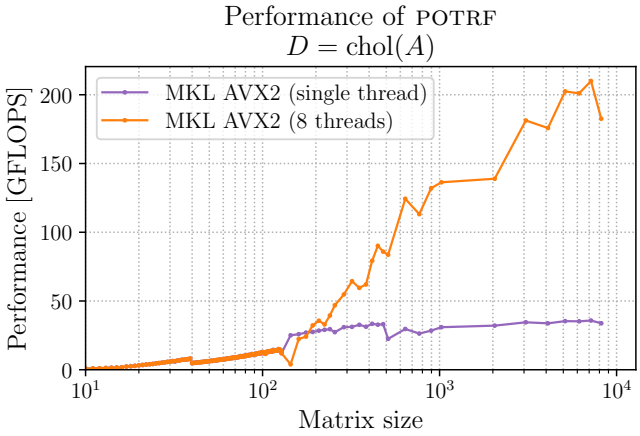

Figure 1. Performance of single-threaded and multithreaded Cholesky factorization (POTRF) routines from the Intel MKL, on an Intel Core Ultra 7 265. The multithreaded variant becomes more performant than the single-threaded variant for matrices of size 192×192 and larger.

1.1 Exploiting the parallelism available in modern hardware

From dual-core Raspberry Pi Pico microcontrollers to 192-core x86-64 server CPUs, multiprocessors dominate the hardware landscape. As gains in clock speed and instructions per cycle have slowed over the past decades, parallel processing has become essential for reaching peak performance on modern systems [17, Ch. 5]. In the specific context of optimal control, existing methods such as the well-known Riccati recursion used in solvers like HPIPM and FATROP [45] are inherently sequential methods that are not directly amenable to parallelization at the algorithmic level because of the recursive backward dynamic programming approach. For very large problems with large numbers of states, individual linear algebra operations can be parallelized using multithreaded BLAS (basic linear algebra subprograms) and LAPACK (linear algebra package) implementations [43]. However, for small matrices, parallelization within a single linear algebra operation comes with significant overhead compared to single-threaded alternatives, largely canceling out the performance gained from using multiple processors [4, Fig. 16], as shown in Figure 1.

For this reason, the CYQLONE method presented in this paper considers parallelization along the time dimension of the OCPs, resulting in high speedups compared to state-of-the-art serial methods, even for very small matrices. Existing solvers that parallelize along the time dimension of OCPs such as parallel Schur complement methods [35, 9], partitioned dynamic programming [49], related parametric subdivision approaches [31, 22] and cyclic reduction methods [30] suffer from a number of limitations, as discussed in more detail in section 9: some methods contain serial bottlenecks by only parallelizing part of the algorithm, they may require significantly higher FLOP (floating-point operation) counts compared to serial alternatives, or they may be suboptimal when the number of available processors is smaller than the horizon length of the OCP. CYQLONE unifies the sequential Riccati recursion, parallel Schur complement methods and cyclic reduction methods to minimize the FLOP count by exploiting the problem-specific structure, while allowing the user to select the amount of parallelism that is most suitable for the hardware at hand.

Efficient linear algebra routines form the basis of an optimized implementation of the CYQLONE solver. Existing optimization solvers for optimal control, such as HPIPM, FATROP, and PIQP multi-stage [41] rely on the linear algebra routines provided by the BLASFEO library [13]. BLASFEO stores the matrices in a specialized packed format, referred to as *panel-major* storage order. This storage format obviates the need for the online packing step performed by general-purpose BLAS implementations [44], improves cache locality, and enables vectorization. These properties make BLASFEO an attractive choice for the small to medium-sized matrices commonly used in solvers for optimal control.

One limitation of the vectorization technique used by BLASFEO is that it is less efficient for small and/or structured (triangular or symmetric) matrices. Additionally, some critical steps in e.g. the Cholesky factorization routine cannot be vectorized effectively. We will see that the parallelism exposed by the CYQLONE solver not only enables parallelization across different processors, but also vectorization along the time dimension of the OCP, or more generally, *batch-wise vectorization*. This vectorization strategy, which was previously inapplicable to solvers for optimal control, is able to outperform BLAS and BLASFEO for small matrices such as the ones arising in optimal control, as demonstrated by Figure 2.

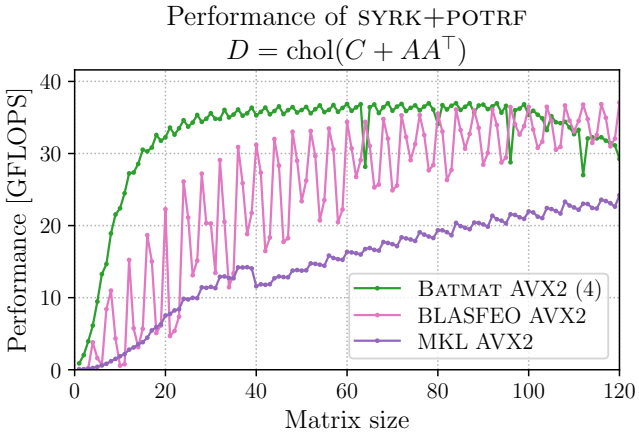

Figure 2. Performance of different (fused) symmetric rank-k update (SYRK) and Cholesky factorization (POTRF) routines [10, §3.3.2] when applied to batches of eight square matrices A and C. The BATMAT implementation using batch-wise vectorization [33] significantly outperforms the BLASFEO and Intel MKL BLAS implementations for small matrices. Unlike BATMAT and BLASFEO, the Intel MKL does not provide a fused SYRK+POTRF implementation, so the individual SYRK and POTRF routines are used instead. Experiments were carried out on an Intel Core Ultra 7 265 with a sustained clock speed of 5.2 GHz and a theoretical peak AVX2 throughput of 41.6 GFLOPS, using BLASFEO 0.1.4.2 and version 2025.0.1 of the Intel MKL.

In the development of CYQLONE and CYQPALM, we consider the following levels of parallelism to fully exploit the performance of modern hardware [17]:

1. **Thread-level parallelism (TLP)**: The proposed CYQLONE solver for linear systems with optimal control structure enables parallelization across different stages of the OCP by partitioning the horizon into smaller sub-intervals which are solved by different processors in parallel. This allows the method to leverage the full computational power of modern multi-core processors, with run times scaling logarithmically with the horizon length if enough processors are available (in contrast to sequential solvers such as the Riccati-based solvers used by HPIPM and FATROP, whose run times scale linearly with horizon length).
2. **Data-level parallelism (DLP)**: Batches of four or eight matrices from different stages are interleaved into a specialized compact storage format, enabling the use of fast batched linear algebra routines. By using single instruction, multiple data (SIMD) operations across such a batch, these routines can be implemented more efficiently than standard BLAS routines using row-wise vectorization for small matrices. When applied to matrices whose dimensions are not multiples of the hardware's vector length, or when applied to triangular or symmetric matrices, vectorization along the time dimension results in improved utilization of the available vector lanes compared to row-wise vectorization.
3. **Instruction-level parallelism (ILP)**: Our implementation of the CYQLONE solver makes use of custom batched linear algebra routines built on top of optimized micro-kernels [33, 44]: these low-level implementations of fundamental linear algebra operations for small blocks of matrices are hand-coded to exploit pipelining, out-of-order execution and superscalar capabilities of modern CPUs. By building blocked algorithms on top of these micro-kernels, keeping in mind the properties of the memory hierarchy, high performance can be achieved for larger matrices as well. Furthermore, by processing matrices in batches rather than one at a time, bottlenecks posed by data dependencies within a matrix (such as dependencies on the pivot in Gaussian elimination) can be avoided, further improving ILP.

1.2 Contributions

The main contributions of this work can be summarized as follows:

1. We present CYQLONE, a new **parallel linear solver for linear–quadratic control problems**. This solver exploits the particular structure of optimal control problems by using a modified Riccati recursion on different intervals of the OCP horizon. Furthermore, it is able to adapt to various degrees of parallelism, depending on how many processors are available, without restrictions on the horizon length, enabling excellent scaling to long horizons and large numbers of processors. (Section 4)

2. A **parallel solver for quadratic programs with optimal control structure** named CyQPALM is developed that leverages CyqLONE as its linear solver. This solver is based on the recently proposed QPALM-OCP method [26], and aims to fully parallelize all steps of the algorithm, including parallel matrix factorizations and back substitutions using CyqLONE, parallel evaluation of the necessary gradients and residuals, and a parallel exact line search. (Section 5)
   A popular optimal control benchmark is used to demonstrate that the CyqLONE and CyQPALM solvers outperform existing state-of-the-art solvers by a significant margin. (Section 8)
3. An **optimized parallel and vectorized C++ implementation** of the CyqLONE solver is developed, which is able to achieve close to the peak performance of the hardware, even for small matrices. The source code is made available under an open-source license [34]. (Section 7)
4. Parallel **factorization update routines** for CyqLONE are derived, and we show how they can further improve the performance of CyQPALM. (Section 6)
5. We perform a detailed **comparison of different solution methods for linear–quadratic control problems**, discussing the tradeoffs in total computational cost and parallelizability. We show how CyqLONE relates to and improves upon existing methods. (Section 9)

2 Problem statement and notation

We consider discrete-time linear-quadratic optimal control problems with inequality constraints of the form

$$\begin{aligned} \underset{\mathbf{u},\mathbf{x}}{\textbf{minimize}} \quad & \sum_{j=0}^{N-1} \ell_j(u^j, x^j) + \ell_N(x^N) \\ \textbf{subject to} \quad & E_0 x^0 = x_{\text{init}} \\ & E_{j+1} x^{j+1} = A_j x^j + B_j u^j + f^j \quad (0 \le j < N) \\ & b_l^j \le C_j x^j + D_j u^j \le b_u^j \quad (0 \le j < N) \\ & b_l^N \le C_N x^N \le b_u^N, \end{aligned} \tag{1}$$

where the convex stage-wise and terminal cost functions are given by

$$\ell_j(u,x) = \tfrac{1}{2}\begin{pmatrix} u \\ x \end{pmatrix}^\top \begin{pmatrix} R_j^\ell & S_j^\ell \\ S_j^{\ell\top} & Q_j^\ell \end{pmatrix} \begin{pmatrix} u \\ x \end{pmatrix} + \begin{pmatrix} u \\ x \end{pmatrix}^\top \begin{pmatrix} r_\ell^j \\ q_\ell^j \end{pmatrix} \quad \text{and} \quad \ell_N(x) = \tfrac{1}{2} x^\top Q_N^\ell x + x^\top q_\ell^N, \tag{2}$$

with $Q_j^\ell \in \mathrm{Sym}_+(\mathbb{R}^{n_x})$, $S_j^\ell \in \mathbb{R}^{n_u \times n_x}$ and $R_j^\ell \in \mathrm{Sym}_+(\mathbb{R}^{n_u})$. The state variables $\mathbf{x} = [x^0 \cdots x^N] \in \mathbb{R}^{n_x \times (N+1)}$ and controls $\mathbf{u} = [u^0 \cdots u^{N-1}] \in \mathbb{R}^{n_u \times N}$ are governed by the linear dynamics described by $A_j \in \mathbb{R}^{n_x \times n_x}$, $B_j \in \mathbb{R}^{n_x \times n_u}$ and $f^j \in \mathbb{R}^{n_x}$. Linear stage-wise mixed state–input constraints are encoded using $C_j \in \mathbb{R}^{n_y \times n_x}$ and $D_j \in \mathbb{R}^{n_y \times n_u}$, with $C_N \in \mathbb{R}^{n_y \times n_x}$ for the terminal constraints. The invertible matrices $E_j \in \mathbb{R}^{n_x \times n_x}$ allow for implicit state equations. Although the matrices E_j are used by some of the methods discussed in Section 9, we will usually assume without loss of generality that $E_j = \mathrm{I}$, which can be achieved by transforming $E_j x^{j+1} = A_j x^j + B_j u^j + f^j$ into $x^{j+1} = E_j^{-1} A_j x^j + E_j^{-1} B_j u^j + E_j^{-1} f^j$.[1]

In the first part of the paper, and in Section 4 specifically, we focus on the following linear-quadratic optimal control problem subject to equality constraints only:

$$\begin{aligned} \underset{\mathbf{u},\mathbf{x}}{\textbf{minimize}} \quad & \sum_{j=0}^{N-1} \varphi_j(x^j, u^j) + \varphi_N(x^N) \\ \textbf{subject to} \quad & E_0 x^0 = x_{\text{init}} \\ & E_{j+1} x^{j+1} = A_j x^j + B_j u^j + f^j, \quad (0 \le j < N) \end{aligned} \tag{3}$$

where the strongly convex stage-wise and terminal cost functions are given by

$$\varphi_j(u,x) = \tfrac{1}{2}\begin{pmatrix} u \\ x \end{pmatrix}^\top \begin{pmatrix} R_j & S_j \\ S_j^\top & Q_j \end{pmatrix} \begin{pmatrix} u \\ x \end{pmatrix} + \begin{pmatrix} u \\ x \end{pmatrix}^\top \begin{pmatrix} r^j \\ q^j \end{pmatrix} \quad \text{and} \quad \varphi_N(x) = \tfrac{1}{2} x^\top Q_N x + x^\top q^N. \tag{4}$$

[1] In fact, this transformation also improves performance in practice, since it lowers the number of operations required when solving linear systems involving the constraints. The upfront cost of computing an LU factorization of E_j is often negligible compared to the resulting performance gains in the optimization solver.

The connection between the general optimal control problem (1) and the strongly convex equality constrained variant (3) will be discussed in Section 5.

After elimination of the fixed initial state x^0, the optimality conditions of (3) are given by

$$\begin{cases} Q_N x^N - E_N^\top \lambda^{N-1} & = -q^N \\ R_j u^j + S_j x^j + B_j^\top \lambda^j & = -r^j \quad (0<j<N) \\ S_j^\top u^j + Q_j x^j + A_j^\top \lambda^j - E_j^\top \lambda^{j-1} & = -q^j \quad (0<j<N) \\ R_0 u^0 + B_0^\top \lambda^0 & = -\tilde{r}^0 \\ B_j u^j + A_j x^j - E_{j+1} x^{j+1} & = -f^j \quad (0<j<N) \\ B_0 u^0 - E_1 x^1 & = -\tilde{f}^0, \end{cases} \tag{5}$$

with $\tilde{r}^0 = r^0 + S_0 E_0^{-1} x_{\text{init}}$ and $\tilde{f}^0 = f^0 + A_0 E_0^{-1} x_{\text{init}}$. The Lagrange multipliers $\lambda^j \in \mathbb{R}^{n_x}$ correspond to the dynamics constraints between stage j and stage $j+1$. We will refer to linear systems of the form (5) as *KKT systems with optimal control structure.*

3 Preliminaries

We start by reviewing some facts about the block Cholesky factorization that will be used throughout the remainder of this paper. The block Cholesky factorization formula introduced below is key in deriving the CYQLONE algorithm; the cyclic reduction algorithm is used to solve block-tridiagonal systems in parallel; and the concepts of fill-in and the elimination tree are essential in analyzing the efficiency and parallelizability of linear solvers.

3.1 Block Cholesky factorizations

A 2×2 block Cholesky factorization $LDL^\top$ of a symmetric matrix H with L block lower triangular and D block diagonal satisfies

$$\begin{aligned} H = \begin{pmatrix} H_{11} & H_{21}^\top \\ H_{21} & H_{22} \end{pmatrix} &= \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} D_1 \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix}^\top + \begin{pmatrix} 0 & 0 \\ 0 & H/H_{11} \end{pmatrix} = \underbrace{\begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} D_1 & \\ & D_2 \end{pmatrix}}_{D} \underbrace{\begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix}^\top}_{L^\top} \\ \text{where} \quad H_{11} &= L_{11} D_1 L_{11}^\top \\ L_{21} &= H_{21} L_{11}^{-\top} D_1^{-1} \\ H/H_{11} &\triangleq H_{22} - H_{21} H_{11}^{-1} H_{21}^\top = H_{22} - L_{21} D_1 L_{21}^\top = L_{22} D_2 L_{22}^\top. \end{aligned} \tag{6}$$

The block H/H_{11} is known as the Schur complement of H_{11} in H. The matrices L_{ii} and D_i are not unique: we only require that the inverses of L_{ii} and D_i are easy to apply, and that D_i is symmetric. In practice, we will restrict ourselves to lower and upper triangular matrices L_{ii}, and signature matrices D_i.[2] This leaves some freedom in how we choose L_{ii} and D_i: a first design parameter is the size of the block H_{11}, a second is the factorization of H_{11}, and a third is the factorization of H/H_{11}. The size of H_{11} will usually follow from the block structure of the original matrix H. If an "obvious" factorization of H_{11} or H/H_{11} exists, we can use it as our choice of L_{ii} and D_i. Specifically, if H_{11} is a 1×1 matrix, we use the scalar factorization $L_{11} = \sqrt{|H_{11}|}$ and $D_1 = \mathbf{sgn}(H_{11})$, and similarly for the case where H/H_{11} is a 1×1 matrix. If no such factorization is available, we instead use another smaller block Cholesky factorization, obtained by recursive application of (6) to H_{11} and/or H/H_{11}.

The general block Cholesky factorization procedure can be summarized as follows:

1. Isolate the top-left block H_{11};
2. Factorize H_{11}: select L_{11} and D_1 such that $H_{11} = L_{11} D_1 L_{11}^\top$ (either using specific knowledge about H_{11} or by recursively computing a block Cholesky factorization);
3. Compute the subdiagonal block $L_{21} = H_{21} L_{11}^{-\top} D_1^{-1}$;
4. Subtract the product $L_{21} D_1 L_{21}^\top$ from the block H_{22} to compute the Schur complement H/H_{11};
5. Factorize H/H_{11}: select L_{22} and D_2 such that $H/H_{11} = L_{22} D_2 L_{22}^\top$ (either using specific knowledge about H/H_{11} or by recursively computing a block Cholesky factorization).

[2] Signature matrices are diagonal matrices with elements ± 1.

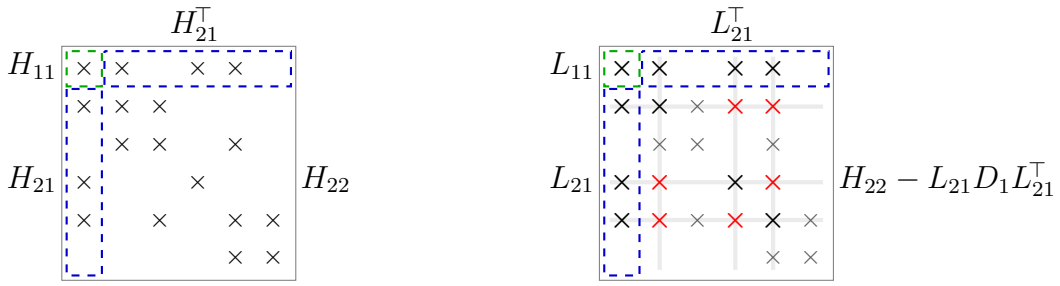

Figure 3. Visualization of steps 1–4 of the block Cholesky factorization of a sparse symmetric matrix H as in (6). Nonzero blocks of H are indicated by crosses ($\times$). Blocks that are modified by the elimination of H_{11} and H_{21} are shown in bold, and blocks of fill-in in the Schur complement are shown in red.

When computing $LDL^\top$ factorizations of sparse matrices, an important consideration is *fill-in*: the outer product $L_{21}D_1L_{21}^\top$ is nonzero at the intersections of the rows where L_{21} is nonzero and the columns where $L_{21}^\top$ is nonzero. Consequently, the matrix H/H_{11} may be significantly less sparse than H_{22}. Zero entries of H_{22} that become nonzero in H/H_{11} are referred to as fill-in. Figure 3 visualizes the phenomenon for a small example matrix. Such fill-in may cause more fill-in down the line, and causes the factorization of H/H_{11} to require more floating-point operations. Practical sparse solvers therefore use (heuristic) fill-in-reducing permutations of the original matrix H to maintain sparsity.

3.2 Cyclic reduction of block-tridiagonal linear systems

Symmetric (block-)tridiagonal linear systems play a crucial role in the solution of optimal control and other dynamic optimization problems, because they can be used to represent coupling between successive time steps, as is the case in (5). In this section, we discuss solution methods for block-tridiagonal systems, and in particular, the cyclic reduction algorithm for solving such systems in parallel. Consider a symmetric positive definite block-tridiagonal system of equations of size N with diagonal blocks $M_k \in \mathrm{Sym}_{++}(\mathbb{R}^n)$ and subdiagonal blocks $K_k \in \mathbb{R}^{n\times n}$:

$$\mathcal{M}\mathbf{x} = \begin{pmatrix} M_0 & K_0^\top & & & \\ K_0 & M_1 & K_1^\top & & \\ & K_1 & \ddots & \ddots & \\ & & \ddots & M_{N-2} & K_{N-2}^\top \\ & & & K_{N-2} & M_{N-1} \end{pmatrix} \begin{pmatrix} x^0 \\ x^1 \\ \vdots \\ x^{N-2} \\ x^{N-1} \end{pmatrix} = \begin{pmatrix} b^0 \\ b^1 \\ \vdots \\ b^{N-2} \\ b^{N-1} \end{pmatrix} = \mathbf{b}. \tag{7}$$

Symmetric block-tridiagonal matrices can be factorized using the block Cholesky factorization outlined in the previous section. Thanks to the special structure, only the top block of L_{21} will be nonzero, which means that only a single block of H_{22} needs to be updated, without any fill-in. If $M_0 = L_{11}D_1L_{11}^\top$, then

$$L_{21} = \begin{pmatrix} K_0 L_{11}^{-\top} D_1^{-1} \\ \hline 0 \\ \vdots \\ 0 \end{pmatrix}, \qquad \mathcal{M}/M_0 = \left(\begin{array}{c|ccc} M_1 - K_0 L_{11}^{-\top} D_1^{-1} L_{11}^{-1} K_0^\top & K_1^\top & & \\ \hline K_1 & \ddots & \ddots & \\ & \ddots & M_{N-2} & K_{N-2}^\top \\ & & K_{N-2} & M_{N-1} \end{array}\right). \tag{8}$$

This block-tridiagonal factorization procedure requires N steps and produces no fill-in. However, due to the coupling through the off-diagonal blocks, this process is inherently serial: the block M_k can only be updated and factorized once M_{k-1} and K_{k-1} have been eliminated, because of the direct coupling between equations $k-1$ and k. A well-known approach that avoids this serial bottleneck is *cyclic reduction* (CR) [16, 14, 2].

Cyclic reduction is built on the idea of simultaneous elimination of the odd block rows: Since there is no direct coupling between equations $2k-1$ and $2k+1$, their elimination can be performed in parallel. After eliminating the odd equations of a system of size N, the remaining equations form a smaller block-tridiagonal system of size $\lceil \frac{N}{2} \rceil$, and the process can be repeated recursively, for a total of $\lceil \log_2(N) \rceil + 1$ steps. Because equations $2k-1$ and $2k+1$ are still coupled (indirectly) through equation $2k$, this remaining even equation needs to be updated. Consequently, we will find that each block row elimination in CR causes one block of fill-in. This is a disadvantage compared to the straightforward block-tridiagonal Cholesky factorization from (8), but the increased number of operations per row is often an acceptable trade-off for the gained parallelism, especially for large N.

3.2.1 Derivation via simultaneous elimination of odd equations

A single application of the cyclic reduction algorithm involves updating the even block rows $2i$ by eliminating the odd block rows $2i \pm 1$ around them. Let us consider three equations from (7) in isolation, for $0 \leq i < \frac{N}{2}$, with the boundary conditions $K_{-1} = 0 = K_{N-1}$.

$$\begin{cases} K_{2i-2}\, x^{2i-2} \;+\; M_{2i-1}\, x^{2i-1} \;+\; K_{2i-1}^\top\, x^{2i} & & & = b^{2i-1} \\ \qquad K_{2i-1}\, x^{2i-1} \;+\; M_{2i}\, x^{2i} \;+\; K_{2i}^\top\, x^{2i+1} & & & = b^{2i} \\ \qquad\qquad K_{2i}\, x^{2i} \;+\; M_{2i+1}\, x^{2i+1} \;+\; K_{2i+1}^\top\, x^{2i+2} & & & = b^{2i+1} \end{cases} \tag{9}$$

We can eliminate x^{2i-1} from equation $2i$ by multiplying equation $2i-1$ by $K_{2i-1}M_{2i-1}^{-1}$ and subtracting it from equation $2i$. Similarly, subtracting $K_{2i}^\top M_{2i+1}^{-1}$ times equation $2i+1$ eliminates x^{2i+1}.

$$\begin{cases} K_{2i-1}M_{2i-1}^{-1}K_{2i-2}\, x^{2i-2} + \cancel{K_{2i-1}\, x^{2i-1}} + K_{2i-1}M_{2i-1}^{-1}K_{2i-1}^\top\, x^{2i} & & & = K_{2i-1}M_{2i-1}^{-1}\, b^{2i-1} \\ \qquad \cancel{K_{2i-1}\, x^{2i-1}} + M_{2i}\, x^{2i} & & + \cancel{K_{2i}^\top\, x^{2i+1}} & = b^{2i} \\ \qquad\qquad K_{2i}^\top M_{2i+1}^{-1}K_{2i}\, x^{2i} & & + \cancel{K_{2i}^\top\, x^{2i+1}} + K_{2i}^\top M_{2i+1}^{-1}K_{2i+1}^\top\, x^{2i+2} & = K_{2i}^\top M_{2i+1}^{-1}\, b^{2i+1}. \end{cases} \begin{matrix} \\ \swarrow - \\ \\ \nwarrow - \end{matrix}$$

By introducing $L_k \triangleq \mathrm{chol}(M_k)$, $Y_k \triangleq K_k L_k^{-\top}$, $U_k \triangleq K_{k-1}^\top L_k^{-\top}$ and $\tilde{b}^k \triangleq L_k^{-1} b^k$, this simplifies to

$$\begin{cases} Y_{2i-1}U_{2i-1}^\top\, x^{2i-2} + \cancel{K_{2i-1}\, x^{2i-1}} + Y_{2i-1}Y_{2i-1}^\top\, x^{2i} & & & = Y_{2i-1}\, \tilde{b}^{2i-1} \\ \qquad \cancel{K_{2i-1}\, x^{2i-1}} + M_{2i}\, x^{2i} & & + \cancel{K_{2i}^\top\, x^{2i+1}} & = b^{2i} \\ \qquad\qquad U_{2i+1}U_{2i+1}^\top\, x^{2i} & & + \cancel{K_{2i}^\top\, x^{2i+1}} + U_{2i+1}Y_{2i+1}^\top\, x^{2i+2} & = U_{2i+1}\, \tilde{b}^{2i+1}. \end{cases} \begin{matrix} \\ \swarrow - \\ \\ \nwarrow - \end{matrix} \tag{10}$$

After subtracting equations $2i \pm 1$ from equation $2i$, the resulting equation contains only even variables:

$$-Y_{2i-1}U_{2i-1}^\top\, x^{2i-2} + \left(M_{2i} - Y_{2i-1}Y_{2i-1}^\top - U_{2i+1}U_{2i+1}^\top\right) x^{2i} - U_{2i+1}Y_{2i+1}^\top\, x^{2i+2} = b^{2i} - Y_{2i-1}\, \tilde{b}^{2i-1} - U_{2i+1}\, \tilde{b}^{2i+1}. \tag{11}$$

This is another tridiagonal system, half the size of the original system, with diagonal blocks $M_{2i}^{(1)} \triangleq M_{2i} - Y_{2i-1}Y_{2i-1}^\top - U_{2i+1}U_{2i+1}^\top$ and subdiagonal blocks $K_{2i-2}^{(1)} \triangleq -Y_{2i-1}U_{2i-1}^\top$. It can either be solved directly, or by another recursive application of CR. Once the even variables x^{2i} have been determined, the odd variables can be recovered from the odd equations in (9). Multiplying by L_{2i-1}^{-1} and rearranging, clearly x^{2i-1} solves

$$L_{2i-1}^\top\, x^{2i-1} = \tilde{b}^{2i-1} - U_{2i-1}^\top\, x^{2i-2} - Y_{2i-1}^\top x^{2i}. \tag{12}$$

The full recursive procedure for solving a symmetric block-tridiagonal linear system using cyclic reduction is described by Algorithm 5 in Appendix A.

3.2.2 Cyclic reduction as block Cholesky factorization

Both (block) cyclic reduction and (block) Cholesky factorization are instances of Gaussian elimination, and there exists a natural equivalence between the two methods: Block cyclic reduction of a block-tridiagonal matrix can be interpreted as the block Cholesky factorization of a specific permutation of the original matrix [14, (3)], where the block rows and columns are ordered by increasing 2-adic valuation ν_2, i.e. the multiplicity of two in the prime factorization of their index i in the original block-tridiagonal matrix:

$$\nu_2(i) \triangleq \begin{cases} \infty & \text{if } i = 0, \\ \max\left\{m \in \mathbb{N} \;\middle|\; 2^m \mid i\right\} & \text{otherwise.} \end{cases} \tag{13}$$

All block rows/columns with odd indices (valuation 0) are ordered first, then the block rows/columns whose indices are divisible by 2 but not 4 (valuation 1), then indices divisible by 4 but not 8 (valuation 2), etc. The block row/column with index 0 (valuation ∞) is ordered last. This permutation gives rise to block diagonal submatrices of decreasing sizes ($N/2$, $N/4$, ...) that can be factorized in parallel, as visualized in Figure 4. Figure 11 shows the same representation of CR for a larger matrix, clearly highlighting the self-similar, recursive structure of the method.

For practical reasons, it is easier to label the subdiagonal blocks in the permuted block-tridiagonal matrix by the column they belong to: we define $\vec{K}_k \triangleq K_k$ and $\overleftarrow{K}_k \triangleq K_{k-1}^\top$ (for odd k). For example, as can be seen in Figure 4, $\vec{K}_1 \triangleq K_1$ is the subdiagonal block that couples column 1 to column 2 (increasing the index, hence the forward arrow), and $\overleftarrow{K}_1 \triangleq K_0^\top$ is the subdiagonal block that couples column 1 to column 0 (decreasing the index, hence the backward arrow). This results in more symmetric formulas $Y_k = \vec{K}_k L_k^{-\top}$ and $U_k = \overleftarrow{K}_k L_k^{-\top}$.

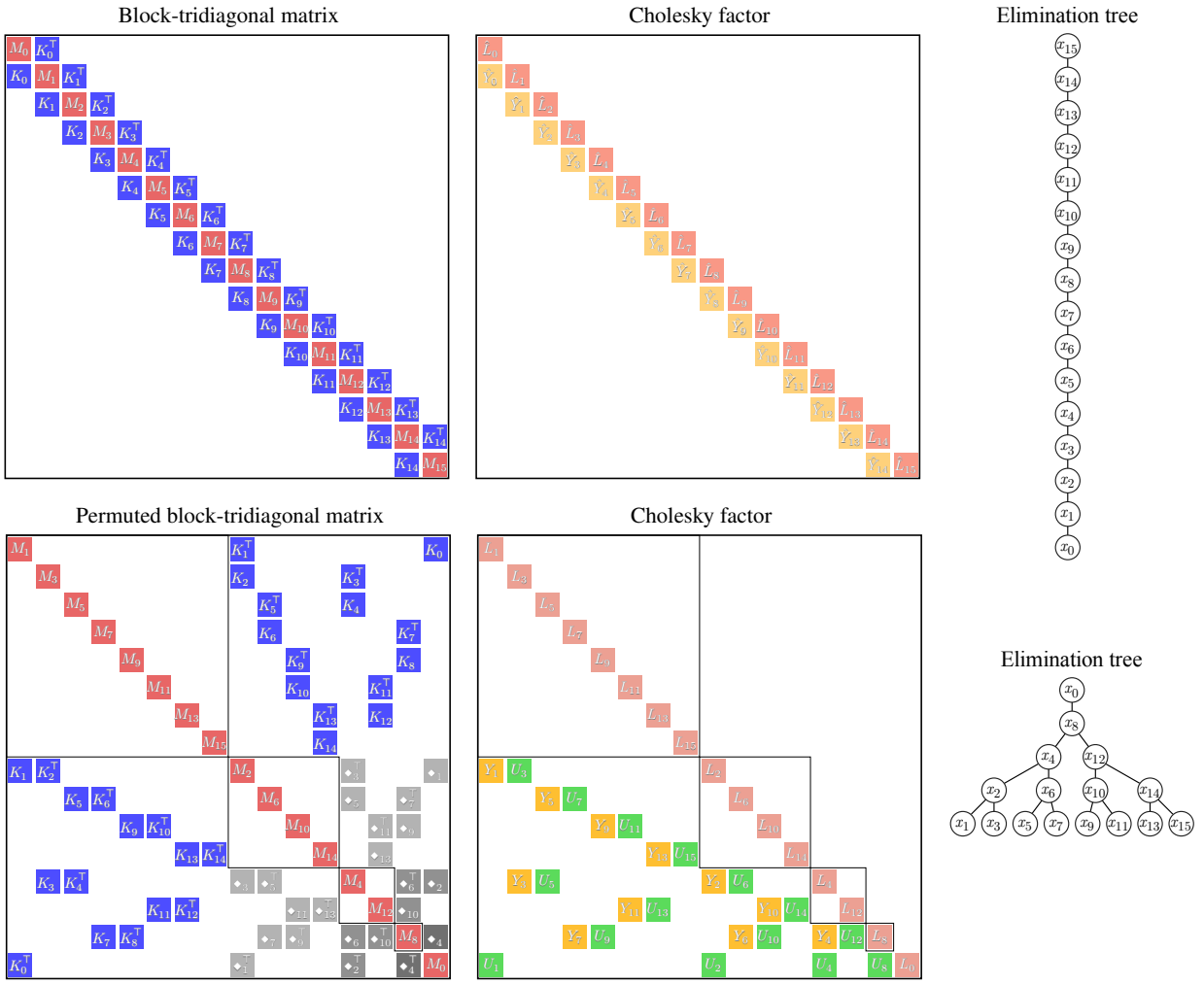

Figure 4. Top: a 16×16 symmetric block-tridiagonal matrix with its block Cholesky factor and the corresponding elimination tree. No fill-in is incurred, but the single linear path in the elimination tree highlights the sequential nature of this factorization procedure. Bottom: cyclic reduction as the factorization of a permutation of the same block-tridiagonal matrix, along with its Cholesky factor (with fill-in generated during the factorization shown in gray and labels matching the derivation in Section 3.2.1). Blocks in the square outlines along the diagonal can be factorized simultaneously. The parallel branches in the elimination tree visualize the available parallelism during the Cholesky factorization of the permuted matrix.

3.2.3 Parallelism and the elimination tree

A tool that helps visualize the available parallelism in the block Cholesky factorization of a given sparse matrix is the *elimination tree* [6, Ch. 4]. Each node of such a tree represents a variable in the system (or a block column in the matrix), and the edges indicate the order in which the variables are eliminated: a node can only be eliminated once all its children have been eliminated. The elimination tree of the original block-tridiagonal matrix (7) is a linear graph: the variables have to be eliminated in order and one at a time. In contrast, the elimination tree of the permuted matrix in Figure 4 is a perfect binary tree with many leaf nodes that can be eliminated in parallel. These leaf nodes are exactly the odd variables that are eliminated in the first step of cyclic reduction. Thanks to this parallelism, the total height of the elimination tree for the CR method ($\log_2(N) + 1$) is much lower than that of the original block-tridiagonal matrix (N). This height difference indicates that the wall-clock time of an optimal parallel implementation of CR could be significantly lower than for the block-tridiagonal Cholesky method. Because of the additional fill-in, the asymptotic reduction in wall time of CR compared to the Cholesky factorization of the block-tridiagonal system for the case of $P = N$ processors turns out to be $\frac{10\log_2(N)+1}{7(N-1)+1}$[3].

[3]Based on the theoretical FLOPs in the critical path, and performing as much of Algorithm 5 in parallel as possible, including the loops, as well as lines 12 and 13 and lines 19 and 20

4 CYQLONE: Parallel factorization and solution of KKT systems with optimal control structure

This section describes a parallelizable algorithm for the factorization of a particular permutation of the coefficient matrix of the KKT system with optimal control structure introduced in (5). We will refer to this ordering of the coefficient matrix and the corresponding factorization algorithm as the CYQLONE method. Figure 5 shows an illustrative example for an OCP with horizon $N = 12$ and parallelization across $P = 4$ processors. This matrix visualization gives a clear idea of the computational cost (by considering the fill-in) and the parallelizability (using the elimination tree). The specific permutations and the structure of the matrix for the general case will be discussed below.

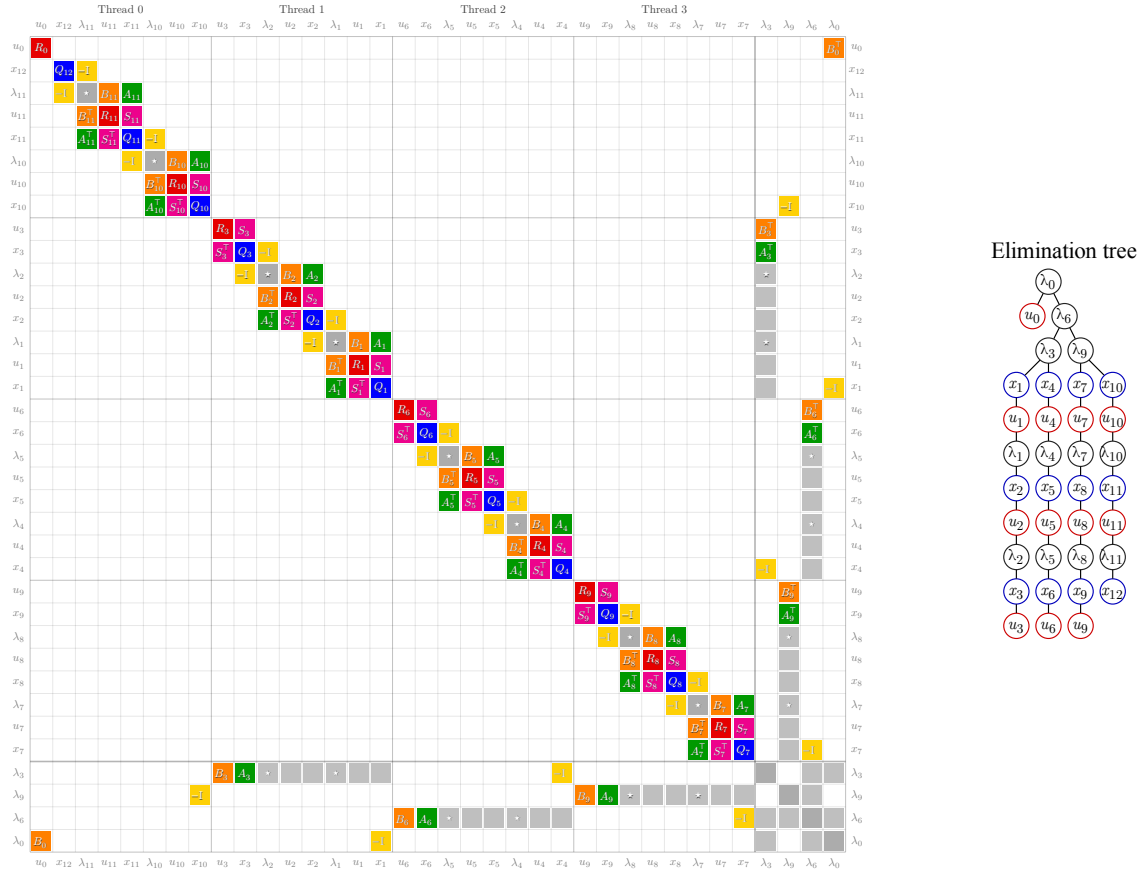

Figure 5. CYQLONE ordering of the coefficient matrix of a KKT system with optimal control structure (5) with horizon $N = 12$ for $P = 4$ processors. Grey blocks indicate fill-in incurred during the factorization of the matrix. Stars ($\star$) mark structured or redundant fill-in (see Section 4.2). The corresponding elimination tree visualizes the available parallelism during the factorization.

4.1 Matrix structure and high-level factorization procedure

We partition the matrix visualized in Figure 5 as follows:

$$\text{(Fig. 5)}: \quad \mathcal{K} = \left(\begin{array}{c||c} \mathcal{R} & \mathcal{A}^\top \\ \hline\hline \mathcal{A} & 0 \end{array}\right) = \left(\begin{array}{c|c|c|c||c} \mathcal{R}_0 & & & & \mathcal{A}_0^\top \mathrm{P}_0^\top \\ \hline & \mathcal{R}_1 & & & \mathcal{A}_1^\top \mathrm{P}_1^\top \\ \hline & & \mathcal{R}_2 & & \mathcal{A}_2^\top \mathrm{P}_2^\top \\ \hline & & & \mathcal{R}_3 & \mathcal{A}_3^\top \mathrm{P}_3^\top \\ \hline\hline \mathrm{P}_0\mathcal{A}_0 & \mathrm{P}_1\mathcal{A}_1 & \mathrm{P}_2\mathcal{A}_2 & \mathrm{P}_3\mathcal{A}_3 & 0 \\ \hline c{=}0 & c{=}1 & c{=}2 & c{=}3 & \end{array}\right). \tag{14}$$

The structure of this matrix is based on the partitioning of the full horizon of length N into P smaller intervals of length $n \triangleq N/P$. Each block column $c \in \{0, 1, \dots, P-1\}$ contains the states and controls of the stages in the interval $\{nc-n+1,\ nc-n+2,\ \dots,\ nc\}$ (in reverse order), as well as the Lagrange multipliers corresponding to the dynamics constraints in the interior of the interval. The purpose of this partitioning is to allow the different intervals to be solved in parallel on P processors.
Block column $c = 0$ is special, because it contains the controls u^0 of the first stage, combined with the variables of the last n stages. It is natural to handle this edge case uniformly by reducing the appropriate indices modulo N and closing the horizon in a circular or periodic fashion. One could imagine introducing matrices $S_0 = S_{12} = 0$ and $A_0 = A_{12} = 0$, ensuring that all block columns have the same structure.
The matrices $\mathcal{R}_c$ represent optimality conditions of smaller OCPs with horizon length n on the different intervals of the partitioning. The dynamics constraints describing the coupling between different intervals are found in the bottom row, described by matrices $\mathcal{A}_c$, and they serve as initial and terminal state constraints for the smaller OCPs on each interval. An odd–even permutation of these coupling constraints, represented by the matrices P_c, is used to enable cyclic reduction of the bottom-right block, as introduced in Section 3.2.2: the constraints corresponding to Lagrange multipliers λ^{ni} for odd i are ordered first, then the ones for which i is divisible by 2 but not 4, then i divisible by 4 but not 8, etc., with λ^0 last.

The CYQLONE algorithm for solving KKT systems with optimal control structure is derived by systematically performing block Cholesky factorization of different blocks of (14), leading to a factorization of the full matrix $\mathcal{K}$ that can be used to solve (5) by forward and back substitution. Specifically, we obtain a factorization of $\mathcal{K}$ by applying the straightforward block Cholesky procedure outlined in Section 3.1:

1. Compute block Cholesky factorizations of the diagonal blocks $\mathcal{R}_c = L_c^{\mathcal{R}} \mathcal{D}_c L_c^{\mathcal{R}\,\top}$ (in parallel).
2. Use these factorizations to solve $L_c^{\mathcal{A}} = \mathcal{A}_c L_c^{\mathcal{R}\,-\top} \mathcal{D}_c^{-1}$ (in parallel).
3. Compute the Schur complement $\mathcal{M} = -\mathcal{K}/\mathcal{R} = -\sum_{c=0}^{P-1} \mathrm{P}_c L_c^{\mathcal{A}} \mathcal{D}_c L_c^{\mathcal{A}\,\top} \mathrm{P}_c^\top$ (in parallel).
4. Compute a block Cholesky factorization of the Schur complement $\mathcal{M} = L^{\mathcal{M}} L^{\mathcal{M}\,\top}$ (using CR).

Each step makes use of the internal block structure of the matrices $\mathcal{R}_c$ and $\mathcal{A}_c$: For example, steps 1 and 2 can be combined into a single Riccati-like recursion that exploits the optimal control structure; many of the terms in step 3 cancel out; and the structure of symmetric or triangular blocks is exploited during the individual block matrix operations.

The full Cholesky factorization of $\mathcal{K}$ can then be written as

$$\mathcal{K} = \left(\begin{array}{c||c} L^{\mathcal{R}} & \\ \hline\hline L^{\mathcal{A}} & L^{\mathcal{M}} \end{array}\right) \left(\begin{array}{c||c} \mathcal{D} & \\ \hline\hline & -\mathrm{I} \end{array}\right) \left(\begin{array}{c||c} L^{\mathcal{R}} & \\ \hline\hline L^{\mathcal{A}} & L^{\mathcal{M}} \end{array}\right)^\top, \tag{15}$$

where the matrices $L^{\mathcal{R}}$, $L^{\mathcal{A}}$ and $\mathcal{D}$ consist of the blocks $L_c^{\mathcal{R}}$, $\mathrm{P}_c L_c^{\mathcal{A}}$ and $\mathcal{D}_c$, respectively.

4.2 Modified Riccati recursion (steps 1 and 2)

Consider one of the first four block columns of $\mathcal{K}$ in (14) in isolation, for example the second block column ($c = 1$), corresponding to variables $(u_3, x_3, \lambda_2, u_2, x_2, \lambda_1, u_1, x_1)$. Together with the corresponding initial and terminal constraint equations associated with Lagrange multipliers λ_0 and λ_3, this results in the following smaller block matrix:

$$\mathcal{K}_1 \triangleq \left(\begin{array}{c|c} \mathcal{R}_1 & \mathcal{A}_1^\top \\ \hline \mathcal{A}_1 & 0 \end{array}\right) = \left(\begin{array}{cccccccc|cc} \scriptstyle u^3 & \scriptstyle x^3 & \scriptstyle \lambda^2 & \scriptstyle u^2 & \scriptstyle x^2 & \scriptstyle \lambda^1 & \scriptstyle u^1 & \scriptstyle x^1 & \scriptstyle \lambda^3 & \scriptstyle \lambda^0 \\ R_3 & S_3 & & & & & & & B_3^\top & \\ S_3^\top & Q_3 & -\mathrm{I} & & & & & & A_3^\top & \\ & -\mathrm{I} & 0 & B_2 & A_2 & & & & & -\mathrm{I} \\ & & B_2^\top & R_2 & S_2 & & & & & \\ & & A_2^\top & S_2^\top & Q_2 & -\mathrm{I} & & & & \\ & & & & -\mathrm{I} & 0 & B_1 & A_1 & & \\ & & & & & B_1^\top & R_1 & S_1 & & \\ & & & & & A_1^\top & S_1^\top & Q_1 & & \\ \hline B_3 & A_3 & & & & & & & 0 & \\ & & & & & & & -\mathrm{I} & & 0 \end{array}\right).$$

Performing block Cholesky factorization of $\mathcal{R}_1$ using (6), we have

$$\mathcal{K}_1 = \left(\begin{array}{c} L_1^{\mathcal{R}} \\ \hline L_1^{\mathcal{A}} \end{array}\right) \mathcal{D}_1 \left(\begin{array}{c} L_1^{\mathcal{R}} \\ \hline L_1^{\mathcal{A}} \end{array}\right)^\top + \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & \mathcal{K}_1/\mathcal{R}_1 \end{array}\right), \tag{16}$$

$$\text{where} \quad L_1^{\mathcal{R}} = \begin{pmatrix} L_3^R & & & & & & & \\ L_3^S & L_3^Q & & & & & & \\ & -L_3^{Q\,-\top} & -L_3^{Q\,-\top} & & & & & \\ & & B_2^\top L_3^Q & L_2^R & & & & \\ & & A_2^\top L_3^Q & L_2^S & L_2^Q & & & \\ & & & & -L_2^{Q\,-\top} & -L_2^{Q\,-\top} & & \\ & & & & & B_1^\top L_2^Q & L_1^R & \\ & & & & & A_1^\top L_2^Q & L_1^S & L_1^Q \end{pmatrix}, \tag{17}$$

$$L_1^{\mathcal{A}} \triangleq \mathcal{A}_1 L_1^{\mathcal{R}\,-\top} \mathcal{D}_1^{-1} = \begin{pmatrix} L_3^B & A_3^{\mathrm{cl}} L_3^{Q\,-\top} & A_3^{\mathrm{cl}} L_3^{Q\,-\top} & L_2^B & A_2^{\mathrm{cl}} L_2^{Q\,-\top} & A_2^{\mathrm{cl}} L_2^{Q\,-\top} & L_1^B & L_1^A \\ & & & & & & & -L_1^{Q\,-\top} \end{pmatrix}, \tag{18}$$

$$\mathcal{D}_1 \triangleq \mathbf{blkdiag}\begin{pmatrix} \mathrm{I} & \mathrm{I} & -\mathrm{I} & \mathrm{I} & \mathrm{I} & -\mathrm{I} & \mathrm{I} & \mathrm{I} \end{pmatrix}. \tag{19}$$

The matrices L_j^R, L_j^S, and L_j^Q, which make up $L_1^{\mathcal{R}}$ in (17), are computed using the block Cholesky factorization procedure described in Section 3.1. Recall that there is some freedom in the choice of the factors of the diagonal blocks. Thanks to the special structure of $\mathcal{R}_1$, the following factorization can be used for the blocks corresponding to the states and the Lagrange multipliers:

$$\begin{pmatrix} P & -\mathrm{I} \\ -\mathrm{I} & 0 \end{pmatrix} = \begin{pmatrix} L^Q & \\ -L^{Q\,-\top} & -L^{Q\,-\top} \end{pmatrix} \begin{pmatrix} \mathrm{I} & \\ & -\mathrm{I} \end{pmatrix} \begin{pmatrix} L^{Q\,\top} & -L^{Q\,-1} \\ & -L^{Q\,-1} \end{pmatrix} \qquad \text{where} \quad P = L^Q L^{Q\,\top}. \tag{20}$$

Note how the bottom-right block $L^{Q\,-\top}$ does not need to be computed or stored: Storing L^Q is sufficient, because multiplication by $L^{Q\,-\top}$ (required during the solution of the linear system) can be implemented as back substitution using L^Q; its inverse does not need to be formed explicitly. The factorization in (20) is also responsible for the duplicate $A_j^{\mathrm{cl}} L_j^{Q\,-\top}$ blocks in $L_1^{\mathcal{A}}$. Since these redundant blocks do not have to be computed or stored explicitly, we denoted them using a star in Figure 5. The optimized block Cholesky factorization of $\mathcal{R}_1$ is described by Algorithm 1 (invoked with $c = 1$, $N = 12$ and $P = 4$ for the example considered here). By using (20), this procedure fully exploits the optimal control structure of $\mathcal{R}_1$, and it is similar [4] to the factorized Riccati recursion described in [12, Alg. 3]. It is also closely related to backward dynamic programming (see Section 9.2). Additionally, Algorithm 1 returns the blocks L_j^B, A_j^{cl} and L_1^A of $L_1^{\mathcal{A}}$. Merging the evaluation of $L_1^{\mathcal{R}}$ and $L_1^{\mathcal{A}}$ results in improved locality and cache performance. The products $A_j^{\mathrm{cl}} L_j^{Q\,-\top}$ are not carried out explicitly, except for the very first stage j_1 in each interval.

To obtain an alternative block Cholesky factorization of $\mathcal{R}_1$, one could use the trivial block Cholesky factorization of the entire 2×2 block of states and Lagrange multipliers, leading to an equivalent algorithm that is closer to the asymmetric implementation of the Riccati recursion from [12, Alg. 1]:

$$\begin{pmatrix} P & -\mathrm{I} \\ -\mathrm{I} & 0 \end{pmatrix} = \underbrace{\begin{pmatrix} \mathrm{I} & \\ & \mathrm{I} \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} P & -\mathrm{I} \\ -\mathrm{I} & 0 \end{pmatrix}}_{D} \underbrace{\begin{pmatrix} \mathrm{I} & \\ & \mathrm{I} \end{pmatrix}}_{L^\top} \qquad \text{with} \qquad \begin{pmatrix} P & -\mathrm{I} \\ -\mathrm{I} & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & -\mathrm{I} \\ -\mathrm{I} & -P \end{pmatrix}. \tag{21}$$

4.3 Computation of the Schur complement (step 3)

Once $L_1^{\mathcal{A}}$ has been computed by Algorithm 1, evaluating the contribution of each block column to the Schur complement is done using a straightforward structured symmetric matrix product:

$$-(\mathcal{K}_1/\mathcal{R}_1) = L_1^{\mathcal{A}} \mathcal{D}_1 L_1^{\mathcal{A}\,\top} = \begin{pmatrix} L_3^B L_3^{B\,\top} + L_2^B L_2^{B\,\top} + L_1^B L_1^{B\,\top} + L_1^A L_1^{A\,\top} & -L_1^A L_1^{Q\,-1} \\ -L_1^{Q\,-\top} L_1^{A\,\top} & L_1^{Q\,-\top} L_1^{Q\,-1} \end{pmatrix} \triangleq \begin{pmatrix} \hat{M}_1 & \hat{K}_1^\top \\ \hat{K}_1 & \hat{M}_0 \end{pmatrix}. \tag{22}$$

[4] In the case where $P = 1$, we have $A_0 = A_N = 0 = S_N = S_0$, and lines 9 and 11 in Algorithm 1 can be skipped since A_j^{cl} and L_j^S are zero as well. The resulting algorithm corresponds to [12, Alg. 3] almost exactly.

Algorithm 1: Factorization of a single modified Riccati block column

Input: A, B, Q, R, S and N: OCP data matrices and horizon length

Input: P: number of intervals to partition the horizon into

Input: $c \in \mathbb{N}_{[0,P)}$: index of the block column to factor (determines the interval)

Output: $L_c^{\mathcal{R}}$ (L^R, L^S, L^Q) and $L_c^{\mathcal{A}}$ ($L^B, L^A, A^{\mathrm{cl}}$): blocks of the Cholesky factor of $\mathcal{K}_c$

Function FACTOR–BLOCK–COLUMN–RICCATI(c)

$n = N/P$ ▷ Number of stages per interval

$j_1 = n(c-1)+1, \quad j_n = nc$ ▷ Indices of the first and last stages in the selected block column

$\begin{pmatrix} \hat{B}_{j_n} & \hat{A}_{j_n} \end{pmatrix} = \begin{pmatrix} B_{j_n} & A_{j_n} \end{pmatrix}$

$\begin{pmatrix} \hat{R}_{j_n} & \hat{S}_{j_n} \\ \hat{S}_{j_n}^\top & \hat{Q}_{j_n} \end{pmatrix} = \begin{pmatrix} R_{j_n} & S_{j_n} \\ S_{j_n}^\top & Q_{j_n} \end{pmatrix}$

for $j = j_n, j_n - 1, j_n - 2, \ldots, j_1$

$\begin{pmatrix} L_j^R & \\ L_j^S & L_j^Q \end{pmatrix} = \mathrm{chol} \begin{pmatrix} \hat{R}_j & \hat{S}_j \\ \hat{S}_j^\top & \hat{Q}_j \end{pmatrix}$ ▷ Cost-to-go $P_j = L_j^Q L_j^{Q\top} = \hat{Q}_j - \hat{S}_j^\top \hat{R}_j^{-1} \hat{S}_j$ ▷ (potrf) $\frac{1}{6} n_{ux}^3$

$L_j^B = \hat{B}_j L_j^{R\,-\top}$ ▷ Gain matrix $K_j = L_j^{R\,-\top} L_j^{S\,\top} = \hat{R}_j^{-1} \hat{S}_j$ ▷ (trsm) $\frac{1}{2} n_x n_u^2$

$A_j^{\mathrm{cl}} = \hat{A}_j - L_j^B L_j^{S\,\top}$ ▷ Stabilized dynamics $A_j^{\mathrm{cl}} = \hat{A}_j - \hat{B}_j K_j$ ▷ (gemm) $n_x^2 n_u$

if $j > j_1$ ▷ Propagate dynamics and cost-to-go to the previous stage

$\begin{pmatrix} \hat{B}_{j-1} & \hat{A}_{j-1} \end{pmatrix} = A_j^{\mathrm{cl}} \begin{pmatrix} B_{j-1} & A_{j-1} \end{pmatrix}$ ▷ (gemm) $n_{ux} n_x^2$

$V_{j-1} = \begin{pmatrix} B_{j-1}^\top \\ A_{j-1}^\top \end{pmatrix} L_j^Q$ ▷ Riccati recursion $\begin{aligned} \hat{R}_{j-1} &= R_{j-1} + B_{j-1}^\top P_j B_{j-1} \\ \hat{S}_{j-1} &= S_{j-1} + B_{j-1}^\top P_j A_{j-1} \\ \hat{Q}_{j-1} &= Q_{j-1} + A_{j-1}^\top P_j A_{j-1} \end{aligned}$ ▷ (trmm) $\frac{1}{2} n_{ux} n_x^2$

$\begin{pmatrix} \hat{R}_{j-1} & \hat{S}_{j-1} \\ \hat{S}_{j-1}^\top & \hat{Q}_{j-1} \end{pmatrix} = \begin{pmatrix} R_{j-1} & S_{j-1} \\ S_{j-1}^\top & Q_{j-1} \end{pmatrix} + V_{j-1} V_{j-1}^\top$ ▷ (syrk) $\frac{1}{2} n_{ux}^2 n_x$

$L_{j_1}^A = A_{j_1}^{\mathrm{cl}} L_{j_1}^{Q\,-\top}$ ▷ Only the first stage needs L^A ▷ (trsm) $\frac{1}{2} n_x^3$

To avoid unintelligible index manipulations in the pseudocode, we use the convention that stage indices are reduced modulo N, e.g. R_j represents $R_{j\,\%N}$. The matrices Q_j and L_j^Q are offset by one because of the elimination of x^0, hence Q_j represents $Q_{((j-1)\%N)+1}$. Furthermore, $A_0 = 0$ and $S_0 = 0$.

Notice how the blocks involving $A_j^{\mathrm{cl}} L_j^{Q\,-\top}$ from (18) cancel out for all but the first stage. Arrows are used to indicate the direction of the coupling to other block columns. For example, $\vec{M}_1$ is the contribution to the Schur complement from the constraints representing the coupling forward in time, from the second to the third block columns ($c = 1$ to $c = 2$), and $\overleftarrow{M}_0$ is the contribution of the coupling back in time, from the second to the first block column ($c = 1$ to $c = 0$). The updated forward coupling matrices $\vec{M}_i$ are derived from the matrices A_j and B_j in the eliminated dynamics constraints between intervals, whereas the backward coupling matrices $\overleftarrow{M}_i$ are derived from the corresponding matrix $E_{j+1} = \mathrm{I}$. Because of the odd–even structure of CR, the order of the block rows in $\mathcal{A}_c$ is reversed for even c. For example, for the third block column ($c = 2$):

$$-(\mathcal{K}_2/\mathcal{R}_2) = \begin{pmatrix} L_4^{Q\,-\top} L_4^{Q\,-1} & -L_4^{Q\,-\top} L_4^{A\,\top} \\ -L_4^A L_4^{Q\,-1} & L_6^B L_6^{B\,\top} + L_5^B L_5^{B\,\top} + L_4^B L_4^{B\,\top} + L_4^A L_4^{A\,\top} \end{pmatrix} \triangleq \begin{pmatrix} \overleftarrow{M}_1 & \vec{K}_1^{\top} \\ \overleftarrow{K}_1 & \vec{M}_2 \end{pmatrix}. \tag{23}$$

The indices of the subdiagonal blocks $\overleftarrow{K}_i$ and $\vec{K}_i$ are chosen to match the ones of the diagonal blocks in the same column, as this simplifies the notation in the upcoming CR step. For completeness, we also list the Schur complement matrices for the first and last block columns. The matrix $\vec{K}_3$ is zero because the original OCP has no terminal constraints ($A_{12} = 0$). Blocks that are zero are shown in gray and have been included to demonstrate the regular structure of the factorization. A straightforward generalization to problems with coupling between the first and last stage is possible, but is not considered here for the sake of brevity.

$$-(\mathcal{K}_0/\mathcal{R}_0) = \begin{pmatrix} L_{10}^{Q\,-\top} L_{10}^{Q\,-1} & -L_{10}^{Q\,-\top} L_{10}^{A} \\ -L_{10}^{A\,\top} L_{10}^{Q\,-1} & L_0^B L_0^{B\,\top} + L_{11}^B L_{11}^{B\,\top} + L_{10}^B L_{10}^{B\,\top} + L_{10}^A L_{10}^{A\,\top} \end{pmatrix} \triangleq \begin{pmatrix} \overleftarrow{M}_3 & \vec{K}_3^{\top} \\ \overleftarrow{K}_3 & \vec{M}_0 \end{pmatrix}. \tag{24}$$

$$-(\mathcal{K}_3/\mathcal{R}_3) = \begin{pmatrix} L_9^B L_9^{B\,\top} + L_8^B L_8^{B\,\top} + L_7^B L_7^{B\,\top} + L_7^A L_7^{A\,\top} & -L_7^A L_7^{Q\,-1} \\ -L_7^{Q\,-\top} L_7^{A\,\top} & L_7^{Q\,-\top} L_7^{Q\,-1} \end{pmatrix} \triangleq \begin{pmatrix} \overleftarrow{M}_3 & \vec{K}_3^{\top} \\ \overleftarrow{K}_3 & \vec{M}_2 \end{pmatrix}. \tag{25}$$

Combining the contributions from all four block columns in (22)–(25), the full Schur complement reads:

$$\mathcal{M} = -\mathcal{K}/\mathcal{R} = \begin{array}{c} \\ \lambda^3 \\ \lambda^9 \\ \lambda^6 \\ \lambda^0 \end{array} \begin{array}{c} \begin{array}{cccc} \lambda^3 & \lambda^9 & \lambda^6 & \lambda^0 \\ i=1 & i=3 & i=2 & i=0 \end{array} \\ \begin{pmatrix} \overleftarrow{M}_1 + \vec{M}_1 & & \vec{K}_1^{\top} & \vec{K}_1^{\top} \\ & \overleftarrow{M}_3 + \vec{M}_3 & \vec{K}_3^{\top} & \vec{K}_3^{\top} \\ \vec{K}_1 & \vec{K}_3 & \overleftarrow{M}_2 + \vec{M}_2 & \\ \vec{K}_1 & \vec{K}_3 & & \overleftarrow{M}_0 + \vec{M}_0 \end{pmatrix} \end{array}. \tag{26}$$

In the following sections and in the pseudocode, we use indices j to refer to the original OCP stages, and indices i to refer to the block columns of the (permuted) Schur complement (26). Column i corresponds to the Lagrange multipliers λ^{ni}. The rows and columns of the Schur complement are permuted to enable cyclic reduction: they are ordered by increasing 2-adic valuation $\nu_2(i)$ (see Section 3.2.2). Consequently, the off-diagonal blocks $\overleftarrow{K}_i$ (representing coupling back in time) are found in column i and row $i - 2^{\nu_2(i)}$; and blocks $\vec{K}_i$ (representing coupling forward in time) are found in column i and row $i + 2^{\nu_2(i)}$.

The construction of $\mathcal{M}$ can be fully parallelized, assuming that the final additions of the two terms in the diagonal blocks are synchronized correctly. A possible parallel procedure for constructing $\mathcal{M}$ is given in the COMPUTE–SCHUR function of Algorithm 2. Since the diagonal blocks will be updated during the CR algorithm in the following step, we use an explicit superscript to indicate the level l in the CR recursion (as in Algorithm 5), with initial values $M_i^{(0)} = \overleftarrow{M}_i + \vec{M}_i$.

4.4 Factorization of the Schur complement (step 4)

Thanks to the odd–even ordering of the Lagrange multipliers in Figure 5, cyclic reduction of the Schur complement $\mathcal{M}$ corresponds directly to its block Cholesky factorization. The resulting block Cholesky factor $L^{\mathcal{M}}$ is a sparse matrix with at most two subdiagonal blocks per block column, with a particular structure as visualized in Figure 4. For the specific case of Figure 5 with only $P = 4$ processors, we have:

$$L^{\mathcal{M}} = \begin{array}{c} \\ \lambda^3 \\ \lambda^9 \\ \lambda^6 \\ \lambda^0 \end{array} \begin{array}{c} \begin{array}{cccc} \lambda^3 & \lambda^9 & \lambda^6 & \lambda^0 \end{array} \\ \begin{pmatrix} L_1 & & & \\ & L_3 & & \\ Y_1 & U_3 & L_2 & \\ U_1 & Y_3 & U_2 & L_0 \end{pmatrix} \end{array}. \tag{27}$$

A procedure for the factorization of $\mathcal{M}$ is given in Algorithm 2, which is a specialization of Algorithm 5. The main difficulty here is keeping track of the appropriate indices, so it may be advisable to study the

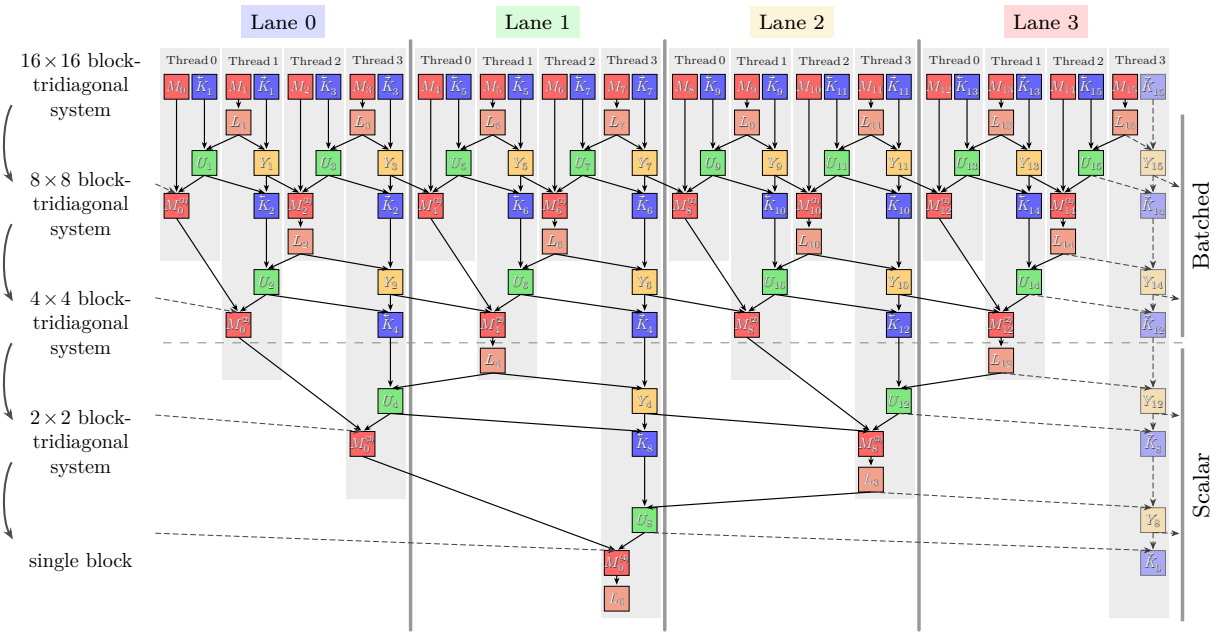

Figure 6. Graph representation of the different steps required for the cyclic reduction of a symmetric 16×16 block-tridiagonal matrix. The 16 block columns of the original matrix are distributed across 4 different threads and 4 vector lanes. Each square node represents one of the blocks of the block-tridiagonal matrix or its Cholesky factor; nodes including a superscript are blocks of the intermediate block-tridiagonal systems of reduced size from (11). Node colors match the colors in Algorithm 2. Edges represent data dependencies between steps of the factorization. For instance, following Section 3.2.1, $U_1 \triangleq \vec{K}_0 L_1^{-\top}$, so there are edges from $\vec{K}_0$ and L_1 to U_1. The vertical axis represents the wall time. The horizontal position of each block indicates the thread and vector lane where it is computed. Notice the recursive, self-similar structure of the method: the operations performed at each level are the same, just spread out horizontally and executed by half as many threads than the level before it. Dashed arrows complete the structure, and are required when solving a system with circular coupling $\vec{K}_{15} \neq 0$ or when extending the figure to larger matrix sizes. Vector lane assignment, the distinction between scalar and batched levels, and other vectorization aspects will be discussed in Section 7.

indices in Figure 4 to develop an intuition before deciphering the pseudocode. To clarify the structure of the algorithm, we note that at level l in the CR recursion, the columns i for which $\nu_2(i) = l$ are eliminated. For these columns, the diagonal blocks $M_i^{(l)} = L_i L_i^\top$ are factorized first, and then the two subdiagonal blocks $U_i = \overleftarrow{K}_i L_i^{-\top}$ and $Y_i = \vec{K}_i L_i^{-\top}$ are computed. Finally, the Schur complement of the eliminated columns is computed by subtracting the appropriate products of U_i and Y_i from the bottom-right corner: The symmetric products $U_i U_i^\top$ and $Y_i Y_i^\top$ are subtracted from the diagonal blocks $M_{i-2^l}^{(l)}$ and $M_{i+2^l}^{(l)}$ respectively, labeling the updated matrices $M_{i-2^l}^{(l+1)}$ and $M_{i+2^l}^{(l+1)}$; and the asymmetric products $U_i Y_i^\top$ and $Y_i U_i^\top$ result in off-diagonal fill-in, labeled $\overleftarrow{K}_{i+2^l}$ and $\vec{K}_{i-2^l}$. Since the matrix is symmetric, only the diagonal and subdiagonal blocks are computed. The stride between the indices of successive rows/columns of $\mathcal{M}$ and $L^{\mathcal{M}}$ at level l is 2^l, so index offsets of $\pm 2^l$ can be interpreted as the next/previous row/column of the reduced system.

A visual representation of the steps of the cyclic reduction algorithm is shown in Figure 6 (with colors of the different matrices matching the colors in the margin of Algorithm 2). Further details about the thread and vector lane scheduling will be discussed in Section 7.

4.5 Operation counts

Algorithms 1 and 2 list the names of the different BLAS and LAPACK routines that can be used to implement the different steps, together with the cubic terms of the FLOP count[5] for each routine (with $n_{ux} = n_x + n_u$). The nonstandard routine `trsyrk` computes the lower triangular part of the symmetric product of an upper triangular matrix and its transpose.[6] Ignoring lower-order terms, the total number of floating-point

[5] A fused multiply–add (FMA) is counted as a single operation, reflecting the properties of modern hardware where FMA instructions have the same latency and throughput as a standalone multiplication. Using this convention, $n \times n$ matrix–matrix multiplication requires n^3 FLOPs, and Cholesky factorization of an $n \times n$ matrix requires $\frac{1}{6}n^3$ operations.

[6] Similar to the LAPACK `lauum` routine, but with transposed arguments.

Algorithm 2: CYQLONE factorization

Input: A, B, Q, R, S and N: OCP data matrices and horizon length
Input: P: number of intervals to partition the horizon into
Output: $L^{\mathcal{R}}$ (L^R, L^S, L^Q), $L^{\mathcal{A}}$ $(L^B, L^A, A^{\mathrm{cl}})$ and $L^{\mathcal{M}}$ (L, Y, U): blocks of the Cholesky factor of $\mathcal{K}$

for $c = 0, ..., P-1$ (in parallel)
  FACTOR–BLOCK–COLUMN–RICCATI(c) ▷ (steps 1 and 2)
  COMPUTE–SCHUR(c) ▷ (step 3)
  FACTOR–SCHUR(c) ▷ (step 4)

Function COMPUTE–SCHUR(c) ▷ Computes part of the Schur complement $\mathcal{M}$
  $n = N/P$ ▷ Number of stages per interval
  $j_1 = n(c-1)+1, \quad j_n = nc$ ▷ Indices of the first and last stages in the selected block column
  $i_{\rightarrow} = c, \quad i_{\leftarrow} = c-1$ ▷ Row indices of forward (resp. backward) coupling equations in $\mathrm{P}_c\mathcal{A}_c$
  $T_c = L^{Q\,-\top}_{j_1}$ ▷ (trtri) $\frac{1}{6}n_x^3$
  if $\nu_2^P(i_{\leftarrow}) > \nu_2^P(i_{\rightarrow})$: $\overleftarrow{K}_{i_{\rightarrow}} = -T_c L^{A\,\top}_{j_1}$ **else**: $\overrightarrow{K}_{i_{\leftarrow}} = -L^A_{j_1} T_c^{\top}$ ▷ (trmm) $\frac{1}{2}n_x^3$
  $\overleftarrow{M}_{i_{\leftarrow}} = T_c T_c^{\top}$ ▷ (trsyrk) $\frac{1}{6}n_x^3$
  — sync — ▷ Wait for $\overleftarrow{M}_{i_{\rightarrow}}$
  $M^{(0)}_{i_{\rightarrow}} = \overleftarrow{M}_{i_{\rightarrow}} + \overrightarrow{M}_{i_{\rightarrow}} = \overleftarrow{M}_{i_{\rightarrow}} + \begin{pmatrix} L^B_{j_n} & \cdots & L^B_{j_1} & L^A_{j_1} \end{pmatrix} \begin{pmatrix} L^B_{j_n} & \cdots & L^B_{j_1} & L^A_{j_1} \end{pmatrix}^{\top}$ ▷ (syrk) $\frac{n}{2}n_u n_x^2 + \frac{1}{2}n_x^3$

Function FACTOR–SCHUR(c) ▷ Uses cyclic reduction to factorize the Schur complement $\mathcal{M}$
  if $\nu_2^P(c) = 0$: $L_c = \mathrm{chol}\left(M_c^{(0)}\right)$ ▷ (potrf) $\frac{1}{6}n_x^3$
  for $l = 0, ..., \log_2(P) - 1$ ▷ Recursion level of cyclic reduction
    $i_U = c+1, \quad i_Y = c+1-2^l$ ▷ Assign the evaluation of blocks U and Y to different threads
    — sync — ▷ Wait for L
    if $\nu_2^P(i_U) = l$: $U_{i_U} = \overleftarrow{K}_{i_U} L^{-\top}_{i_U}$ ▷ (trsm) $\frac{1}{2}n_x^3$
    elif $\nu_2^P(i_Y) = l$: $Y_{i_Y} = \overrightarrow{K}_{i_Y} L^{-\top}_{i_Y}$ ▷ (trsm) $\frac{1}{2}n_x^3$
    — sync — ▷ Wait for U, Y
    if $\nu_2^P(i_U) = l$: FACTOR–L(l, i_Y)
    elif $\nu_2^P(i_Y) = l$: UPDATE–K(l, i_Y)

Function FACTOR–L(l, i) ▷ Update diagonal block i of $\mathcal{M}$ after elimination of level l
  $i_U = i + 2^l, \quad i_Y = i - 2^l$ ▷ Column indices of U_{i_U} and Y_{i_Y} in row i and level l
  $M_i^{(l+1)} = M_i^{(l)} - U_{i_U} U_{i_U}^{\top} - Y_{i_Y} Y_{i_Y}^{\top}$ ▷ (2×syrk) n_x^3
  if $\nu_2^P(i) = l+1$: $L_i = \mathrm{chol}\left(M_i^{(l+1)}\right)$ ▷ Factorization for the next level ▷ (potrf) $\frac{1}{6}n_x^3$

Function UPDATE–K(l, i) ▷ Compute fill-in caused by elimination of column i of $\mathcal{M}$
  $i_{\leftarrow} = i - 2^l, \quad i_{\rightarrow} = i + 2^l$ ▷ Row indices of U_i and Y_i in column i
  if $\nu_2^P(i_{\leftarrow}) > \nu_2^P(i_{\rightarrow})$: $\overleftarrow{K}_{i_{\rightarrow}} = -U_i Y_i^{\top}$ **else**: $\overrightarrow{K}_{i_{\leftarrow}} = -Y_i U_i^{\top}$ ▷ (gemm) n_x^3

We again use implicit reduction of the indices i modulo P to simplify the notation, and we define $\nu_2^P(0) \triangleq \nu_2(P)$ and $\nu_2^P(i) \triangleq \nu_2(i)$ for $0 < i < P$. Color coding matches Figures 4 and 6.

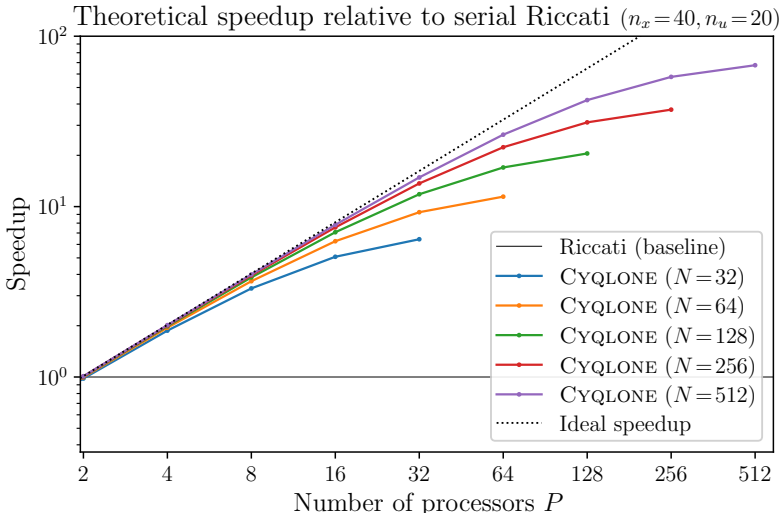

Figure 7. Speedup of the parallel CYQLONE method compared to the serial Riccati recursion for various horizon lengths, considering the number of floating-point operations in the critical path.

operations in the critical path of Algorithm 2 is given by

$$\tfrac{N}{P}\left(\tfrac{1}{6}n_{ux}^3 + \tfrac{1}{2}n_x n_u^2 + n_x^2 n_u\right) + \left(\tfrac{N}{P} - 1\right)\left(\tfrac{3}{2}n_{ux}n_x^2 + \tfrac{1}{2}n_{ux}^2 n_x\right) + \tfrac{1}{2}n_x^3 \tag{28a}$$

$$+ \tfrac{4}{3}n_x^3 + \tfrac{N}{P}\tfrac{1}{2}\, n_u n_x^2 \tag{28b}$$

$$+ \tfrac{1}{6}n_x^3 + \log_2(P)\, \tfrac{5}{3}n_x^3, \tag{28c}$$

where (28a) is the cost of the modified Riccati recursion from Algorithm 1, (28b) is the cost of computing the Schur complement $\mathcal{M}$ in the COMPUTE–SCHUR function of Algorithm 2, and (28c) is the cost of the cyclic reduction of $\mathcal{M}$ in the FACTOR–SCHUR function of Algorithm 2.

The Riccati-like elimination of the different intervals in steps 1 and 2, and the evaluation of the Schur complement in step 3 are embarrassingly parallel: Each interval can be processed fully independently, without communication (which can be seen from the parallel branches at the bottom of the elimination tree of Figure 5). For (28a)–(28b), the number of operations in the critical path scales linearly with $\mathcal{O}(N/P)$. The cyclic reduction in step 4 requires $\log_2(P) + 1$ steps, with communication in each step (each node near the root of the elimination tree receives updated matrices from its two children). If the number of stages is significantly larger than the number of processors ($N \gg P$), the run time of the first three steps dominates and the expected overall speedup is close to linear. If the number of stages is closer to the number of processors, ($N \approx P$), the fourth step dominates the overall run time (scaling logarithmically with P), resulting in diminishing returns when doubling P from $N/2$ to N, for example.

Figure 7 shows the theoretical limits of the speedup of CYQLONE over a serial implementation based on the Riccati recursion [12, Alg. 3], which requires $\frac{1}{6}n_x^3 + N\left(\frac{1}{2}n_{ux}n_x^2 + \frac{1}{2}n_{ux}^2 n_x + \frac{1}{6}n_{ux}^3\right) + \mathcal{O}(Nn_{ux}^2)$ operations. The speedup is computed by considering the number of floating-point operations in the critical path of both methods (which is what determines the wall-clock time). The partitioning into parallel sub-intervals in CYQLONE results in some additional fill-in compared to the serial Riccati recursion, which explains why CYQLONE does not see any speedup in the case with two threads. When the number of processors P is greater than two, the parallelism gained from the partitioning already makes up for this extra work, and doubling the number of processors approximately halves the run time for sufficiently large N. For a single processor, no partitioning of the horizon is performed, and CYQLONE reduces to an algorithm that is almost equivalent to the factorized Riccati recursion from [12, Alg. 3], as discussed in the footnote in Section 4.2. The only difference is the early elimination of u^0, which has a negligible impact on the run time for large N.

4.6 Remainders and padding

Since CYQLONE relies on cyclic reduction, the number of processors P must be a power of two. Performance is optimal if P divides the horizon length N, but this is not a strict requirement: CYQLONE can be used for any N by rounding it up to the next multiple of P, and introducing padding blocks $Q_{j+1} = \mathrm{I}$, $R_j = \mathrm{I}$, $S_{j+1} = 0$ and $E_{j+1} = \mathrm{I}$, $A_j = 0$, $B_j = 0$ for $N \leq j < P\lceil N/P \rceil$.

5 Solving problems with inequality constraints

To handle optimal control problems with inequality constraints of the form (1), we make use of QPALM, a proximal augmented Lagrangian method for quadratic programs with inequality constraints [19]. This section briefly summarizes the structure and properties of the inner optimization problems that need to be solved iteratively by QPALM's semismooth Newton solver, which is responsible for the main computational cost of the method. When preserving the states and the dynamics constraints in the inner problems as in the QPALM-OCP solver described in [26, Sec. III], the resulting Newton systems are KKT systems with optimal control structure (5), which we solve using the CYQLONE method described in Section 4. Other operations required in the QPALM algorithm, such as matrix–vector products, dot products, vector addition, etc., can be parallelized as well. Furthermore, we discuss how QPALM's exact line search procedure can be optimized and parallelized. We will refer to the parallel QPALM-based solver for linear–quadratic optimal control problems of the form (1) that uses CYQLONE as its linear solver as CYQPALM.

5.1 The augmented Lagrangian inner problem

QPALM and QPALM-OCP solve problem (1) by relaxing the inequality constraints $b_l^j \le C_j x^j + D_j u^j \le b_u^j$ and $b_l^N \le C_N x^N \le b_u^N$ using an augmented Lagrangian method (ALM). The inner problem to be solved at each outer iteration of this method amounts to the minimization of a piecewise quadratic augmented Lagrangian function, subject to equality constraints:

$$\begin{aligned} \underset{\mathbf{u},\mathbf{x}}{\textbf{minimize}} \quad & \sum_{j=0}^{N-1} \ell_j^\Sigma(x^j, u^j) + \ell_N^\Sigma(x^N) \\ \textbf{subject to} \quad & E_0 x^0 = x_{\text{init}} \\ & E_{j+1} x^{j+1} = A_j x^j + B_j u^j + f^j \quad {\scriptstyle (0 \le j < N)} \end{aligned} \tag{29}$$

where the strongly convex stage-wise cost functions are given by the Γ-proximal Σ_j-augmented Lagrangians

$$\begin{aligned} \ell_j^\Sigma(x,u) &= \ell_j(x,u) + \tfrac{1}{2}\,\mathbf{dist}^2_{\Sigma_j}\big(C_j x + D_j u + \Sigma_j^{-1} y^j;\, [b_l^j, b_u^j]\big) + \tfrac{1}{2}\|x - \bar{x}^j\|^2_{\Gamma_x^{-1}} + \tfrac{1}{2}\|u - \bar{u}^j\|^2_{\Gamma_u^{-1}} \\ \text{and} \quad \ell_N^\Sigma(x) &= \ell_N(x) + \tfrac{1}{2}\,\mathbf{dist}^2_{\Sigma_N}\big(C_N x + \Sigma_N^{-1} y^N;\, [b_l^N, b_u^N]\big) + \tfrac{1}{2}\|x - \bar{x}^N\|^2_{\Gamma_x^{-1}}. \end{aligned} \tag{30}$$

In these expressions, the vectors $y^j \in \mathbb{R}^{n_y}$ represent the current estimate of the Lagrange multipliers corresponding to the inequality constraints in stage j, and $\Sigma_j \in \mathrm{Sym}_{++}(\mathbb{R}^{n_y})$ are diagonal matrices containing the ALM penalty weights. The matrices $\Gamma_x \in \mathrm{Sym}_{++}(\mathbb{R}^{n_x})$ and $\Gamma_u \in \mathrm{Sym}_{++}(\mathbb{R}^{n_u})$ add primal regularization through a proximal term, relative to the fixed vectors $\bar{x}^j \in \mathbb{R}^{n_x}$ and $\bar{u}^j \in \mathbb{R}^{n_u}$. The function $\mathbf{dist}^2_\Sigma(w;\Omega)$ denotes the squared distance in Σ-norm between a point w and a set Ω. The intervals $[b_l^j, b_u^j]$ should be interpreted component-wise, describing n_y-dimensional boxes. Because of the squared distance, the augmented Lagrangians are not twice continuously differentiable. Therefore, QPALM–OCP employs a semismooth Newton method for solving (29), using the generalized Hessian matrices $H_j(u,x)$ [26, Eq. 15]:

$$\begin{aligned} H_j(u,x) &\triangleq \begin{pmatrix} R_j(u,x) & S_j(u,x) \\ S_j(u,x)^\top & Q_j(u,x) \end{pmatrix} \\ &= \begin{pmatrix} R_j^\ell & S_j^\ell \\ S_j^{\ell\top} & Q_j^\ell \end{pmatrix} + \begin{pmatrix} D_j^\top \\ C_j^\top \end{pmatrix} \Sigma_j^{\mathcal{J}}(u,x) \begin{pmatrix} D_j & C_j \end{pmatrix} + \begin{pmatrix} \Gamma_u^{-1} & \\ & \Gamma_x^{-1} \end{pmatrix} \in \partial_C\big(\nabla \ell_j^\Sigma(u,x)\big), \end{aligned} \tag{31}$$

where we defined $\Sigma_j^{\mathcal{J}}(u,x)$ as Σ_j with the entries corresponding to inactive constraints set to zero:

$$\Big(\Sigma_j^{\mathcal{J}}(u,x)\Big)_{ii} \triangleq \begin{cases} \big(\Sigma_j\big)_{ii} & \text{if } i \in \mathcal{J}_j(u,x) \triangleq \Big\{ i \in \mathbb{N}_{[1,m]} \;\Big|\; \big(C_j x + D_j u + \Sigma_j^{-1} y^j\big)_i \notin \big[(b_l^j)_i, (b_u^j)_i\big] \Big\}, \\ 0 & \text{otherwise.} \end{cases} \tag{32}$$

The quantities $H_N(x)$, $\Sigma_N^{\mathcal{J}}(x)$ and $\mathcal{J}_N(x)$ are defined analogously.

5.2 Structured Newton systems

The Newton step $(d_{\mathbf{x}}, d_{\mathbf{u}})$ with $d_{\mathbf{x}} \triangleq [\Delta x^0 \cdots \Delta x^N] \in \mathbb{R}^{n_x \times (N+1)}$, $d_{\mathbf{u}} \triangleq [\Delta u^0 \cdots \Delta u^{N-1}] \in \mathbb{R}^{n_u \times N}$ and the Lagrange multipliers corresponding to the equality constraints $\boldsymbol{\lambda} \triangleq [\lambda^{-1} \cdots \lambda^{N-1}]$[7] $\in \mathbb{R}^{n_x \times (N+1)}$ can be

[7] The multiplier λ^{-1} corresponds to the initial state constraint $E_0 x^0 = x_{\text{init}}$, which will be eliminated later. The remaining N multiplier vectors are then numbered starting from 0.

found by solving a Newton system of the linearized optimality conditions of (29) at a point $(\mathbf{u}, \mathbf{x})$:

$$\begin{cases} Q_N(x^N)\,\Delta x^N - E_N\lambda^{N-1} & = -q^N(x^N) \\ R_j(u^j,x^j)\,\Delta u^j + S_j(u^j,x^j)\,\Delta x^j + B_j^\top\lambda^j & = -r^j(u^j,x^j) \qquad (0\le j<N) \\ S_j(u^j,x^j)^\top\Delta u^j + Q_j(u^j,x^j)\,\Delta x^j + A_j^\top\lambda^j - E_j\lambda^{j-1} & = -q^j(u^j,x^j) \qquad (0\le j<N) \\ B_j\Delta u^j + A_j\Delta x^j - E_{j+1}\Delta x^{j+1} & = -c^j(u^j,x^j,x^{j+1}) \qquad (0\le j<N) \\ -E_0\Delta x^0 & = -c^{-1}(x^0). \end{cases} \tag{33}$$

The right-hand side is given by the negative gradients of the augmented Lagrangians and the negative residuals of the equality constraints:

$$\begin{aligned} q^N(x^N) \triangleq \nabla\ell_N^\Sigma(x^N) &= Q_N^\ell x^N + q_\ell^N + C_N^\top\,\hat y^N(x^N) + \Gamma_x^{-1}(x^N - \bar x^N) \\ &\text{with } \hat y^N(x^N) \triangleq y^N + \Sigma_N\Big(C_N x^N - \mathbf{\Pi}\big(C_N x^N + \Sigma_N^{-1}y^N;\,[b_l^N, b_u^N]\big)\Big) \\ \begin{pmatrix} r^j(u^j,x^j) \\ q^j(u^j,x^j)\end{pmatrix} \triangleq \nabla\ell_j^\Sigma(u^j,x^j) &= \begin{pmatrix} R_j^\ell & S_j^\ell \\ S_j^{\ell\top} & Q_j^\ell\end{pmatrix}\begin{pmatrix} u^j \\ x^j\end{pmatrix} + \begin{pmatrix} r_\ell^j \\ q_\ell^j\end{pmatrix} + \begin{pmatrix} D_j^\top \\ C_j^\top\end{pmatrix}\hat y^j(u^j,x^j) + \begin{pmatrix} \Gamma_u^{-1}(u^j-\bar u^j) \\ \Gamma_x^{-1}(x^j-\bar x^j)\end{pmatrix} \\ &\text{with } \hat y^j(u^j,x^j) \triangleq y^j + \Sigma_j\Big(D_j u^j + C_j x^j - \mathbf{\Pi}\big(D_j u^j + C_j x^j + \Sigma_j^{-1}y^j;\,[b_l^j, b_u^j]\big)\Big) \\ c^j(u^j,x^j,x^{j+1}) &\triangleq B_j u^j + A_j x^j + f^j - E_{j+1}x^{j+1} \\ c^{-1}(x^0) &\triangleq x_{\text{init}} - E_0 x^0. \end{aligned} \tag{34}$$

Computation of the quantities $\hat y^j$, r^j, q^j can be done independently and in parallel for all stages j. The equality constraint residuals c^j can also be computed in parallel, but require communication between stages.

The initial state x^0 and the corresponding steps Δx^0 and λ^{-1} are often eliminated for practical reasons. The stationarity conditions and equality constraints for the first stage then become

$$\begin{aligned} R_j(u^0,x^0)\,\Delta u^0 + B_0^\top\lambda^0 &= -r^0(u^0, E_0^{-1}x_{\text{init}}) \\ B_0\Delta u^0 - E_1\Delta x^1 &= -c^0(u^0, E_0^{-1}x_{\text{init}}, x^1). \end{aligned} \tag{35}$$

The result is a KKT system with optimal control structure that can be written in the form of (5).

5.3 Parallel exact line search

A key component in the semismooth Newton solver used in QPALM is the line search procedure that minimizes the augmented Lagrangian along the direction of the Newton step. Thanks to the specific piecewise quadratic structure of the objective, the minimizer along the Newton step can be computed exactly [19, §3.2].

Below, we derive how a bisection algorithm can be used to find the optimal step size in linear time. This is an improvement compared to the complexity of the sorting-based algorithm in [19, Alg. 2]. Finally, we discuss how the line search algorithm can be parallelized and vectorized.

For the sake of simplicity, consider the abstract QP formulation of (1):

$$\begin{aligned} \underset{z}{\textbf{minimize}} \quad & \tfrac{1}{2}z^\top Q z + q^\top z + c \;\triangleq\; \ell(z) \\ \textbf{subject to} \quad & Mz = b \\ & b_l \le Gz \le b_u. \end{aligned} \tag{36}$$

We denote the number of inequality constraints (i.e. the number of rows of G) by m. Given the current iterate z and a descent direction d, the line search procedure aims to find the step size τ that minimizes the merit function $\psi(\tau) \triangleq \ell^\Sigma(z+\tau d)$, where $\ell^\Sigma(z) \triangleq \ell(z) + \frac{1}{2}\,\mathbf{dist}_\Sigma^2\big(Gz + \Sigma^{-1}y;\,[b_l, b_u]\big) + \frac{1}{2}\|z - \bar z\|_{\Gamma^{-1}}^2$ is the Γ-proximal Σ-augmented Lagrangian of (36) with respect to the inequality constraints, parallel to (30). By convexity, the optimal step size τ is the root of $\psi'(\tau) = 0$. If z is feasible w.r.t. the equality constraints, the Newton direction d lies in the null space of M, maintaining feasibility of $z + \tau d$ for any τ, so the equality constraints need not be included in the merit function [26].

5.3.1 Bisection of the step size

It can be shown that the derivative of ψ with respect to the step size is given by [19, Eq. 3.5]

$$\psi'(\tau) = \langle \nabla \ell^{\Sigma}(z + \tau d) \,|\, d \rangle = \eta\tau + \beta + \langle \delta \,|\, [\delta\tau - \alpha]_+ \rangle, \tag{37}$$

where $\eta \triangleq d^\top (Q + \Gamma^{-1}) d$, $\beta \triangleq d^\top (Qz + q + \Gamma^{-1}(z - \bar{z}))$, $\alpha \triangleq \begin{pmatrix} \sqrt{\Sigma}^{-1}\big(y + \Sigma(Gz - b_l)\big) \\ -\sqrt{\Sigma}^{-1}\big(y + \Sigma(Gz - b_u)\big) \end{pmatrix}$, and $\delta \triangleq \begin{pmatrix} -\sqrt{\Sigma} G d \\ \sqrt{\Sigma} G d \end{pmatrix}$.
The values of τ where one or more of the constraints change activity are given by the breakpoints $t_i = \alpha_i / \delta_i$. The derivative of the merit function ψ can be evaluated efficiently for a given breakpoint t_j:

$$\psi'(t_j) \overset{(37)}{=} t_j \eta + \beta + \sum_{i \in \mathcal{I}_j} \delta_i \left(\delta_i \tfrac{\alpha_j}{\delta_j} - \alpha_i \right) = t_j \underbrace{\left(\eta + \sum_{i \in \mathcal{I}_j} \delta_i^2 \right)}_{\eta(t_j)} + \underbrace{\beta - \sum_{i \in \mathcal{I}_j} \delta_i \alpha_i}_{\beta(t_j)}, \tag{38}$$

$$\begin{aligned} \text{where} \quad \mathcal{I}_j &\triangleq \mathcal{I}_j^- \cup \mathcal{I}_j^+ \\ \mathcal{I}_j^- &\triangleq \left\{ i \in \mathbb{N}_{[1,2m]} \;\middle|\; \tfrac{\alpha_j}{\delta_j}\delta_i - \alpha_i \geq 0 \wedge \delta_i < 0 \right\} = \left\{ i \in \mathbb{N}_{[1,2m]} \;\middle|\; t_i \geq t_j \wedge \delta_i < 0 \right\} \\ \mathcal{I}_j^+ &\triangleq \left\{ i \in \mathbb{N}_{[1,2m]} \;\middle|\; \tfrac{\alpha_j}{\delta_j}\delta_i - \alpha_i > 0 \wedge \delta_i > 0 \right\} = \left\{ i \in \mathbb{N}_{[1,2m]} \;\middle|\; t_i < t_j \wedge \delta_i > 0 \right\}. \end{aligned} \tag{39}$$

A simple bisection technique can be used to find the root τ of the monotonically increasing and piecewise linear function $\psi'(\tau)$.

1. Initialize interval $\underline{\tau} = 0$, $\overline{\tau} = +\infty$
2. Select some j for which $\underline{\tau} < t_j < \overline{\tau}$
3. If no such j exists, find τ by interpolation between $\underline{\tau}$ and $\overline{\tau}$, and return
4. If $\psi'(t_j) = 0$, return $\tau = t_j$
5. If $\psi'(t_j) < 0$, set $\underline{\tau} = t_j$
6. If $\psi'(t_j) > 0$, set $\overline{\tau} = t_j$
7. Go to 2.

As an invariant, we have $\psi'(\underline{\tau}) < 0$ and $\psi'(\overline{\tau}) > 0$. If no breakpoint t_j can be found in the interval $(\underline{\tau},\ \overline{\tau})$ in step 3, then ψ' is linear on that interval, and finding its root τ is trivial. Thanks to (38), the value of $\psi'(t_j)$ does not have to be recomputed from scratch at each iteration: if $t_k > t_j$, we have

$$\eta(t_k) = \eta(t_j) - \sum_{i \in \mathcal{I}_{j,k}^-} \delta_i^2 + \sum_{i \in \mathcal{I}_{j,k}^+} \delta_i^2, \quad \text{where} \quad \begin{cases} \mathcal{I}_{j,k}^- \triangleq \left\{ i \in \mathbb{N}_{[1,2m]} \;\middle|\; t_k > t_i \geq t_j \wedge \delta_i < 0 \right\} \\ \mathcal{I}_{j,k}^+ \triangleq \left\{ i \in \mathbb{N}_{[1,2m]} \;\middle|\; t_j \leq t_i < t_k \wedge \delta_i > 0 \right\}, \end{cases} \tag{40}$$

significantly reducing the number of terms that need to be summed compared to a naive implementation of (38). Similar update formulas can be obtained for $\beta(t_k)$ and for the case where $t_k < t_j$.

Unlike the original implementation in [19, Alg. 2], we do not sort the breakpoints t_i, as this would require $\mathcal{O}(m \log m)$ operations. An algorithm like quickselect or introselect [29] can be used to select a breakpoint in $(\underline{\tau},\ \overline{\tau})$. As a side effect, it partitions the interval into breakpoints smaller and greater than the selected pivot t_j, and the same partitioning is applied to α and δ. This partitioning helps avoid scanning the full vectors α and δ for the evaluation of (40) at each trial breakpoint t_j. The cost of performing introselect is linear in the number of breakpoints in $(\underline{\tau},\ \overline{\tau})$. If the median element is selected, this number is halved at each iteration, resulting in an overall linear complexity, $\mathcal{O}(m)$.

Rather than selecting the median breakpoint as the pivot, it is also possible to select a specific value t_j directly, and partition the interval based on that choice. While the average cost of this approach may be lower than a version using introselect, its worst case complexity is quadratic in the number of breakpoints. Despite the asymptotic concerns, such a strategy could nevertheless be useful to prune the search space early on: in practice, the optimal step size is rarely greater than one, so selecting the largest breakpoint below one as t_j in the first iteration eliminates a potentially large number of uninteresting breakpoints.

Practical implementations can be further optimized by storing just t_i and $\delta_i|\delta_i|$ instead of t_i, α_i and δ_i. This choice reduces memory usage, memory bandwidth and the number of branches, and avoids computing square roots. The update formulas from (40) can then be written as

$$\eta(t_k) = \eta(t_j) + \sum_{i \in \mathcal{I}_{j,k}} \delta_i |\delta_i| \;\text{ and }\; \beta(t_k) = \beta(t_j) - \sum_{i \in \mathcal{I}_{j,k}} t_i \delta_i |\delta_i| \;\text{ where }\; \mathcal{I}_{j,k} \triangleq \left\{ i \in \mathbb{N}_{[1,2m]} \;\middle|\; t_j \leq t_i < t_k \right\}.$$

5.3.2 Parallelization of the line search algorithm

The evaluation of the vectors α, δ and t requires only stage-wise matrix-vector products and element-wise operations, and can thus be fully parallelized and vectorized. The breakpoints are then partitioned in parallel: non-finite values of t_i, which do not affect $\psi'(\tau)$, are removed (e.g. due to one-sided constraints or constraints with gradients orthogonal to d), and negative breakpoints are isolated, as they are not considered during the bisection (but they do affect $\psi'(\tau)$). Each thread produces its local partitions in parallel. After selecting a trial breakpoint t_j, all threads update their partitions using t_j as a pivot and compute their local sums to evaluate $\eta(t_j)$ and $\beta(t_j)$, which are then combined to compute $\psi'(t_j)$. We can leverage a vast literature on optimal and parallel selection algorithms [40, 42, 27]; although these operations are unlikely to be a bottleneck in the full implementation of CyQPALM. Once the remaining interval in the bisection algorithm is sufficiently small, the local partitions of all threads can be merged to complete the procedure on a single thread, thereby reducing synchronization overhead. When the size of the interval drops below a second threshold, the remaining breakpoints are sorted, and a linear search is used to find the optimal step size τ (this is faster than continuing the bisection all the way down).

6 Factorization updates

At every iteration of QPALM's inner semismooth Newton method, the generalized Hessian matrices $H_j(u, x)$ defined in (31) may vary because of changes in the active set at the current iterate, $\mathcal{J}_j(u, x)$. If the activity of only a limited number of constraints changes, the factorization of the new Newton system can be expressed as a low-rank modification of the factorization of the previous system. We will refer to this modification of triangular factors as a *factorization update*. If the rank of the modification is sufficiently small, factorization updates require fewer operations than a full factorization of the entire matrix (without reusing the previous factors) [15] and enable significant performance gains in the resulting QPALM solver.

To derive factorization update formulas for the full block Cholesky factor of the KKT system with optimal control structure from Section 4, we make use of the hyperbolic Householder transformations introduced in [36] as the main building block.

Consider an iterate $(\mathbf{u}, \mathbf{x})$ for which an existing factorization of the Newton system (5) is available. We now wish to obtain a factorization of the corresponding Newton system at a different point $(\tilde{\mathbf{u}}, \tilde{\mathbf{x}})$. By (31), we have that

$$H_j(\tilde{u}^j, \tilde{x}^j) = H_j(u^j, x^j) + \begin{pmatrix} D_j^\top \\ C_j^\top \end{pmatrix} \left(\Sigma_j^{\mathcal{J}}(\tilde{u}^j, \tilde{x}^j) - \Sigma_j^{\mathcal{J}}(u^j, x^j) \right) \begin{pmatrix} D_j & C_j \end{pmatrix}. \tag{41}$$

When the active sets at (u^j, x^j) and $(\tilde{u}^j, \tilde{x}^j)$ differ for a small number of constraints only, the diagonal matrix $\Delta\Sigma_j \triangleq \Sigma_j^{\mathcal{J}}(\tilde{u}^j, \tilde{x}^j) - \Sigma_j^{\mathcal{J}}(u^j, x^j)$ has a small number of nonzero entries, and $H_j(\tilde{u}^j, \tilde{x}^j)$ and $H_j(u^j, x^j)$ differ by a low-rank term.

6.1 Parallel factorization updates in CyQLONE

To see how the low-rank updates in (41) affect the factorization computed by the CyQLONE method, we will derive a factorization of the updated KKT matrix $\widetilde{\mathcal{K}}$, which we define as the matrix $\mathcal{K}$ from (14) with a term $\mathcal{Y}\mathcal{S}\mathcal{Y}^\top$ added to the top-left block:

$$\widetilde{\mathcal{K}} \triangleq \left(\begin{array}{c|c} \widetilde{\mathcal{R}} & \mathcal{A}^\top \\ \hline \mathcal{A} & 0 \end{array}\right) = \left(\begin{array}{c|c} \mathcal{R} & \mathcal{A}^\top \\ \hline \mathcal{A} & 0 \end{array}\right) + \left(\begin{array}{c} \mathcal{Y} \\ \hline 0 \end{array}\right) \mathcal{S} \left(\begin{array}{c} \mathcal{Y} \\ \hline 0 \end{array}\right)^\top \tag{42}$$

$$\overset{(15)}{=} \left(\begin{array}{c|c} L^{\mathcal{R}} & \\ \hline L^{\mathcal{A}} & L^{\mathcal{M}} \end{array}\right) \left(\begin{array}{c|c} \mathcal{D} & \\ \hline & -\mathrm{I} \end{array}\right) \left(\begin{array}{c|c} L^{\mathcal{R}} & \\ \hline L^{\mathcal{A}} & L^{\mathcal{M}} \end{array}\right)^\top + \left(\begin{array}{c} \mathcal{Y} \\ \hline 0 \end{array}\right) \mathcal{S} \left(\begin{array}{c} \mathcal{Y} \\ \hline 0 \end{array}\right)^\top. \tag{43}$$

Following the structure of the algorithm in Section 4, we apply a factorization update to $\mathcal{R}$ first, and then update the Schur complement. To this end, we use a hyperbolic Householder transformation $\breve{\mathcal{Q}}^{\mathcal{R}}$ such that

$$\begin{pmatrix} \tilde{L}^{\mathcal{R}} & 0 & 0 \\ \tilde{L}^{\mathcal{A}} & L^{\mathcal{M}} & \Xi \end{pmatrix} = \begin{pmatrix} L^{\mathcal{R}} & 0 & \mathcal{Y} \\ L^{\mathcal{A}} & L^{\mathcal{M}} & 0 \end{pmatrix} \breve{\mathcal{Q}}^{\mathcal{R}}, \qquad \text{with} \quad \breve{\mathcal{Q}}^{\mathcal{R}} \begin{pmatrix} \mathcal{D} & & \\ & -\mathrm{I} & \\ & & \mathcal{S} \end{pmatrix} \breve{\mathcal{Q}}^{\mathcal{R}\top} = \begin{pmatrix} \mathcal{D} & & \\ & -\mathrm{I} & \\ & & \mathcal{S} \end{pmatrix}, \tag{44}$$

followed by another hyperbolic Householder transformation $\breve{\mathcal{Q}}^{\mathcal{M}}$ such that

$$\begin{pmatrix} \tilde{L}^{\mathcal{R}} & 0 & 0 \\ \tilde{L}^{\mathcal{A}} & \tilde{L}^{\mathcal{M}} & 0 \end{pmatrix} = \begin{pmatrix} \tilde{L}^{\mathcal{R}} & 0 & 0 \\ \tilde{L}^{\mathcal{A}} & L^{\mathcal{M}} & \Xi \end{pmatrix} \breve{\mathcal{Q}}^{\mathcal{M}}, \qquad \text{with} \quad \breve{\mathcal{Q}}^{\mathcal{M}} \begin{pmatrix} \mathcal{D} & & \\ & -\mathrm{I} & \\ & & \mathcal{S} \end{pmatrix} \breve{\mathcal{Q}}^{\mathcal{M}\top} = \begin{pmatrix} \mathcal{D} & & \\ & -\mathrm{I} & \\ & & \mathcal{S} \end{pmatrix}. \tag{45}$$

Such transformations $\breve{\mathcal{Q}}^{\mathcal{R}}$ and $\breve{\mathcal{Q}}^{\mathcal{M}}$ can be constructed using the algorithms outlined in [36, §III]. From (44)–(45), we conclude that $\tilde{L}^{\mathcal{R}}$, $\tilde{L}^{\mathcal{A}}$ and $\tilde{L}^{\mathcal{M}}$ indeed form the desired factorization of $\widetilde{\mathcal{K}}$:

$$\begin{pmatrix} \tilde{L}^{\mathcal{R}} & 0 \\ \tilde{L}^{\mathcal{A}} & \tilde{L}^{\mathcal{M}} \end{pmatrix} \begin{pmatrix} \mathcal{D} & \\ & -\mathrm{I} \end{pmatrix} \begin{pmatrix} \tilde{L}^{\mathcal{R}} & 0 \\ \tilde{L}^{\mathcal{A}} & \tilde{L}^{\mathcal{M}} \end{pmatrix}^{\top} = \begin{pmatrix} \tilde{L}^{\mathcal{R}} & 0 & 0 \\ \tilde{L}^{\mathcal{A}} & \tilde{L}^{\mathcal{M}} & 0 \end{pmatrix} \begin{pmatrix} \mathcal{D} & & \\ & -\mathrm{I} & \\ & & \mathcal{S} \end{pmatrix} \begin{pmatrix} \tilde{L}^{\mathcal{R}} & 0 & 0 \\ \tilde{L}^{\mathcal{A}} & \tilde{L}^{\mathcal{M}} & 0 \end{pmatrix}^{\top} \tag{46}$$

$$= \begin{pmatrix} L^{\mathcal{R}} & 0 & \Upsilon \\ L^{\mathcal{A}} & L^{\mathcal{M}} & 0 \end{pmatrix} \breve{\mathcal{Q}}^{\mathcal{R}} \breve{\mathcal{Q}}^{\mathcal{M}} \begin{pmatrix} \mathcal{D} & & \\ & -\mathrm{I} & \\ & & \mathcal{S} \end{pmatrix} \breve{\mathcal{Q}}^{\mathcal{M}\top} \breve{\mathcal{Q}}^{\mathcal{R}\top} \begin{pmatrix} L^{\mathcal{R}} & 0 & \Upsilon \\ L^{\mathcal{A}} & L^{\mathcal{M}} & 0 \end{pmatrix}^{\top} \tag{47}$$

$$= \begin{pmatrix} L^{\mathcal{R}} & 0 & \Upsilon \\ L^{\mathcal{A}} & L^{\mathcal{M}} & 0 \end{pmatrix} \begin{pmatrix} \mathcal{D} & & \\ & -\mathrm{I} & \\ & & \mathcal{S} \end{pmatrix} \begin{pmatrix} L^{\mathcal{R}} & 0 & \Upsilon \\ L^{\mathcal{A}} & L^{\mathcal{M}} & 0 \end{pmatrix}^{\top} \overset{(43)}{=} \widetilde{\mathcal{K}}. \tag{48}$$

6.2 Modified Riccati recursion update (steps 1 and 2)

Computing the updated factors $\tilde{L}^{\mathcal{R}}$ and $\tilde{L}^{\mathcal{A}}$ as in (44) can be done using a modified version of [36, Alg. 4], listed in Algorithm 3. This algorithm mirrors the factorization in Algorithm 1. In addition to the updated factors, it also returns blocks of the matrix Ξ that can be used to perform the update of the Schur complement. Following the specific example from Section 4, the second block column of $\mathcal{K}$ is updated as follows:

$$\widetilde{\mathcal{K}}_1 \triangleq \left(\begin{array}{c|c} \widetilde{\mathcal{R}}_1 & \mathcal{A}_1^{\top} \\ \hline \mathcal{A}_1 & 0 \end{array}\right) = \mathcal{K}_1 + \begin{pmatrix} \Upsilon_1 \\ 0 \end{pmatrix} \mathcal{S}_1 \begin{pmatrix} \Upsilon_1 \\ 0 \end{pmatrix}^{\top}, \tag{49}$$

$$\text{where} \quad \Upsilon_1 \triangleq \left(\begin{array}{c|c|c} \begin{matrix} D_3^{\top} \\ C_3^{\top} \\ 0 \end{matrix} & & \\ \hline & \begin{matrix} D_2^{\top} \\ C_2^{\top} \\ 0 \end{matrix} & \\ \hline & & \begin{matrix} D_1^{\top} \\ C_1^{\top} \end{matrix} \end{array}\right) \quad \text{and} \quad \mathcal{S}_1 \triangleq \left(\begin{array}{c|c|c} \Delta\Sigma_3 & & \\ \hline & \Delta\Sigma_2 & \\ \hline & & \Delta\Sigma_1 \end{array}\right). \tag{50}$$

6.3 Low-rank update of the Schur complement (step 3)

Combining the contributions of Algorithm 3 from all block columns, the matrix Ξ from (44) that updates the Schur complement $\tilde{\mathcal{M}} = \mathcal{M} + \Xi \mathcal{S} \Xi^{\top} = \tilde{L}^{\mathcal{M}} \tilde{L}^{\mathcal{M}\top}$ in (45) has the form

$$\Xi = \begin{array}{c} \\ \lambda^3 \\ \lambda^9 \\ \lambda^6 \\ \lambda^0 \end{array} \begin{array}{c} \begin{array}{cccc} c=0 & c=1 & c=2 & c=3 \end{array} \\ \begin{pmatrix} & \vec{\Upsilon}_1 & \overleftarrow{\Upsilon}_1 & \\ \overleftarrow{\Upsilon}_3 & & & \vec{\Upsilon}_3 \\ & & \vec{\Upsilon}_2 & \overleftarrow{\Upsilon}_2 \\ \vec{\Upsilon}_0 & \overleftarrow{\Upsilon}_0 & & \end{pmatrix} \end{array}, \quad \text{for an updated factor} \quad \tilde{L}^{\mathcal{M}} = \begin{array}{c} \\ \lambda^3 \\ \lambda^9 \\ \lambda^6 \\ \lambda^0 \end{array} \begin{array}{c} \begin{array}{cccc} \lambda^3 & \lambda^9 & \lambda^6 & \lambda^0 \\ i=1 & i=3 & i=2 & i=0 \end{array} \\ \begin{pmatrix} \tilde{L}_1 & & & \\ & \tilde{L}_3 & & \\ \tilde{Y}_1 & \tilde{U}_3 & \tilde{L}_2 & \\ \tilde{U}_1 & \tilde{Y}_3 & \tilde{U}_2 & \tilde{L}_0 \end{pmatrix} \end{array}. \tag{51}$$

The blocks $\vec{\Upsilon}_0$ and $\tilde{Y}_3$ can be shown to be zero in the absence of coupling between the first and final stages.

6.4 Cyclic reduction factorization updates of the Schur complement (step 4)

To update the cyclic reduction factorization, we use a systematic application of the following operation to the appropriate blocks of the factors computed by Algorithm 2:

$$\left(\begin{array}{c|cc} \tilde{L}_{11} & 0 & 0 \\ \tilde{L}_{21} & \widetilde{\Xi}_{21} & \widetilde{\Xi}_{22} \\ \tilde{L}_{31} & \widetilde{\Xi}_{31} & \widetilde{\Xi}_{32} \end{array}\right) = \left(\begin{array}{c|cc} L_{11} & \Xi_{11} & \Xi_{12} \\ L_{21} & \Xi_{21} & 0 \\ L_{31} & 0 & \Xi_{32} \end{array}\right) \breve{Q}. \tag{52}$$

This operation can be implemented using the algorithms in [36, §III].

Algorithm 3: Factorization update of a single modified Riccati block column

Input: A, B, C, D and N: OCP data matrices and horizon length
Input: P: number of intervals to partition the horizon into
Input: $c \in \mathbb{N}_{[0,P)}$: index of the block column to update (determines the interval)
Input: $\Delta\Sigma$: changes in the penalty factors
Input: $L_c^{\mathcal{R}}$ (L^R, L^S, L^Q) and $L_c^{\mathcal{A}}$ ($L^B, L^A, A^{\mathrm{cl}}$): existing factorization of $\mathcal{K}_c$
Output: $\tilde{L}_c^{\mathcal{R}}$ ($\tilde{L}^R, \tilde{L}^S, \tilde{L}^Q$) and $\tilde{L}_c^{\mathcal{A}}$ ($\tilde{L}^B, \tilde{L}^A, \tilde{A}^{\mathrm{cl}}$): updated factorization of $\widetilde{\mathcal{K}}_c$
Output: $\vec{\Upsilon}_c, \overleftarrow{\Upsilon}_{c-1}, \mathcal{S}_c$: contribution to the update of the Schur complement

Function UPDATE–BLOCK–COLUMN–RICCATI(c)

$n = N/P$ ▷ Number of stages per interval

$j_1 = n(c-1)+1, \quad j_n = nc$ ▷ Indices of the first and last stages in the selected block column

$$\begin{pmatrix} \Upsilon^u_{j_n} \\ \Upsilon^x_{j_n} \\ \Upsilon^\lambda_{j_n} \end{pmatrix} = \begin{pmatrix} D^\top_{j_n} \\ C^\top_{j_n} \\ 0 \end{pmatrix}, \quad \mathcal{S}_{j_n} = \Delta\Sigma_{j_n}$$

for $j = j_n, j_n-1, j_n-2, \ldots, j_1$

$m_j = \operatorname{rank} \mathcal{S}_j$

$$\begin{pmatrix} \tilde{L}^R_j & 0 \\ \tilde{L}^S_j & \Phi^x_j \\ \tilde{L}^B_j & \Phi^\lambda_j \end{pmatrix} = \begin{pmatrix} L^R_j & \Upsilon^u_j \\ L^S_j & \Upsilon^x_j \\ L^B_j & \Upsilon^\lambda_j \end{pmatrix} \breve{Q}^u_j \quad \text{where } \breve{Q}^u_j \text{ is } \begin{pmatrix} \mathrm{I} & \\ & \mathcal{S}_j \end{pmatrix}\text{-orth. [36]}$$ ▷ (hyh) $\begin{smallmatrix} m_j n_u^2 \\ + \\ 4 m_j n_u n_x \end{smallmatrix}$

if $j > j_1$

$\tilde{A}^{\mathrm{cl}}_j = A^{\mathrm{cl}}_j + \Phi^\lambda_j \mathcal{S}_j \Phi^{x\top}_j$ ▷ (gemm) $m_j n_x^2$

$$\begin{pmatrix} \Upsilon^u_{j-1} \\ \Upsilon^x_{j-1} \\ \Upsilon^\lambda_{j-1} \end{pmatrix} = \begin{pmatrix} B^\top_{j-1}\Phi^x_j & D^\top_{j-1} \\ A^\top_{j-1}\Phi^x_j & C^\top_{j-1} \\ \Phi^\lambda_j & 0 \end{pmatrix}, \quad \mathcal{S}_{j-1} = \begin{pmatrix} \mathcal{S}_j & \\ & \Delta\Sigma_{j-1} \end{pmatrix}$$ ▷ (gemm) $m_j n_x (n_x + n_u)$

$$\begin{pmatrix} \tilde{L}^Q_j & 0 \end{pmatrix} = \begin{pmatrix} L^Q_j & \Phi^x_j \end{pmatrix} \breve{Q}^x_j \quad \text{where } \breve{Q}^x_j \text{ is } \begin{pmatrix} \mathrm{I} & \\ & \mathcal{S}_j \end{pmatrix}\text{-orth.}$$ ▷ (hyh) $m_j n_x^2$

$\mathcal{S}_c = \mathcal{S}_{j_1}$

$$\begin{pmatrix} \tilde{L}^A_{j_1} & \vec{\Upsilon}_c \\ -\tilde{L}^{Q\,-\top}_{j_1} & \overleftarrow{\Upsilon}_{c-1} \end{pmatrix} = \begin{pmatrix} L^A_{j_1} & \Phi^\lambda_{j_1} \\ -L^{Q\,-\top}_{j_1} & 0 \end{pmatrix} \breve{Q}^x_{j_1}$$ ▷ (hyh apply) $4 m_{j_1} n_x^2$

For each column i of $L^{\mathcal{M}}$, consider the nonzero blocks, and update them using the columns of Ξ that have nonzero elements in row i, as described by (52). The order in which the columns are updated is the same as the order used during the factorization in Algorithm 2: in the first level, all odd columns are updated, then all multiples of two that are not multiples of four in the second level, etc. Within each level, different columns can be updated in parallel. Note that applying (52) introduces the additional nonzero blocks $\widetilde{\Xi}_{22}$ and $\widetilde{\Xi}_{31}$, which have to be taken into account in the next level of the algorithm. For the specific case of the matrix $L^{\mathcal{M}}$ defined in (27) and with Ξ and $\tilde{L}^{\mathcal{M}}$ from (51), the updating procedure amounts to

$$\underline{\text{Level 0}} \quad \left(\begin{array}{c|c} \tilde{L}_1 & 0 \\ \check{U}_1 & \overleftarrow{\Upsilon}_0^{(1)} \\ \tilde{Y}_1 & \vec{\Upsilon}_2^{(1)} \end{array}\right) = \left(\begin{array}{c|cc} L_1 & \vec{\Upsilon}_1 & \overleftarrow{\Upsilon}_1 \\ U_1 & \overleftarrow{\Upsilon}_0 & 0 \\ Y_1 & 0 & \vec{\Upsilon}_2 \end{array}\right) \breve{Q}_1, \quad \left(\begin{array}{c|c} \tilde{L}_3 & 0 \\ \check{U}_3 & \overleftarrow{\Upsilon}_2^{(1)} \\ \tilde{Y}_3 & \vec{\Upsilon}_0^{(1)} \end{array}\right) = \left(\begin{array}{c|cc} L_3 & \vec{\Upsilon}_3 & \overleftarrow{\Upsilon}_3 \\ U_3 & \overleftarrow{\Upsilon}_2 & 0 \\ Y_3 & 0 & \vec{\Upsilon}_0 \end{array}\right) \breve{Q}_3,$$

$$\underline{\text{Level 1}} \quad \left(\begin{array}{c|c} \tilde{L}_2 & 0 \\ \check{U}_2 & \overleftarrow{\Upsilon}_0^{(2)} \end{array}\right) = \left(\begin{array}{c|cc} L_2 & \vec{\Upsilon}_2^{(1)} & \overleftarrow{\Upsilon}_2^{(1)} \\ U_2 & \overleftarrow{\Upsilon}_0^{(1)} & \vec{\Upsilon}_0^{(1)} \end{array}\right) \breve{Q}_2,$$

$$\underline{\text{Level 2}} \quad \left(\begin{array}{c|c} \tilde{L}_0 & 0 \end{array}\right) = \left(\begin{array}{c|c} L_0 & \overleftarrow{\Upsilon}_0^{(2)} \end{array}\right) \breve{Q}_0.$$

The orthogonality metrics of the hyperbolic Householder transformations $\breve{Q}_i$ are determined by $\mathcal{S}_c$ in (50):

$\breve{Q}_1$ and $\breve{Q}_3$ are $\begin{pmatrix} -\mathrm{I} & & \\ & \mathcal{S}_1 & \\ & & \mathcal{S}_2 \end{pmatrix}$- and $\begin{pmatrix} -\mathrm{I} & & \\ & \mathcal{S}_3 & \\ & & \mathcal{S}_0 \end{pmatrix}$-orthogonal, $\breve{Q}_2$ and $\breve{Q}_0$ are $\begin{pmatrix} -\mathrm{I} & & & \\ & \mathcal{S}_3 & & \\ & & \mathcal{S}_0 & \\ & & & \mathcal{S}_1 \\ & & & & \mathcal{S}_2 \end{pmatrix}$-orthogonal.

Blocks shown in gray are zero in the absence of coupling between the first and final stages. They are included here because the same structure generalizes to larger problems. Levels $l = \log_2 P$ and $l = \log_2 P - 1$ are special cases (at these levels, the system has been reduced to a block 2×2 matrix), while all other levels $l < \log_2 P - 1$ follow the same pattern as shown in (52). The resulting procedure is summarized in Algorithm 4, which follows the same structure as the factorization in Algorithm 2. A graph representation of the algorithm is shown in Figure 8. For brevity, the factorization and update algorithms listed here assume that there is no coupling between the first and final stages;[8] they can easily be extended to include such coupling by replacing levels $l = \log_2 P$ and $l = \log_2 P - 1$ by a simple block 2×2 Cholesky factorization and update, respectively.

Thanks to the similar structure between the factorization (Figure 6) and factorization update procedures (Figure 8), it is possible to switch from factorization updates to re-factorizations at any level in the algorithm. This is useful when the sizes m_i of the update matrices become too large relative to the block size n_x at a certain level in the recursion, making re-factorization – with cost $\mathcal{O}(n_x^3)$ – more economical than factorization updates – with cost $\mathcal{O}(m_i n_x^2)$.

7 Vectorization

Thanks to the parallelism exposed by the CYQLONE permutation from Section 4, operations applied to different stages of the OCP can not only be parallelized across different processors, but they can also be *vectorized*. Vectorization refers to the process of transforming an algorithm into a form where operations are carried out on arrays or vectors rather than on individual values. Consider the example of matrix-vector multiplication Ax: a scalar implementation of this algorithm involves multiplying the individual elements of each row of the matrix A by the corresponding elements of the vector x. One possible vectorized approach could be to compute the sum of the columns of A, weighted by the elements of x:

$$Ax = \underbrace{\begin{pmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 \end{pmatrix}}_{\text{Scalar}} = \begin{pmatrix} | & | & | & | \\ a_1 & a_2 & a_3 & a_4 \\ | & | & | & | \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \underbrace{a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4.}_{\text{Vectorized}} \tag{53}$$

[8] This allows levels $\log_2 P - 1$ and $\log_2 P$ to be handled in the same way as earlier levels, simplifying the pseudocode. For the full, vectorized implementation of CYQLONE, the last levels of the recursion are handled using PCR or PCG instead of CR, so this special case does not occur there.

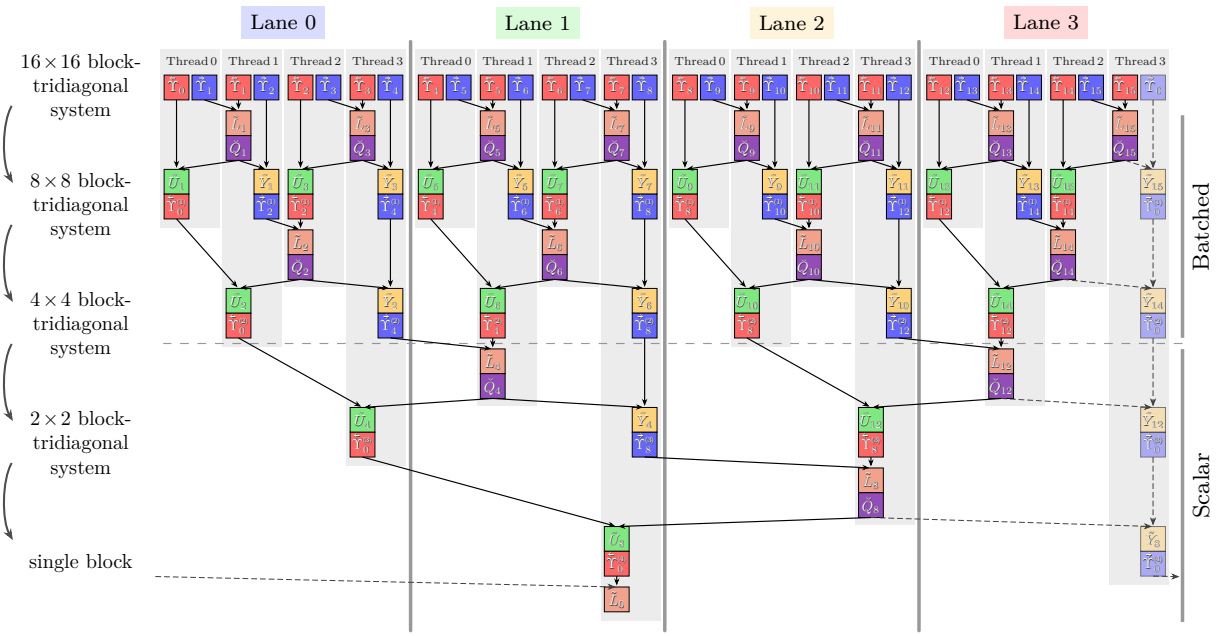

Figure 8. Graph representation of the different steps required for the cyclic reduction factorization updates of a symmetric 16×16 block-tridiagonal matrix. Note the structural similarity to Figure 6. Vector lane assignment, the distinction between scalar and batched levels, and other vectorization aspects will be discussed in Section 7.

Algorithm 4: CYQLONE factorization updates

Input: A, B, C, D and N: OCP data matrices and horizon length
Input: P: number of intervals to partition the horizon into
Input: $L^{\mathcal{R}}$ (L^R, L^S, L^Q), $L^{\mathcal{A}}$ $(L^B, L^A, A^{\mathrm{cl}})$ and $L^{\mathcal{M}}$ (L, Y, U): existing factorization of $\mathcal{K}$
Output: $\tilde{L}^{\mathcal{R}}$ $(\tilde{L}^R, \tilde{L}^S, \tilde{L}^Q)$, $\tilde{L}^{\mathcal{A}}$ $(\tilde{L}^B, \tilde{L}^A, \tilde{A}^{\mathrm{cl}})$ and $\tilde{L}^{\mathcal{M}}$ $(\tilde{L}, \tilde{Y}, \tilde{U})$: updated factorization of $\tilde{\mathcal{K}}$

for $c = 0, \ldots, P-1$ (in parallel)
  $\vec{\Upsilon}_c^{(0)}, \overleftarrow{\Upsilon}_{c-1}^{(0)}, \mathcal{S}_c^{(0)} =$ UPDATE–BLOCK–COLUMN–RICCATI(c) ▷ (steps 1 and 2)
  UPDATE–SCHUR(c) ▷ (step 4)

Function UPDATE–SCHUR(c)
  — sync — ▷ Wait for $\vec{\Upsilon}^{(0)}$, $\overleftarrow{\Upsilon}^{(0)}$
  if $\nu_2^P(c) = 0$: UPDATE–L$(0, c)$
  for $l = 0, \ldots, \log_2(P) - 1$ ▷ Recursion level of cyclic reduction
    — sync — ▷ Wait for $\breve{Q}$
    $i_U = c + 1, \quad i_Y = c + 1 - 2^l$ ▷ Assign the evaluation of blocks $\tilde{U}$ and $\tilde{Y}$ to different threads
    if $\nu_2^P(i_U) = l$: UPDATE–U(l, i_U)
    elif $\nu_2^P(i_Y) = l$: UPDATE–Y(l, i_Y)
    — sync — ▷ Wait for $\vec{\Upsilon}^{(l)}$, $\overleftarrow{\Upsilon}^{(l)}$
    if $\nu_2^P(i_Y) = l + 1$: UPDATE–L$(l+1, i_Y)$

Function UPDATE–L(l, i)
  $\mathcal{S}_i^{(l+1)} = \begin{pmatrix} \mathcal{S}_i^{(l)} & \\ & \mathcal{S}_{i+2^l}^{(l)} \end{pmatrix}, \quad m_i = \operatorname{rank} \mathcal{S}_i^{(l+1)}$
  $\left(\tilde{L}_i \mid 0\right) = \left(L_i \mid \vec{\Upsilon}_i^{(l)} \quad \overleftarrow{\Upsilon}_i^{(l)}\right)\breve{Q}_i$ where $\breve{Q}_i$ is $\begin{pmatrix} -\mathrm{I} & \\ & \mathcal{S}_i^{(l+1)} \end{pmatrix}$-orth. [36] ▷ (hyh) $m_i n_x^2$

Function UPDATE–U(l, i)
  $\left(\tilde{U}_i \mid \overleftarrow{\Upsilon}_{i-2^l}^{(l+1)}\right) = \left(U_i \mid \overleftarrow{\Upsilon}_{i-2^l}^{(l)} \quad 0\right)\breve{Q}_i$ ▷ (hyh apply) $2 m_i n_x^2$

Function UPDATE–Y(l, i)
  $\left(\tilde{Y}_i \mid \vec{\Upsilon}_{i+2^l}^{(l+1)}\right) = \left(Y_i \mid 0 \quad \vec{\Upsilon}_{i+2^l}^{(l)}\right)\breve{Q}_i$ ▷ (hyh apply) $2 m_i n_x^2$

This algorithm parallels Algorithm 2. Color coding matches Figure 8.

The advantages of vectorization become apparent when considering that modern processors implement instructions that operate on an entire vector at once, referred to as *single instruction, multiple data* (SIMD). Examples of SIMD instruction set architecture extensions include AVX2, AVX-512, AVX10, NEON and ARM SVE. Modern SIMD instructions often have the same or a similar throughput as their scalar counterparts, and operate on vector registers whose size is a small power of two, referred to as the *vector length* (e.g. four double-precision elements per vector for AVX2, or eight for AVX-512). The *lane* of a scalar value refers to its position within a vector register. The first lane is lane zero, and the index of the highest lane is one less than the vector length. Many SIMD instructions operate on all lanes in parallel (e.g. an element-wise sum of two vector registers), but instructions that perform horizontal reductions across lanes or that permute the values between lanes are also available. In ideal cases, a vectorized implementation using SIMD can achieve speedups by a factor of the vector length: the scalar variant of the matrix-vector example in (53) requires 16 multiplication instructions, whereas the vectorized implementation requires just four vector multiplication instructions (assuming a vector length of four). Furthermore, the number of elements that can be stored in fast registers inside of the processor generally also increases when using SIMD instructions, possibly resulting in lower memory traffic and improved arithmetic intensity. Finally, because of the higher throughput, the lower number of instructions and the reduced memory traffic, implementations using SIMD typically consume less energy than equivalent scalar implementations [28, 21].

An important step in the vectorization process is so-called *strip-mining* [48], which involves transforming specific loops in the algorithm into loops with fewer iterations that operate on chunks with the same size as the platform's SIMD vector length. In general-purpose implementations of linear algebra algorithms, a common strategy is to apply strip-mining to the loops that iterate over the rows within a given column of a matrix, processing multiple elements in the same column at once using SIMD instructions (similar to (53)). In the following subsection, we discuss the limitations of this row-wise vectorization strategy, and propose vectorization across different matrices in a batch as a more performant alternative.

7.1 Limitations of classical row-wise vectorization

Most vectorized general-purpose linear algebra libraries such as BLAS, LAPACK, Eigen and BLASFEO make use of row-wise vectorization where multiple adjacent elements in the same column are loaded in a single SIMD register. If the number of rows of the matrix is not an integer multiple of the vector length, the remainders at the bottom of each column do not fill an entire SIMD register, leading to poor vector lane utilization. [9] When dealing with large matrices, the fraction of time taken up by the processing of this remainder is negligible. However, for small matrices, this part of the algorithm may take up a significant portion of the overall run time. Furthermore, triangular matrices — which are important workhorses in numerical linear algebra, used in Cholesky factorizations, QR factorizations, block-Householder reflectors, etc. — have the property that the number of nonzero elements differs in each column. As a result, there will always be a remainder in most of the columns, even if the number of rows of the matrix is a multiple of the vector length. For example, if the matrix A in (53) is lower triangular, 6 of the 16 multiplications are redundant multiplications by zero, reducing the efficiency of a vectorized implementation.

In addition to the loss of efficiency because of nonuniform remainders, vectorization can also be prevented by inherent dependencies encountered in certain linear algebra operations. For example, Cholesky factorization requires computing the inverse square root of the pivot, multiplying the subdiagonal elements by this value, and then subtracting the symmetric outer product of the result from the trailing submatrix to obtain the Schur complement. The top left element of the Schur complement is then used as the next pivot (cf. §3.1). Because of the dependency of the second pivot on the first pivot, it is not possible to compute the inverse square roots of multiple pivots at once, preventing the vectorization of this operation. Vectorization can still be used to multiply multiple subdiagonal elements by the inverse square root of the pivot at once, but the scalar divisions and square roots dominate the run time for small matrices. [10] For the Cholesky factorization specifically, vector lane utilization is further reduced because of the triangular structure of the resulting Cholesky factor and the symmetry of the Schur complement, as discussed in the previous paragraph.

[9] The remainder could be processed using scalar operations, or using operations with successively smaller vector lengths, e.g. a remainder of three could be implemented using a SIMD instruction with vector length two and a scalar instruction rather than three scalar instructions. Alternatively, padding could be used to round up the number of rows to the next multiple of the vector length. However, this places the burden of properly padding the input data on the user of the software. If the target architecture supports it (e.g. AVX-512 or ARM SVE), masked or predicated SIMD instructions could be used to avoid padding or scalar loops.

[10] The latency and reciprocal throughput of division and (inverse) square root instructions are much higher than for multiplication and addition instructions.

7.2 Batch-wise vectorization

When the same operation is applied to multiple matrices of the same size (as is the case in Algorithm 2), the limitations discussed in the previous section can be avoided by vectorizing across different matrices rather than along the rows of a single matrix. We will refer to the strip-mining of the loop that iterates over the different matrices in the batch as *batch-wise* vectorization. SIMD instructions are then applied to the elements of just one row of the matrices at a time, and there are no dependencies between elements of different matrices in the batch. For example, when applying batch-wise vectorization to the Cholesky factorization, the inverse square roots of the first pivots of all matrices in the batch can be computed at once, instead of one at a time (for the row-wise vectorization case discussed earlier).

Furthermore, the batch-wise vectorization of batched linear algebra routines is also easier to implement than row-wise vectorization: the scalar versions of these routines can simply be applied element-wise to tuples containing the values for the corresponding elements of all matrices in the batch. In contrast, row-wise vectorization requires some operations to be carried out using scalar arithmetic, scalars may need to be broadcast to a full vector register, and masked operations are needed for the remainders.

A downside of batch-wise vectorization is that the size of the working set is larger than for the case where a single matrix is processed at a time. Since all matrices in the batch are processed simultaneously, the cache utilization is higher, which affects performance for large matrices (cf. Figure 2). In optimal control applications, the sizes of the matrices are determined by the number of states and the number of controls, which are generally relatively small. Consequently, cache utilization is less of a concern than optimal vector lane utilization, and we find that batch-wise vectorization is highly effective for this application.

7.2.1 Compact batched matrix storage format

SIMD load and store operations are most efficient when the elements of the SIMD vector are stored contiguously in memory. When using row-wise vectorization, this is achieved by storing the matrix in *column-major order*, which is a commonly used storage format for dense matrices. When using batch-wise vectorization, e.g. by strip-mining along the time dimension of an OCP, contiguous storage can be achieved by storing the matrices in memory in such a way that elements of matrices of different stages are interleaved. We refer to this interleaved matrix storage format as *compact storage*. Step 1 of Figure 9 visualizes the conversion to a compact storage format for four 3-by-3 matrices A_0 through A_3 and a vector length of two.

Practically, the most common and intuitive way to store N different m-by-n matrices is to store the matrices A_k back-to-back, each in column-major order. We will refer to this format as the *conceptual storage format*. Using zero-based indexing, the element at row r and column c of the j-th matrix would then be stored in memory at offset $r + mc + mnj$. In contrast, the offset of the same element in compact storage with vector length v would be $j \,\%\, v + vr + vmc + vmn\lfloor j/v \rfloor$. The naive storage format can be written as $(m, n, N){:}(1, m, mn)$ in CuTe notation [5], while the CuTe layout of the interleaved storage format can be written as $\big(m, n, (v, \lceil N/v \rceil)\big){:}\big(v, vm, (1, vmn)\big)$.

7.3 Vectorized modified Riccati recursion (steps 1 and 2)

The batch-wise vectorization of Algorithm 1 is relatively straightforward, because it closely resembles the existing parallel implementation: First, the horizon is partitioned into v partitions, where v is the available vector length. All stages within each partition are assigned to the same vector lane, corresponding to strip mining across the loop of line 1 in Algorithm 2. Figure 10 illustrates how this corresponds to distributing the block columns of the matrix $\mathcal{K}$ across different vector lanes: All blocks within a given block column c are assigned to the same lane, highlighted by the four background colors. Since the control flow in Algorithm 1 does not depend on the selected interval c, and since there are no dependencies between different intervals, all operations can be effectively vectorized. Second, after partitioning the horizon according to the vector length v, further subdivide these intervals to distribute them across p available processors. The total parallelism is then $P = v \cdot p$. We will refer to these two levels of partitions as the v-partition and p-partition, respectively. An example of this type of partitioning with $v = 4$ and $p = 2$ is shown in the right matrix of Figure 10: stage 0/8 is processed by lane 0 of processor 0, stage 1 is processed by lane 0 of processor 1; stage 2 is processed by lane 1 of processor 0, stage 3 is processed by lane 1 of processor 1, etc.

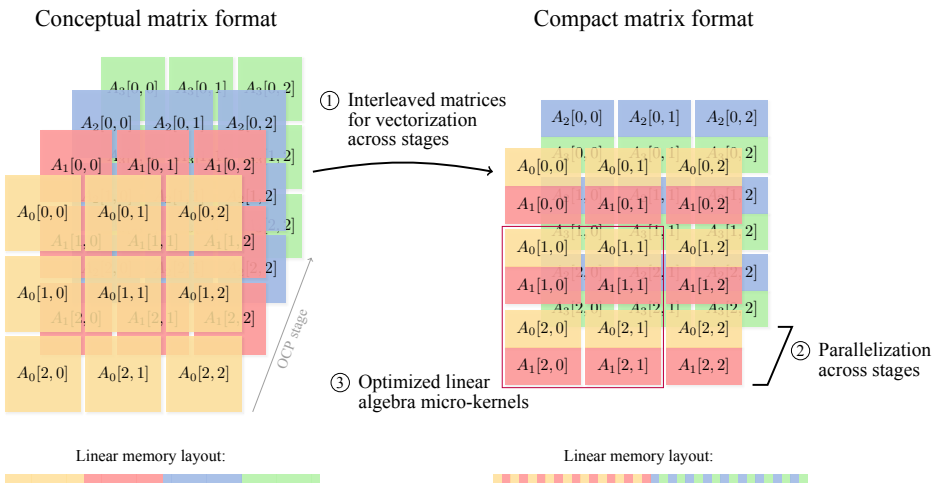

Figure 9. Visual representation of the conversion to compact storage format for four 3-by-3 matrices. Each batch of two matrices is interleaved to match a vector length of two, enabling DLP using SIMD (1). Multiple batches can be processed in parallel by different processors, exploiting TLP using multi-threading (2). Finally, the compactly stored matrices can be processed using specialized batched linear algebra routines with micro-kernels that maximize ILP (3).

7.4 Vectorized Schur complement computation (step 3)

The evaluation of the blocks of the Schur complement (see Section 4.3) may involve data from stages in different vector lanes. For reasons that will become apparent in the following section, we do not yet use an odd–even permutation of $\lambda^0, \lambda^2, \lambda^4, \lambda^6$, resulting in a 4×4 block-tridiagonal Schur complement matrix as shown in Figure 10:

$$-\mathcal{K}/\mathcal{R} = \begin{array}{c} \\ \lambda^0 \\ \lambda^2 \\ \lambda^4 \\ \lambda^6 \end{array} \begin{array}{c} \begin{array}{cccc} \lambda^0 & \lambda^2 & \lambda^4 & \lambda^6 \end{array} \\ \begin{pmatrix} \overleftarrow{M}_0 + \vec{M}_0 & K_0^\top & & \\ K_0 & \overleftarrow{M}_1 + \vec{M}_1 & K_1^\top & \\ & K_1 & \overleftarrow{M}_2 + \vec{M}_2 & K_2^\top \\ & & K_2 & \overleftarrow{M}_3 + \vec{M}_3 \end{pmatrix} \end{array}. \tag{54}$$

The blocks M_i and K_i are stored in vector lane i. For example, consider the blocks in column x^1 and rows λ^0, λ^2 in the left matrix of Figure 10. The result of the product $K_0 \triangleq \overleftarrow{K}_1^\top = -L_1^A L_1^{Q\,-1}$ (where both of the operands are stored in lane 1) ends up in the first subdiagonal block of the Schur complement (row λ^2, column λ^0), which is stored in lane 0. Similarly, the operands of $\overleftarrow{M}_0 = L_1^{Q\,-\top} L_1^{Q\,-1}$ are in lane 1, but the result is added to the first diagonal block $M_0 \triangleq \overleftarrow{M}_0 + \vec{M}_0$ of the Schur complement (row λ^0, column λ^0), stored in lane 0. On the other hand, the result of the symmetric products $L_3^B L_3^{B\,\top} + L_2^B L_2^{B\,\top} + L_1^B L_1^{B\,\top} + L_1^A L_1^{A\,\top}$ is stored in the same lane as the operands, so no special handling is required for this operation. [11]

In general, we see that the results of operations involving the blocks $L_{k_1}^{Q\,-\top}$ of the first p-partition within each v-partition end up in a different vector lane. Practically, this is achieved by performing a lane-wise rotation before storing the results of these operations to memory. Such rotations can be implemented using one of the various lane permutation instructions available in modern SIMD ISA extensions.

7.5 Vectorized cyclic reduction (step 4)

Historically, cyclic reduction has been used on vector machines [25, 23], shared [20] and distributed memory multicore systems [24, 39], and more recently on GPUs [50]. In this section, we combine both vectorization and shared-memory multithreading to increase the available parallelism on modern consumer CPUs.

The basic idea is simple: At each level l in the cyclic reduction algorithm, $N/2^{l+1}$ columns are factorized and eliminated. The operations required are the same for each column, so vectorization and parallelization across these columns is straightforward (similar to Section 7.3). However, updating the remaining even equations

[11] This is by design: since this is the most expensive operation in the Schur complement evaluation, we want to avoid the slight overhead of lane crossings.

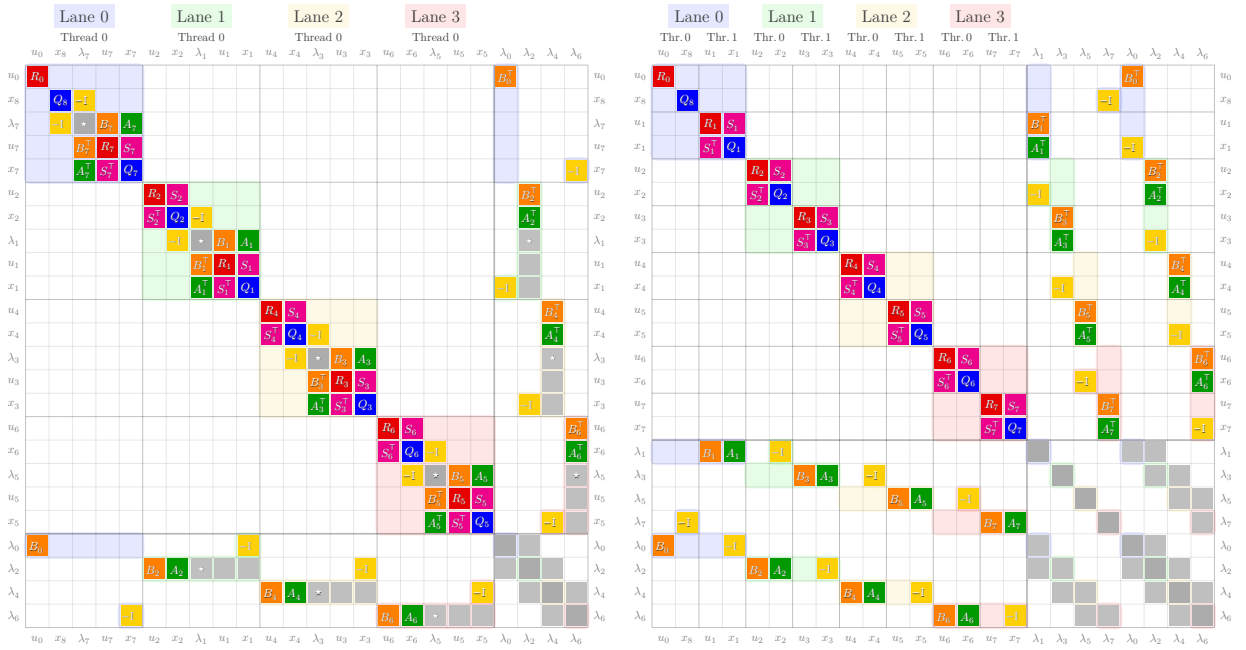

Figure 10. Vectorization of the modified Riccati recursion with vector length $v = 4$ on $p = 1$ (left) and $p = 2$ (right) processors. The distribution of the data across the four vector lanes is indicated by a blue, green, yellow or red background.

at each level is more involved, and requires vector lane crossings (as in Section 7.4). Furthermore, because of the halving of the problem size at each level, strides and indices change in nontrivial ways. In the final levels of CR, the size of the reduced system drops below the vector length, leading to poor resource utilization if not handled as a special case.

We will now consider the CR method in isolation, independent of any optimal control structure of the original system. After all, once the Schur complement has been evaluated as described by Section 7.4, we are left with a general block-tridiagonal matrix. The graphical representation of the CR algorithm applied to a 16×16 block-tridiagonal matrix in Figure 6 will be used as the main visual tool to motivate the approach. It visualizes the blocks of the original system and the intermediate reduced systems ($M_i^{(l)}$ and $K_i^{(l)}$), and the blocks of the resulting block Cholesky factor (L_i, U_i and Y_i). The colors of these blocks match the ones used in Algorithm 1. Additionally, the figure shows which of the four vector lanes on which of the four threads is used to compute each block, with arrows to indicate data dependencies between the blocks. We will refer to matrices that belong to the four vector lanes within the same thread as a *batch*. For example, matrices L_1, L_5, L_9 and L_{13} form a batch (mapped to thread 1). The structure of the scheduling in Figure 6 is chosen in such a way that all matrices in the same batch always undergo the same operation. The following sections explain the derivation of this scheduling in more detail.

7.5.1 Assignment of matrices to batches for effective vectorization

In order to come up with a practical scheduling and vector lane assignment scheme that allows efficient application of vectorized operations, let us first consider the level in the recursion where the number of block columns being eliminated is equal to the vector length. For a 16×16 block matrix and a vector length of four, this happens at the second level, as shown in Figure 6 (at the dividing line between *Batched* and *Scalar* operations). This level is the last level at which the same operations are applied to at least four blocks, and we impose that all four blocks being computed (e.g. $M_0^{(2)}$, $M_4^{(2)}$, $M_8^{(2)}$ and $M_{12}^{(2)}$ in the figure) belong to a single batch, so that they are processed by different vector lanes. This choice minimizes unused lanes and thus maximizes the number of levels that can be processed using fully vectorized operations. From there on, the lane assignment of the remaining blocks is performed by working upwards to the earlier levels, introducing additional batches in such a way that blocks with dependencies between them are stored in the corresponding lanes of their respective batches. This is not possible for every operation in the CR algorithm: In particular, the subtraction of $Y_k Y_k^\top$ from the diagonal blocks necessarily crosses over into different lanes for the last batch of each level. In Figure 6, the arrows from blocks Y_3, Y_7 and Y_{11} to the subsequent diagonal blocks cross the thick gray lines that separate the different vector lanes. These cross-lane operations are

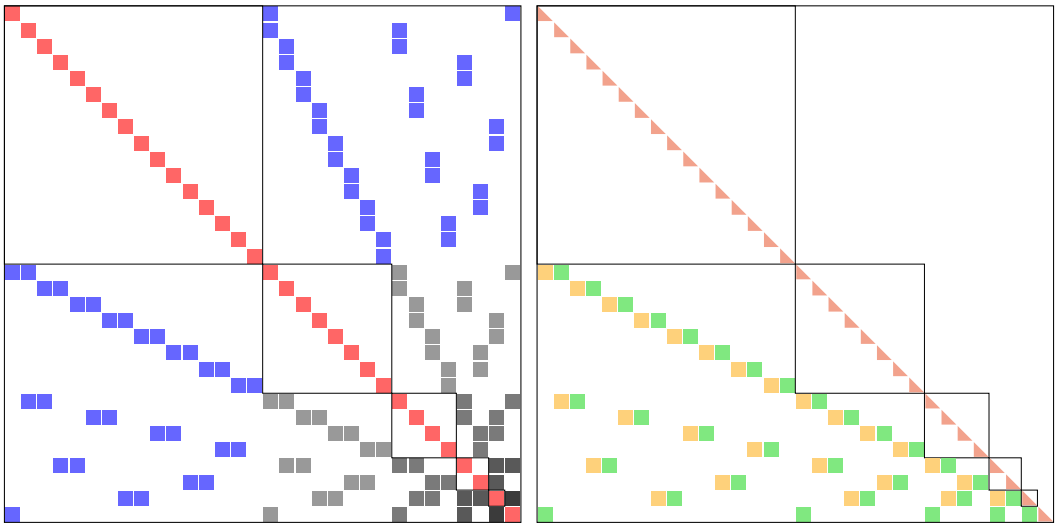

Figure 11. Permutation of a 32×32 block-tridiagonal matrix that enables cyclic reduction down to a single block (left) and its Cholesky factor (right). Color coding matches Figure 4.

limited to a single operation on a single batch per level, and will be handled in the same way as the cross-lane products in the evaluation of the Schur complement in Section 7.4.

7.5.2 Handling of the final scalar levels

Once the size of the reduced system becomes less than or equal to the vector length, it is no longer possible to fill complete batches, because there are simply not enough matrices that undergo the same operation. At this point, one could process the final levels in the recursion using standard (non-batched) linear algebra routines on individual matrices, using vectorization along the row dimension.

However, switching to a scalar implementation of CR may sometimes be suboptimal: converting from the compact storage format to scalar matrices takes time, and linear algebra operations on individual matrices are less performant than batched operations (for small matrices). Keeping the matrices in compact storage format is also an option, but this leads to poor utilization of the available vector lanes. To address these drawbacks, we will consider two alternatives to CR for the final levels: a variant of CR known as parallel cyclic reduction (PCR) and a preconditioned conjugate gradient method (PCG).

PCR [50] addresses the poor utilization in the final levels of CR as follows: instead of reducing an $N \times N$ block-tridiagonal matrix to an $N/2 \times N/2$ block-tridiagonal matrix by eliminating odd block rows as in CR, PCR eliminates both the odd and the even block rows, resulting in not one, but two $N/2 \times N/2$ block-tridiagonal matrices. In $\log_2 N$ steps, PCR reduces an $N \times N$ block-tridiagonal matrix down to N matrices of a single block each. In essence, PCR computes the Cholesky factors of N different permutations of the original block-tridiagonal matrix (every block row of the original matrix appears last in exactly one of these permutations), whereas CR computes just a single Cholesky factor. The PCR algorithm is listed in Algorithm 6 of Appendix A. Eliminating both the odd and even block rows at each level obviously increases the total number of required operations compared to CR, but PCR maintains full vector lane utilization throughout, and completely avoids the back substitution step needed to recover the solution in standard CR. Because of the higher throughput of matrix–matrix operations compared to matrix–vector operations, this is often a worthwhile trade-off, especially if the matrix dimensions are small or if multiple systems need to be solved using the same factorization (cf. Section 6).

A second solution strategy is to use an iterative method such as the preconditioned conjugate gradient method (PCG) to solve the small block-tridiagonal system at the output of the last batched level. This works well because one of the properties of CR is that the off-diagonal blocks decay faster than the diagonal ones with increasing levels of recursion [1, 16]. In combination with a suitable preconditioner such as the recently proposed symmetric stair preconditioner [3], we can expect PCG to converge in a few iterations.

To end up with a block-tridiagonal system of the same size as the vector length (which is the desired format for both PCR and PCG), we permute the final rows of the Schur complement matrix: the permutation for full CR of a 32×32 matrix is shown in Figure 11 (cf. Figure 4), and a variant that uses CR only down to

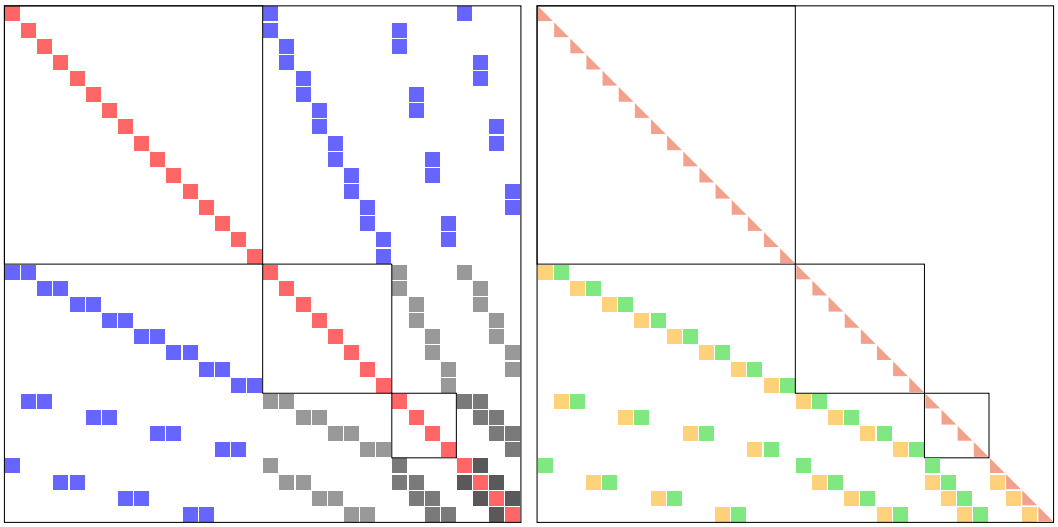

Figure 12. Permutation of a 32×32 block-tridiagonal matrix that enables cyclic reduction down to an unpermuted 4×4 block-tridiagonal matrix (left) and its Cholesky factor (right). Color coding matches Figure 4. Unlike Figure 11, this permutation allows the use of batched operations to compute the blocks of fill-in the last four block columns. Additionally, it leaves the final block matrix in the format of (55).

a reduced 4×4 block-tridiagonal matrix is shown in Figure 12. When using the vectorized approach from Figure 6 for the initial levels of CR, this reduced 4×4 matrix will be stored across the four vector lanes of a single batch. This means that we can readily apply PCR to this batch, and in the case of PCG, we will make use of this fact to compute a preconditioner for this matrix using batched operations.

Consider the scenario where the original matrix has been cyclically reduced to a 4×4 block-tridiagonal matrix

$$\mathcal{M}_{(4)} \triangleq \begin{pmatrix} M_0 & K_0^\top & & \\ K_0 & M_1 & K_1^\top & \\ & K_1 & M_2 & K_2^\top \\ & & K_2 & M_3 \end{pmatrix} = \mathcal{L}_{(4)}\mathcal{L}_{(4)}^\top + \begin{pmatrix} 0 & K_0^\top & & \\ K_0 & 0 & K_1^\top & \\ & K_1 & 0 & K_2^\top \\ & & K_2 & 0 \end{pmatrix} \triangleq \mathcal{L}_{(4)}\mathcal{L}_{(4)}^\top + \mathcal{K}_{(4)}, \tag{55}$$

where $\mathcal{L}_{(4)} \triangleq \mathbf{blkdiag}\big(\mathrm{chol}(M_0), \mathrm{chol}(M_1), \mathrm{chol}(M_2), \mathrm{chol}(M_3)\big)$.

The inverse of the symmetric stair preconditioner is then given by the following expression: [3, Eq. 38]

$$\Phi_{(4)}^{-1} = \begin{pmatrix} M_0^{-1} & -M_0^{-1}K_0^\top M_1^{-1} & & \\ -M_1^{-1}K_0M_0^{-1} & M_1^{-1} & -M_1^{-1}K_1^\top M_2^{-1} & \\ & -M_2^{-1}K_1M_1^{-1} & M_2^{-1} & -M_2^{-1}K_2^\top M_3^{-1} \\ & & -M_3^{-1}K_2M_2^{-1} & M_3^{-1} \end{pmatrix} = \mathcal{L}_{(4)}^{-\top}\mathcal{L}_{(4)}^{-1}\left(\mathrm{I} - \mathcal{K}_{(4)}\mathcal{L}_{(4)}^{-\top}\mathcal{L}_{(4)}^{-1}\right). \tag{56}$$

The application of $\mathcal{M}_{(4)}$ to a vector requires multiplication by $\mathcal{K}_{(4)}$ (as defined in (55)) and batched triangular matrix–vector multiplication by $\mathcal{L}_{(4)}$, whereas application of the preconditioner $\Phi_{(4)}^{-1}$ requires multiplication by $\mathcal{K}_{(4)}$ and batched triangular matrix–vector forward and back substitutions. Thanks to the block-diagonal structure, factorization, multiplication and substitution of $\mathcal{L}_{(4)}$ can be fully vectorized using the same batched linear algebra routines as above. Multiplication by $\mathcal{K}_{(4)}$ involves some lane-wise rotations, but it can also be vectorized effectively.

An alternative choice is the block-Jacobi preconditioner $\tilde{\Phi}_{(4)}^{-1} = \mathcal{L}_{(4)}^{-\top}\mathcal{L}_{(4)}^{-1}$. It only takes into account the diagonal blocks of $\mathcal{M}_{(4)}$, and is therefore cheaper to evaluate. It is particularly effective if the entries of $\mathcal{K}_{(4)}$ are sufficiently small (when $\mathcal{M}_{(4)}$ is highly diagonally dominant).

Note that it is possible to combine CR, PCR and PCG in a hybrid solver, switching between algorithms at different levels in the recursion. For the sake of simplicity, we restrict ourselves to CR+PCR and CR+PCG where the switchover happens when the number of remaining blocks equals the vector length.

8 Benchmark results

To evaluate the performance of the optimized implementation of CYQLONE and CYQPALM from [34], we apply CYQPALM to the commonly used linear mass–spring benchmark described in [46, 47, 7], and compare the solver run times to the state-of-the-art HPIPM solver for optimal control problems [11].

8.1 Problem description

The state vector $x \in \mathbb{R}^{2M}$ consists of the positions and velocities of M masses (with mass m) connected by springs (with spring constant k). The leftmost and rightmost masses are connected to fixed walls (one at the origin, and one at distance w). Following [46], actuators are attached between pairs of masses. For $M > 6$, we simply repeat this configuration for $n_u = M/2$ actuators (as described by the matrix W below). The displacements of the masses from their equilibrium positions are bounded in magnitude by 4 m, and the controls cannot exceed 0.5 N.

$$
\begin{aligned}
A_c &= \begin{pmatrix} 0 & \mathrm{I}_M \\ V & -\mu \mathrm{I}_M \end{pmatrix}, & B_c &= \begin{pmatrix} \mathrm{O}_{M \times M/2} \\ \mathrm{blkdiag}(W, \ldots, W) \end{pmatrix}, & b_c &= \frac{1}{m} \begin{pmatrix} 0 \\ \vdots \\ kw \end{pmatrix} \\
V &= \frac{1}{m} \begin{pmatrix} -2k & k & & & & \\ k & -2k & k & & & \\ & k & -2k & k & & \\ & & \ddots & \ddots & \ddots & \\ & & & k & -2k \end{pmatrix}, & W &= \frac{1}{m} \begin{pmatrix} 1 & & \\ -1 & & \\ & 1 & \\ & & 1 \\ & -1 & \\ & & -1 \end{pmatrix} \\
C_c &= \begin{pmatrix} \mathrm{I}_M & \mathrm{O}_{M \times M} \\ \mathrm{O}_{n_u \times M} & \mathrm{O}_{n_u \times M} \end{pmatrix}, & D_c &= \begin{pmatrix} \mathrm{O}_{M \times n_u} \\ \mathrm{I}_{n_u} \end{pmatrix}
\end{aligned}
\tag{57}
$$

We discretize the continuous-time model using zero-order hold (ZOH) to obtain the discrete-time matrices A, B, b, C, D. The sampling interval is set to $T_s = 15\,\mathrm{s}/N$ (so $T_s = 0.5\,\mathrm{s}$ for the reference horizon $N = 30$ from [46]). For the experiments below, T_s is reduced proportionally as N increases to preserve the original time scale. The value of w in [46] and [7] is set to zero, so all masses are pulled to the origin. The constants k and m are set to one, with no friction ($\mu = 0$). The quadratic MPC cost drives the system to steady state, with $Q = Q_N = \mathrm{I}$ and $R = \mathrm{I}$, that is $\ell(x, u) = \frac{1}{2}(x - x_\infty)^\top Q(x - x_\infty) + \frac{1}{2}u^\top Ru$. The purpose of this problem is to gauge the performance of the QP solver, and we do not consider recursive feasibility.

To generate a set of benchmark problems, we vary the number of masses $M \in \{6, 12, 18, 24, 30\}$ and the horizon length $N \in \{32, 64, 96, 128, 192, 256\}$. For each combination of M and N, we randomly generate 150 initial states with velocity zero and positions whose deviations from the steady state are uniformly distributed in $[-3, 3]$. Source code for the benchmarks is available in the CYQLONE GitHub repository [34].

8.2 Benchmark results – CYQPALM

The solver run times for CYQPALM (QPALM using CYQLONE as an inner solver) are compared to those of HPIPM with the *speed* profile in Figures 13 and 14. One set of experiments measures the run times when the solvers are initialized using all zeros, and another measures the run times when the solvers are initialized with the shifted solution of a previous problem. Specifically, we solve a single optimal control problem (excluded from the measurements) and apply the first controls from the solution to the system, as in MPC. The resulting state is used to construct a second OCP, and we measure the solver run time for this second problem, using the solution to the first problem as an initial guess, after shifting it over by one time step (repeating the variables for the last stage).

Experiments are carried out on an Intel Core i7-11700 processor at 2.5 GHz (without frequency scaling to reduce noise in the timing measurements), using all eight processors and a vector length of four. [12]

Figure 13 shows how the solver run times scale for various horizon lengths N, and Figure 14 shows how the solver run times scale for various numbers of masses M. For the longest horizon length $N = 256$ and with

[12] Even though the processor supports a maximum vector length of eight, it lacks a dedicated functional unit for 512-bit floating-point operations. Instead, two 256-bit units are combined to carry out 512-bit operations [8, §12.9], resulting in a similar throughput for vector lengths four and eight. Since the working set for vector length eight is larger, using a vector length of four is preferable.

$M = 12$ masses, the worst-case run time of CyQPALM is over five times faster than the worst-case run time of HPIPM. This allows the sampling time to be reduced considerably for the same compute budget in real-time applications such as MPC. When warm-starting the solvers, CyQPALM is over ten times faster in terms of worst-case run time, and the speedup observed for the average run times is even larger. The speedup factors for other combinations of M and N are listed in Tables 1 and 2, without and with warm starting, respectively. The first value in each cell is the average of the quotients of the total solver run times for all problem instances, and the second value is the quotient of the maximum (worst-case) run time of all problem instances for each solver. [13]

Table 1. Average speedup of the run time and (speedup of the worst-case run time) for CyQPALM (cold) compared to HPIPM (cold).

$M \setminus N$	32	64	96	128	160	192	224	256
6	1.4 (1.4)	2.6 (2.6)	3.6 (3.2)	4.4 (4.0)	5.2 (4.5)	5.8 (5.3)	6.3 (5.6)	6.7 (5.6)
12	1.8 (2.0)	3.2 (3.3)	4.2 (3.9)	4.8 (4.5)	5.3 (5.1)	5.7 (5.0)	6.0 (5.7)	6.1 (5.4)
18	2.0 (2.4)	3.3 (3.3)	4.2 (3.7)	4.7 (4.0)	5.1 (4.5)	5.3 (5.4)	5.3 (4.7)	5.2 (5.0)
24	1.9 (1.9)	3.1 (3.1)	3.6 (3.6)	3.9 (4.0)	3.7 (3.3)	3.7 (3.5)	3.8 (3.9)	3.8 (3.5)
30	1.9 (2.0)	2.9 (2.6)	3.1 (2.8)	3.2 (2.6)	3.3 (2.2)	3.4 (2.6)	3.5 (2.5)	3.6 (2.5)

Table 2. Average speedup of the run time and (speedup of the worst-case run time) for CyQPALM (warm) compared to HPIPM (warm).

$M \setminus N$	32	64	96	128	160	192	224	256
6	4.0 (2.4)	8.0 (5.1)	11.4 (6.3)	14.7 (7.5)	16.9 (8.3)	18.8 (10.1)	20.6 (12.0)	22.2 (11.8)
12	5.1 (3.4)	9.4 (5.1)	12.5 (8.2)	14.1 (7.9)	15.7 (9.3)	17.0 (9.4)	17.6 (10.4)	18.6 (11.8)
18	5.4 (3.6)	10.1 (6.0)	11.8 (9.1)	13.4 (9.4)	13.9 (10.4)	14.6 (9.6)	14.5 (10.6)	14.4 (10.4)
24	5.2 (3.8)	9.1 (5.8)	10.4 (7.5)	10.9 (7.6)	10.1 (7.5)	10.2 (8.1)	10.4 (7.5)	10.1 (7.9)
30	5.1 (3.1)	8.2 (5.7)	8.8 (6.1)	8.5 (5.7)	8.8 (6.9)	9.1 (7.0)	9.2 (6.8)	9.2 (6.5)

For larger numbers of masses, as in Figure 14, CyQPALM also significantly outperforms HPIPM, although the speedup is lower than for the case $M = 12$ discussed above. There are multiple contributing factors:

1. The cost of the Cyqlone factorization of the Newton system scales cubically with increasing numbers of states (cf. (28)), so the factorization step dominates the overall run time for large M. In contrast, if M is small, quadratic operations such as matrix–vector products also make up a considerable fraction of the run time. While matrix–vector products can be fully parallelized (with a speedup of close to 8× for $P = 8$ processors), the speedup for the Cyqlone factorization is less than half of the number of processors (cf. Figure 7).
2. Performance of the BLASFEO routines used by HPIPM increases for increasing matrix sizes, whereas the batched linear algebra routines used in Cyqlone already reach close to peak performance for much smaller matrices (cf. Figure 2). BLASFEO therefore starts to catch up for larger M.
3. For $M \geq 20$, batches of matrices of size $n_x \times n_x$ no longer fit in the L1 cache (48 KiB).
4. QPALM appears to require slightly more iterations than HPIPM for larger problems (but thanks to the factorization updates, they are potentially cheaper).

For small numbers of masses (e.g. $M = 6$ with $n_x = 12$, $n_u = 3$), the performance of BLASFEO is relatively low, while Cyqlone's vectorization is highly performant, resulting in an even higher speedup.

For shorter horizons, the speedup also decreases, as predicted by the theoretical operation counts (cf. Figure 7). The effect is magnified by the relatively higher synchronization overhead for short horizons: in this case, the embarrassingly parallel modified Riccati recursion accounts for a smaller fraction of the total run time compared to CR, the latter of which requires more synchronization between threads. Such synchronization steps introduce some latency, and the communication of matrices between threads results in L1 and L2 cache misses, limiting the performance of subsequent operations on these matrices.

[13] That is, the quotient of the maxima, in contrast to the maximum of the quotients. We are interested in the former because it determines the minimum sample time that can be used in real-time control applications.

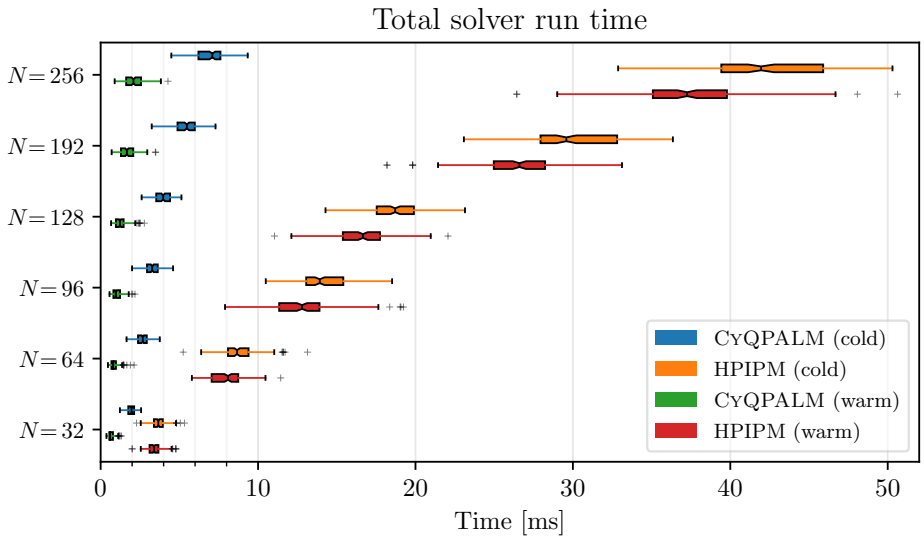

Figure 13. Box plots of the total solver run times for 6×150 instances of the mass–springs benchmark described in Section 8.1, with $M = 12$ masses and varying horizon lengths N. Timings for CyQPALM (with a vector length of four across eight cores) and for HPIPM (using the *speed* profile). We compare the solvers without warm starting (cold) and using the solution of the previous MPC iteration shifted by one time step as initialization (warm).

In conclusion, CyQPALM is significantly faster than HPIPM across the board, with the highest speedup factors being achieved for a small number of masses (where the performance of CyQLONE's batched linear algebra is much higher than that of BLASFEO), and for long horizons (where the cost and the overhead in the CR phase of CyQLONE are small compared to the duration of the Riccati phase).

8.3 Benchmark results – CyQLONE

We also compare the solver run time per iteration between CyQPALM and HPIPM to gauge the performance of the CyQLONE solver, other parallel linear algebra routines, and the parallel line search, independently of the number of operations of the optimization solvers. The left graph of Figure 15 shows that CyQPALM is over seven times faster per iteration than HPIPM. When factorization updates are disabled (as shown in the right graph of Figure 15), the performance of CyQPALM reduces by up to 17%. The speedup factors of the solver run time per iteration are reported in Table 3 (without warm starting).

Finally, we consider the factorization step in isolation: Figure 16 shows detailed execution traces for the CyQLONE factorization (Algorithm 2), and for the factorized Riccati recursion from [12, Alg. 3], implemented using BLASFEO. Thanks to the parallelization and vectorization, CyQLONE is around 3.5 times faster than BLASFEO for this benchmark. Even though the matrices are small, the modified Riccati recursion phase (steps 1 and 2) and the Schur complement construction (step 3) achieve close to the system's peak performance because of the batch-wise vectorization (the performance of the first stages to be processed is slightly lower because the necessary data is not yet in the cache). The linear algebra operations in the CR phase (step 4) achieve lower performance because of the cache misses caused by inter-processor communication (which goes through the slower shared L3 cache).

Table 3. Average speedup of the run time per iteration for CyQPALM (cold) compared to HPIPM (cold).

$M \setminus N$	32	64	96	128	160	192	224	256
6	1.8	3.0	4.0	4.8	5.4	6.0	6.3	6.8
12	2.2	3.6	4.5	5.2	5.7	6.0	6.2	6.4
18	2.4	3.8	4.6	5.1	5.5	5.6	5.5	5.4
24	2.3	3.5	4.1	4.2	4.1	4.0	4.0	4.0
30	2.3	3.4	3.5	3.5	3.6	3.7	3.8	3.8

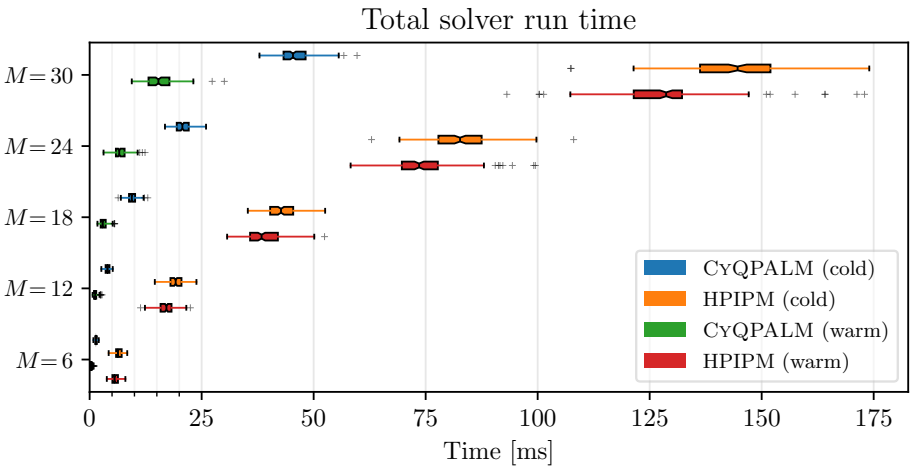

Figure 14. Box plots of the total solver run times for 5×150 instances of the mass–springs benchmark described in Section 8.1, horizon length $N = 128$ and varying numbers of masses M. Timings for CyQPALM (with a vector length of four across eight cores) and for HPIPM (using the *speed* profile). We compare the solvers without warm starting (cold) and using the solution of the previous MPC iteration shifted by one time step as initialization (warm).

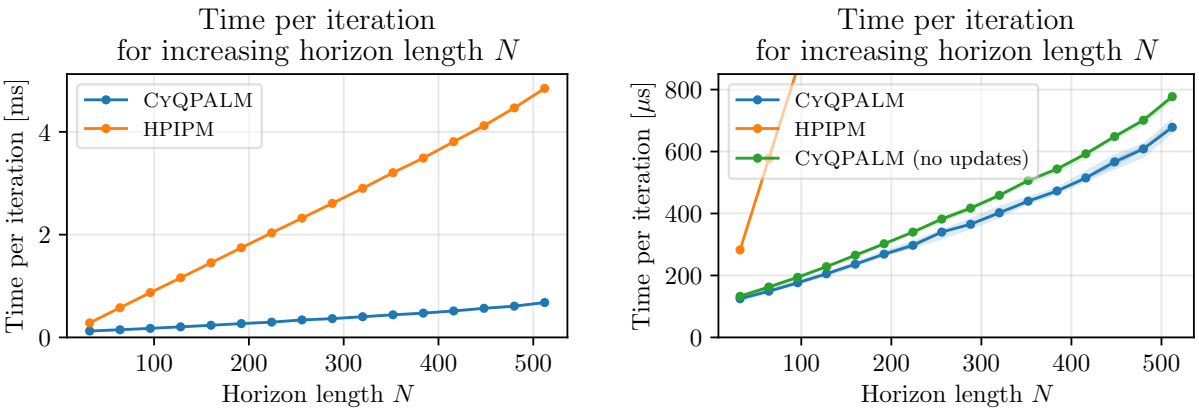

Figure 15. Left: Average run times per iteration of the solvers, for the mass–springs benchmark described in Section 8.1, with $M = 12$ masses and varying horizon lengths N. Timings for CyQPALM (with a vector length of four across eight cores) and for HPIPM (using the *speed* profile), both without warm starting. Right: The same graph, zoomed in and including CyQPALM without factorization updates.

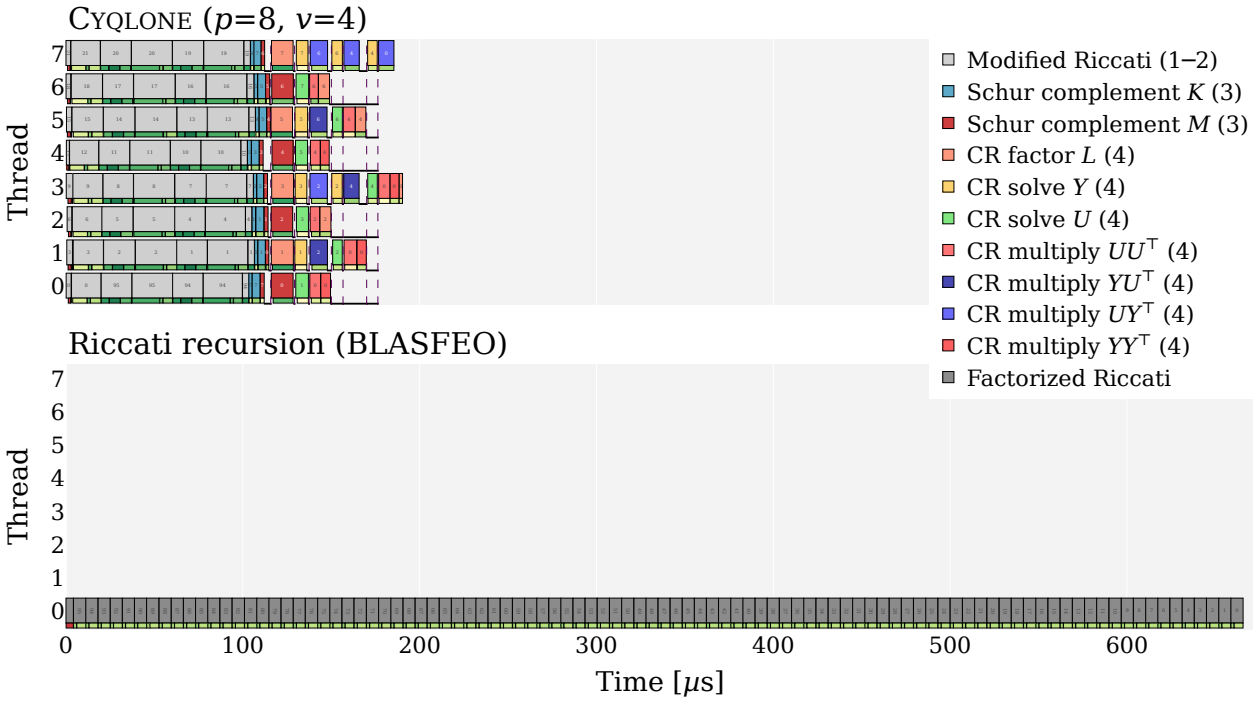

Figure 16. Thread-level execution traces comparing CYQLONE and the serial factorized Riccati recursion implemented using BLASFEO, applied to the KKT system (5) of an OCP with n_x=30, n_u=20 and N=96. Each large rectangle represents a specific step in the algorithm, with the wall time along the horizontal axis, and the thread on which the step is executed along the vertical axis. The index in the center of each rectangle refers to the OCP stage j (the stage corresponding to the first vector lane of the batch in the case of CYQLONE) or the index i in the Schur complement $\mathcal{M}$. Legend labels include the steps of the CYQLONE algorithm: (1–2) for the modified Riccati recursion in Algorithm 1, (3) for the evaluation of the Schur complement, and (4) for its cyclic reduction in Algorithm 2. Colored bars at the bottom of each block indicate the performance of individual linear algebra operations: red for low performance (< 2 GFLOPS), yellow for medium performance (< 10 GFLOPS), and green for high performance (with the darkest green reaching up to 17.7 GFLOPS). For example, each gray block in the bottom graph corresponds to the application of the Riccati recursion (58) to a single stage, which is done using two BLASFEO operations: TRMM to evaluate $\begin{pmatrix} B_j & A_j \end{pmatrix}^\top L_{j+1}$ (≈ 13 GFLOPS), and fused SYRK+POTRF to update and factorize the cost Hessian (≈ 12 GFLOPS). For CYQLONE, the colors in the CR phase of the algorithm (roughly) match the ones in Figure 6, although some SYRK+POTRF operations are fused. Vertical dashed lines represent barriers for synchronization between threads, and horizontal purple lines indicate wait times for threads that already arrived at the barrier.

Table 4. Comparison of linear solvers for KKT systems with optimal control structure. Optimal values for each column are shown in bold.

Method	Figure	Parallelizable	Structure exploitation	Fill-in (λ^j, λ^j) blocks	CR block size
Pas et al. [35]	17	Partial	No	N	–
Frison et al. [12, Alg. 1]	18	No	Asymmetric Riccati	0	–
Frison et al. [12, Alg. 3]	18	No	**Factorized Riccati**	0	–
Wright [49]	19	**Yes**	Asymmetric Riccati	$\boldsymbol{P}$	$2n_x$
Nicholson et al. [30, §4.2]	20	**Yes**	No	N	$2n_x + n_u$
Jallet et al. [22]	21	Partial	Generalized Riccati	$\boldsymbol{P}$	–
Frasch et al. [9, §6.1]	22	**Yes**	No	N	$\boldsymbol{n_x}$
CYQLONE	23	**Yes**	**Factorized Riccati**	$\boldsymbol{P}$	$\boldsymbol{n_x}$

9 Comparison of solvers for KKT systems with OCP structure

This section explores several possible solution methods for linear systems of the form (5), using the matrix visualization introduced in Figure 5 and the corresponding elimination tree as graphical tools for comparing the computational cost (by counting fill-in) and parallelizability (using the elimination tree) of the various methods. Table 4 lists the methods discussed in the remainder of this section, with some important properties that affect their performance.

The coefficient matrix corresponding to (5) is often given in the standard KKT form as in Figure 17, with the cost Hessian in the top left, and constraint Jacobians in the bottom left and top right corners. The following subsections relate existing methods for solving the linear system (5) to the block Cholesky factorization of different permutations of this matrix, based on the order in which the different variables x^j, u^j and λ^j are eliminated. Different orderings lead to various degrees of parallelism and different amounts of fill-in, which may preserve or destroy the optimal control structure of the remaining systems. In the last subsection, we discuss how the proposed CYQLONE method improves upon these existing methods.

9.1 Schur complement method

The Schur complement method used in [26, 35] corresponds to performing the block Cholesky factorization of the matrix in Figure 17. In the process, some fill-in is generated: elements that were structurally zero in the original sparse matrix are nonzero in its Cholesky factor. As discussed in Section 3.1, this loss of sparsity results in a higher number of operations that need to be performed during the factorization and subsequent forward and back substitutions, lowering the efficiency of the overall method.

The block diagonal structure of the cost Hessian allows the different blocks to be eliminated in parallel, which can also be seen in the corresponding elimination tree. Such parallelization techniques are used in [35]. Despite the parallelism available in the factorization of the cost Hessian, the generated fill-in is a block-tridiagonal matrix with N blocks of size $n_x \times n_x$, and the height of the elimination tree scales linearly with the horizon length N, limiting parallel scalability (cf. Section 3.2).

9.2 Riccati recursion

In the case where $E_j = \mathrm{I}$, the Riccati recursion can be used to solve the Newton system without any fill-in. This method is usually derived using backward dynamic programming, resulting in the following recursive expression of the so-called cost-to-go matrix P_k [38, §8.8.3]:

$$\begin{aligned} P_N &= Q_N \\ P_k &= Q_k + A_k^\top P_{k+1} A_k - \left(S_k^\top + A_k^\top P_{k+1} B_k\right) \left(R_k + B_k^\top P_{k+1} B_k\right)^{-1} \left(S_k + B_k^\top P_{k+1} A_k\right). \end{aligned} \tag{58}$$

A straightforward implementation of (58) as in [12, Alg. 1] involves asymmetric products (e.g. $P_{k+1}A_k$), which is suboptimal. Instead, practical implementations often use a factorized variant that exploits the symmetry of $P_k = L_k L_k^\top$ [12, Alg. 3], requiring fewer floating-point operations. Such factorized implementations are specific instances of the block Cholesky factorization of the block-tridiagonal permutation of the

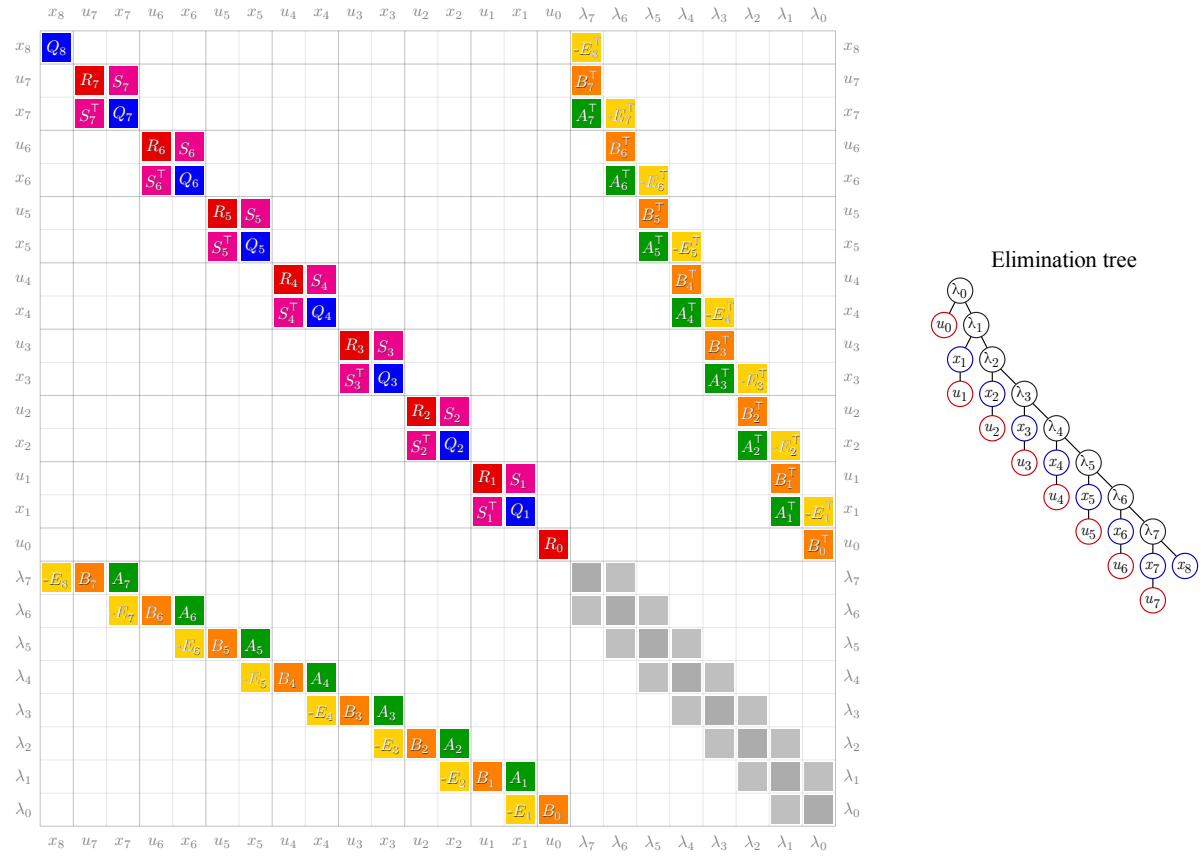

Figure 17. Traditional coefficient matrix corresponding to the optimality conditions (5) of a linear-quadratic OCP (3) with horizon $N = 8$. Fill-in incurred during the block Cholesky factorization is shown in gray.

coefficient matrix shown in Figure 18. The Riccati recursion can easily be related to this block Cholesky factorization by observing that P_k in (58) is the Schur complement of the top-left 3×3 block in

$$\left(\begin{array}{ccc|c} P_{k+1} & -\mathrm{I} & & \\ -\mathrm{I} & 0 & B_k & A_k \\ & B_k^\top & R_k & S_k \\ \hline & A_k^\top & S_k^\top & Q_k \end{array}\right). \tag{59}$$

9.3 Permutations that expose parallelism

The downside of the Riccati recursion is that it is an inherently serial method: Equations are eliminated sequentially, with no room for parallelization. This is evidenced by the shape of the elimination tree from Section 9.2, which is a single linear path. Since the Riccati recursion is a specific instance of the block Cholesky factorization of a block-tridiagonal system (cf. Section 3.2), a natural question that arises is whether cyclic reduction-like techniques can also be used to build parallel variants of the Riccati recursion. The answer turns out to be positive. However, part of the structure is lost in the process. In the following sections, we summarize some different ways of applying recursive CR-like approaches to the Riccati recursion.

9.3.1 Partitioned dynamic programming

In the case of the Riccati recursion, *backward dynamic programming* (DP) is used to summarize all contributions of later stages using a single parametric problem described by the cost-to-go matrix P_j, receding recursively using (58), one stage at a time. After P_1 has been determined using such a backward sweep, Δu^0 can be computed, which is then propagated forward in time using the system dynamics. Both operations are fundamentally sequential. In an attempt to introduce some parallelism into the problem, and inspired by cyclic reduction, Wright [49] proposed the method of *partitioned dynamic programming* (PDP), which considers *multiple* parametric subproblems on different sub-intervals of the horizon rather than a

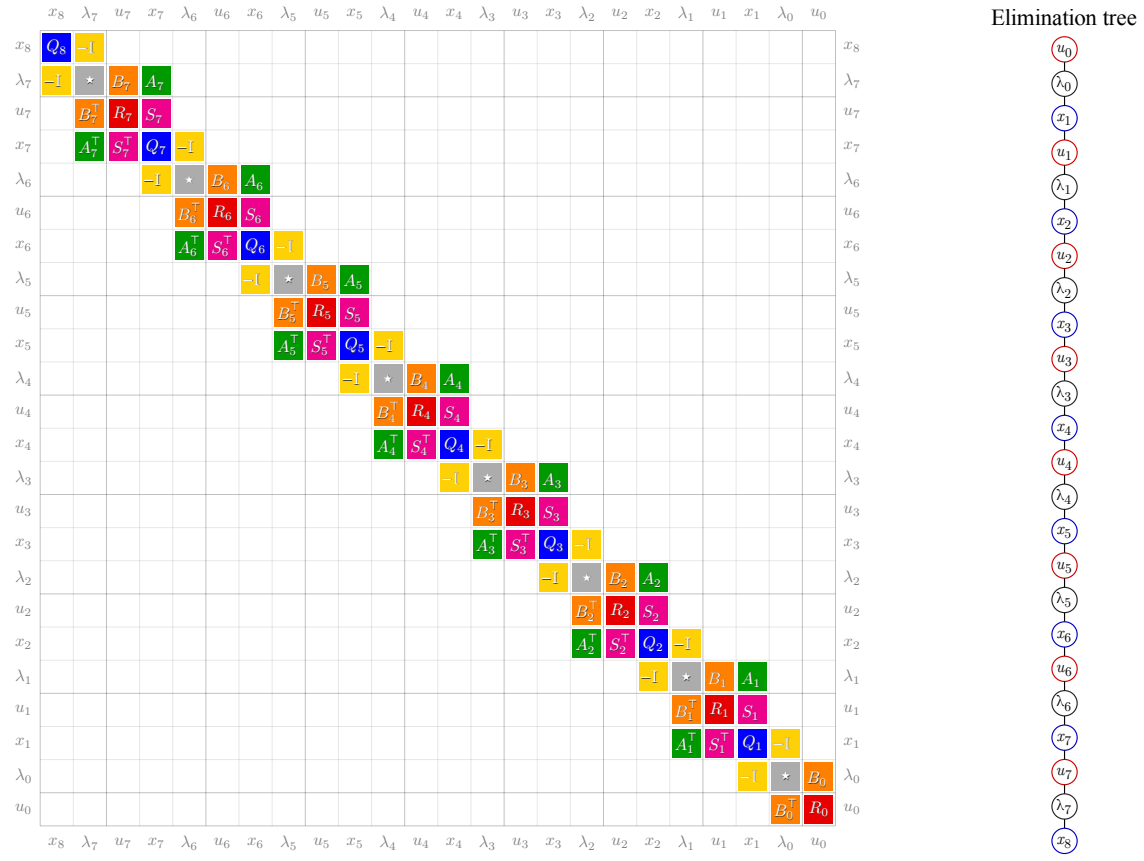

Figure 18. Permutation of Figure 17 corresponding to the Riccati recursion. Gray blocks marked by a star ($\star$) represent special redundant fill-in as described in Section 4.2.

single parametric subproblem that models the entire tail of the horizon as in backward DP. The advantage of this method is that the different sub-intervals can be eliminated in parallel. The downside is that these parametric subproblems introduce an additional term involving the Lagrange multipliers into the dynamics equations of the reduced problem (i.e. fill-in in the (λ_j, λ_j) blocks of the matrix), thus destroying the KKT structure (the bottom-right block of Figure 17 becomes nonzero) [49, (37)]. From the second level onwards, Wright therefore uses a more general method to handle this additional term [49, §3.3], whereas the first level is solved more efficiently using a method based on the Riccati recursion (using the asymmetric variant).

The full method consists of three steps: 1. Use partitioned dynamic programming to eliminate all internal variables of the stages in the selected sub-intervals (e.g. all odd stages), keeping only the variables of the so-called separator stages (e.g. all even stages) [49, §3.1]; 2. Eliminate all controls in the separator stages [49, (15)–(16)]; 3. Solve the remaining block-tridiagonal system of the form [49, (37)] using CR [49, §3.2]. This corresponds to the block Cholesky factorization of the matrix in Figure 19, which clearly shows the three steps of the method. [14]

A related parallel method that also uses parametric subproblems on different sub-intervals of the horizon is described by Nielsen and Axehill in [32] and [31, §6.1]. Their approach is based on a recursive variant of partial condensing, combined with the elimination of redundant controls in the condensed problems: The rank of the condensed transfer matrix $\begin{pmatrix} A_n \cdots A_1 B_0 & A_n \cdots A_2 B_1 & \dots & A_n B_{n-1} & B_n \end{pmatrix}$ is at most the number of states n_x, and therefore, the contributions of all controls of a condensed subproblem can be summarized using a single reduced control vector of dimension at most n_x. A *master problem* is then formed that couples the initial states of all intervals using these reduced controls. The derivation in [31, §6.1.2] makes use of the singular value decomposition of the condensed transfer matrix to eliminate the components of the controls that lie in the null space of the condensed transfer matrix, while [31, Alg. 13] uses a Schur complement to eliminate redundant controls in the subproblems, which can again be viewed as a block Cholesky factorization.

[14] It should be noted that the blocks $\hat{J}_i$ in [49, Alg. PRI] are not necessarily invertible, even when the full blocks $\begin{pmatrix} \hat{J}_i & -\mathrm{I} \\ -I & \hat{Q}_i \end{pmatrix}$ are, so this needs to be considered as a single 2×2 block to prevent the Cholesky factorization from breaking down.

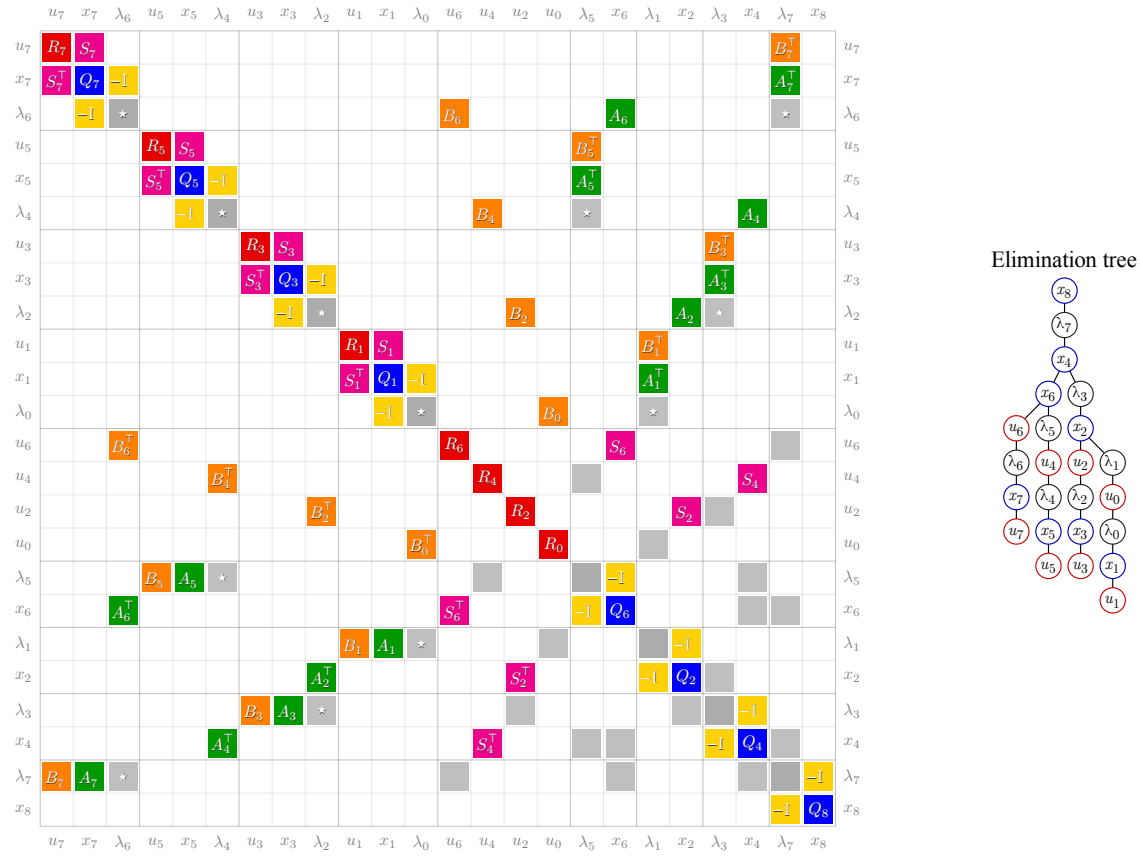

Figure 19. Permutation of Figure 17 corresponding to partitioned dynamic programming followed by elimination of the separator controls and cyclic reduction of the resulting system, as described by [49, §3.3].

An important difference between the proposed CYQLONE method from Section 4 and the methods from [49] and [31] is that CYQLONE eliminates all states and controls before applying cyclic reduction, whereas [49, (42)] keeps the separator states in the reduced problems, and [31, (6.37)] keeps both separator states and controls in the master problem. The reason for keeping some states and controls is to obtain reduced problems with a structure similar to the original problem. However, since the CR phase of these algorithms is the main bottleneck if the number of processors is large, it is advantageous to reduce the size of the blocks that need to be processed using CR by eliminating the states and controls early on, as in CYQLONE. The effect of the block sizes during the CR phase can be observed in the elimination trees of Figures 19 and 23. Table 4 compares the block sizes used during the cyclic reduction phase of different methods.

9.3.2 Direct cyclic reduction

In [30, §4.2], the authors write the linear system (5) as a block-tridiagonal system with diagonal blocks of the form $\begin{pmatrix} Q_j & S_j^\top & A_j^\top \\ S_j & R_j & B_j^\top \\ A_j & B_j & 0 \end{pmatrix}$ and subdiagonal blocks $\begin{pmatrix} 0 & 0 & -\mathrm{I} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$. They then apply cyclic reduction to this system, which is equivalent to block Cholesky factorization of the matrix in Figure 20.

This turns out to be suboptimal, however: the transposes of the subdiagonal blocks have a nonzero sub-block in the first column, which results in two sub-blocks of fill-in per block. An alternative permutation with diagonal blocks of the form $\begin{pmatrix} R_j & S_j & B_j^\top \\ S_j^\top & Q_j & A_j^\top \\ B_j & A_j & 0 \end{pmatrix}$ and subdiagonal blocks $\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -\mathrm{I} \\ 0 & 0 & 0 \end{pmatrix}$ reduces the amount of fill-in.

Regardless, this approach has three main disadvantages: 1. It does not exploit the optimal control structure at the first level (general fill-in is incurred in all (λ^j, λ^j) blocks, unlike in Figures 18 and 19); 2. CR is applied to large block matrices of size $2n_x + n_u$; and 3. The number of nonzeros in the subdiagonal blocks during CR is not uniform, leading to workload imbalance if distributed across different processors.

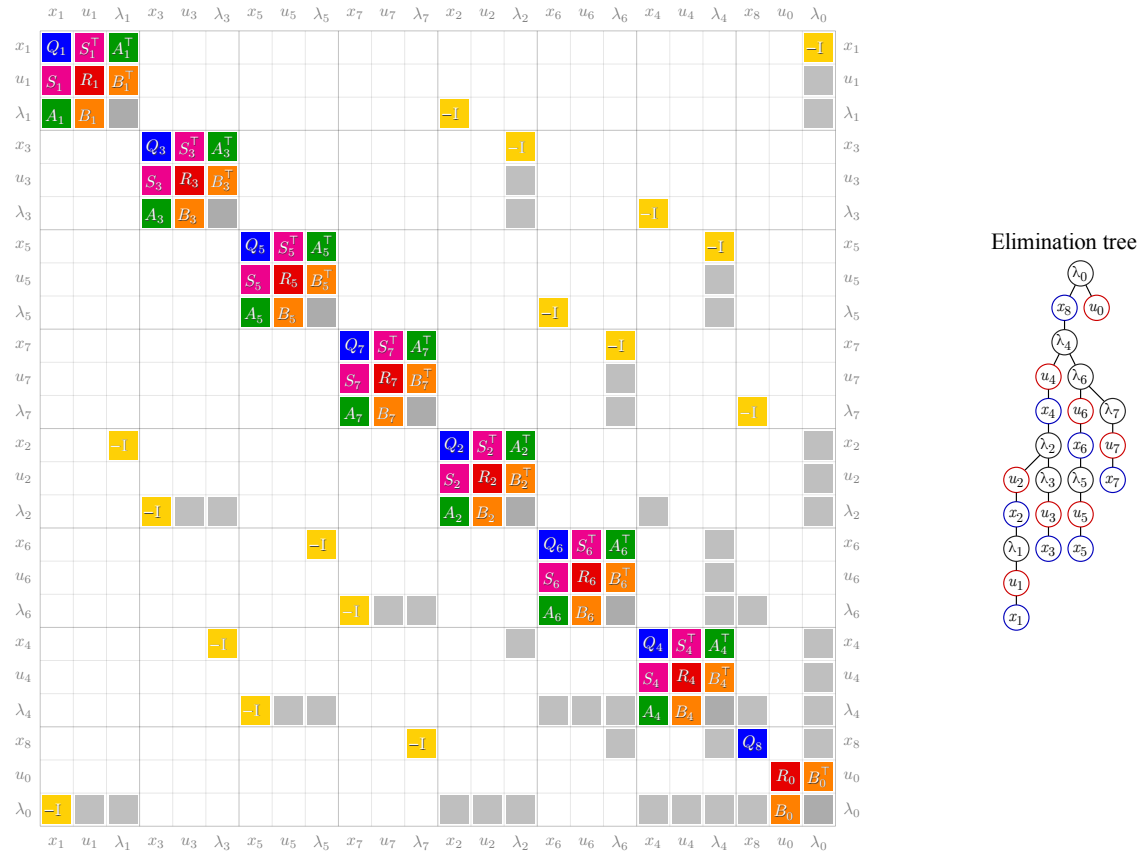

Figure 20. Permutation of Figure 17 corresponding to CR of the block-tridiagonal matrix from [30, (20)–(21)].

9.3.3 Parallel Riccati-based elimination of all states and controls

In [22], the authors partition the horizon into P intervals which they first eliminate in parallel. While the approach is inspired by the partitioning techniques from [49, 31] described in Section 9.3.1, the method in [22] differs from these methods in that it fully eliminates all state and control variables. The elimination is done using a parametric optimization problem with optimal control structure on each interval. This parametric optimization-based derivation is different from the block Cholesky approach used in this paper, and [22, Alg. 2 & 4] differ from Algorithm 1, but the resulting Schur complement matrices are equivalent up to a permutation, since both methods eliminate the same variables during the first phase of the algorithm. The *generalized Riccati* method in [22, Alg. 1] is a generalization of an asymmetric Riccati-based method similar to [12, Alg. 1], whereas CYQLONE's Algorithm 1 can be related to the more efficient factorized variant [12, Alg. 3]. After eliminating all primal variables and the Lagrange multipliers for the coupling between intervals, [22, §V.C] uses a block variant of the Thomas algorithm. This is a sequential algorithm, and the cost of factorizing the Schur complement (consisting of P blocks of size $n_x \times n_x$) therefore scales linearly with the number of processors P, which makes their method less suitable for machines with a large degree of parallelism. In contrast, the cost of the cyclic reduction-based approach used in CYQLONE scales logarithmically with the number of processors, and is therefore preferable for $P > 2$.

For $P = N$, the parallel Riccati-based elimination of all states and controls from [22] reduces to the standard Schur complement method from Section 9.1. Figure 21 relates the method to the block Cholesky factorization of a permuted matrix for the case $N = 2P = 8$.

9.3.4 Cyclic reduction of the Schur complement

The Schur complement method from Section 9.1 offers great parallelizability during the factorization of the cost Hessian. However, its block-tridiagonal Schur complement presents a serial bottleneck. Figure 22 shows how this can be remedied by applying cyclic reduction to the Schur complement.

This approach reveals an excellent level of parallelism, with in the shortest elimination tree so far, thanks to the early elimination of all states and controls before applying CR (cf. Sections 9.3.1 and 9.3.2).

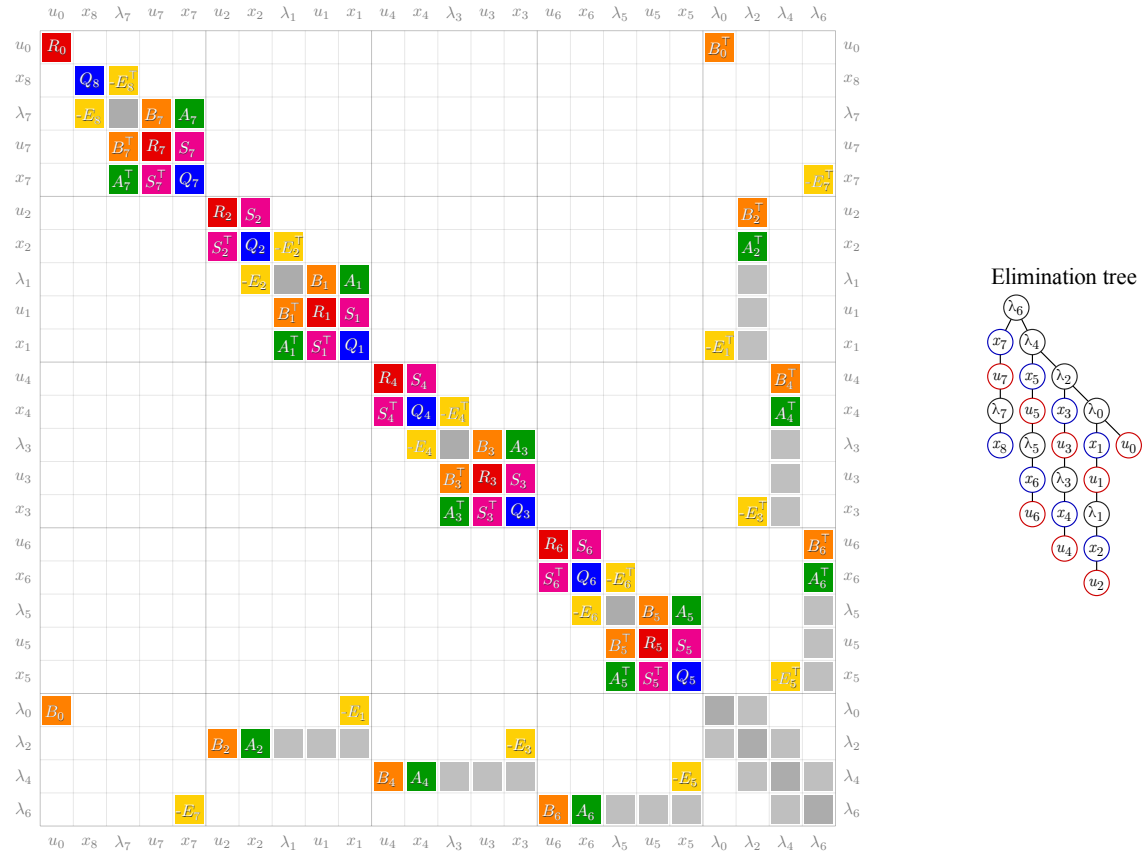

Figure 21. Variant of Figure 17 that eliminates four intervals of two stages in parallel, and then solves the block-tridiagonal Schur complement, similar to what the method from [22] would give rise to if it made use of the Cholesky factorization.

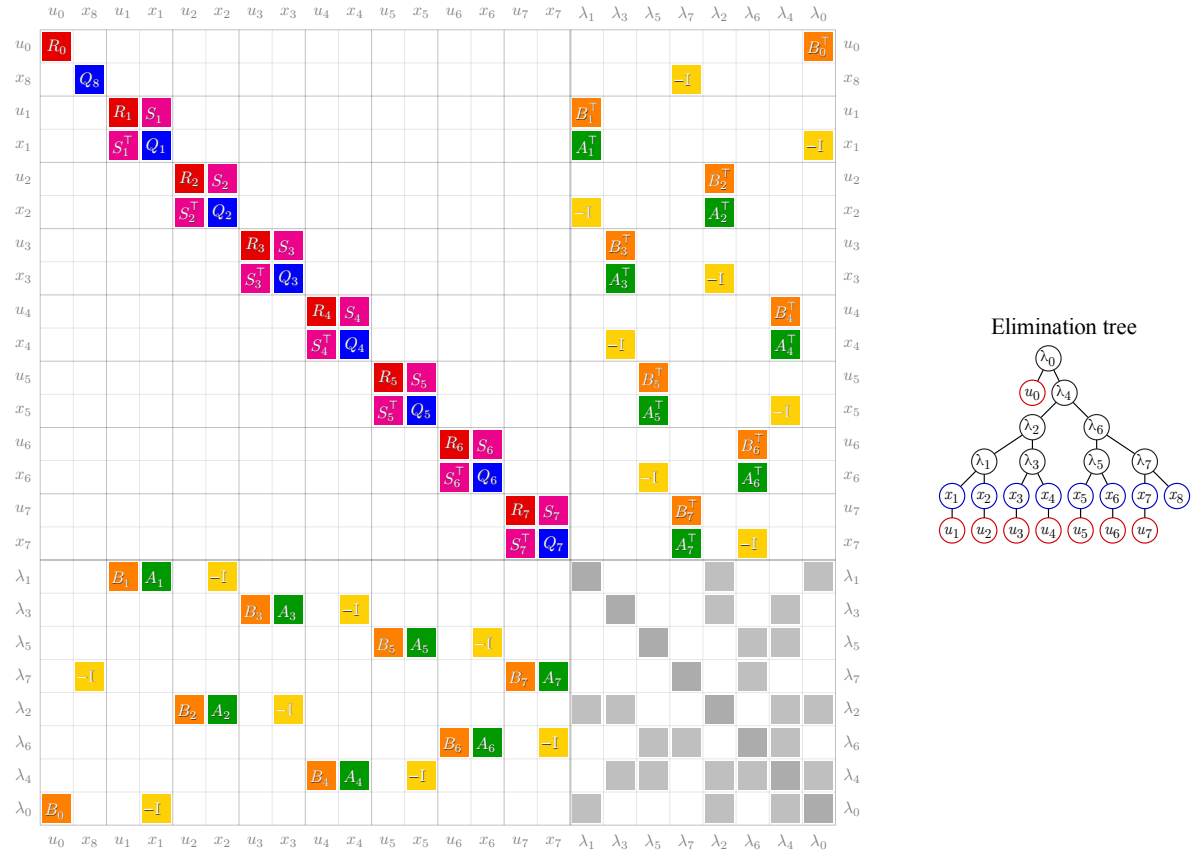

Figure 22. Variant of Figure 17 that uses cyclic reduction for the factorization of the Schur complement. Equivalent to CYQLONE for $P = 8$.

The qpDUNES solver [9, §6.1] uses cyclic reduction of the block-tridiagonal semismooth Newton system that arises from the dual of the original OCP, which is equivalent to applying CR to the Schur complement as described above.

In the special case where the number of processors equals the horizon length ($N = P$), the proposed Cyqlone method becomes equivalent to cyclic reduction of the Schur complement. In this case, each interval in the partitioning from Section 4.1 has length $n = N/P = 1$, so the propagation of the cost-to-go and the dynamics in Algorithm 1 is skipped.

If the number of available processors is smaller than the horizon length ($P < N$), using the Schur complement method is suboptimal. For example, in the extreme case where $P = 1$, it is well known that the Riccati recursion requires fewer FLOPs than the Schur complement method. One reason is that the Schur complement method does not fully exploit the structure of the constraints: By eliminating the variables x^j and x^{j+1} simultaneously, the method suffers from fill-in in the (λ^j, λ^j) block. In contrast, using the Riccati recursion to eliminate x^{j+1} and λ^j before eliminating x^j results in redundant fill-in as discussed in Section 4.2, obviating the need for explicit factorization of the (λ^j, λ^j) block.

This redundant fill-in property of the Riccati recursion is quite fragile: permuting any two rows and columns in Figure 18 either results in additional fill-in in one of the (λ^j, λ^j) blocks, or causes the Cholesky factorization to break down due to zero pivots. As a general rule, permuting the system to expose parallelism introduces additional fill-in. Therefore, by exposing more parallelism than the hardware can exploit, we incur the cost of fill-in that cannot be hidden by further parallelizing any computations. Using these insights, the key to an efficient parallel method for solving KKT systems with optimal control structure lies in:

1. Eliminating the states and controls of at least P independent stages simultaneously to maximize utilization of the parallel hardware;
2. Eliminating no more than P independent stages simultaneously to minimize additional fill-in;
3. Using the Riccati recursion where possible to exploit redundant fill-in;
4. Using cyclic reduction to factorize the Schur complement after eliminating all states and controls.

If the number of (λ^j, λ^j) blocks that receives fill-in is less than P, the first item is violated: this means that fewer than P states are eliminated simultaneously, with some processors being idle. If it is greater than P, the second item is violated: there exist permutations with the same degree of parallelism that incur less fill-in. Table 4 contains the values corresponding to different methods: no fill-in for the serial Riccati recursion (suboptimal utilization of parallel hardware), and fill-in in all N of the (λ^j, λ^j) blocks for Schur complement methods (higher computational cost). Three methods partition the horizon into exactly P intervals which are then eliminated using some variant of the Riccati recursion. This results in exactly P blocks of fill-in in the (λ^j, λ^j) blocks, which is optimal.

9.3.5 Cyqlone

We conclude this section by highlighting the advantages of the proposed Cyqlone method (shown in Figure 23) compared to the existing methods described above:

- If $P = N$, Cyqlone is equivalent to a parallel Schur complement method where the block-tridiagonal Schur complement is solved using cyclic reduction, with a total run time proportional to $\log_2 N$. This method has the shortest elimination tree of all methods considered here, making it the fastest (if sufficient parallelism is available).
- If $P < N$, sub-intervals of the original horizon are eliminated in parallel using a modified Riccati recursion that fully exploits the optimal control structure (with redundant fill-in).
- This elimination procedure makes use of the factorization of the cost-to-go matrix, which requires fewer floating-point operations than the traditional asymmetric variant that uses the recursion (58) directly.
- CR is applied to the block-tridiagonal Schur complement that remains after the parallel sub-interval elimination, with a run time proportional to $\log_2 P$ (in contrast to sequential Thomas-like algorithms).
- The dimension of the blocks used during CR is n_x, as opposed to $2n_x$ or $2n_x + n_u$ in other CR-based methods. This results in fewer floating-point operations in the critical path of the method.
- If $P \ll N$, the embarrassingly parallel sub-interval elimination dominates the total run time, resulting in near-ideal scaling of the run time of approximately $\mathcal{O}(1/P)$ (cf. Figure 7).

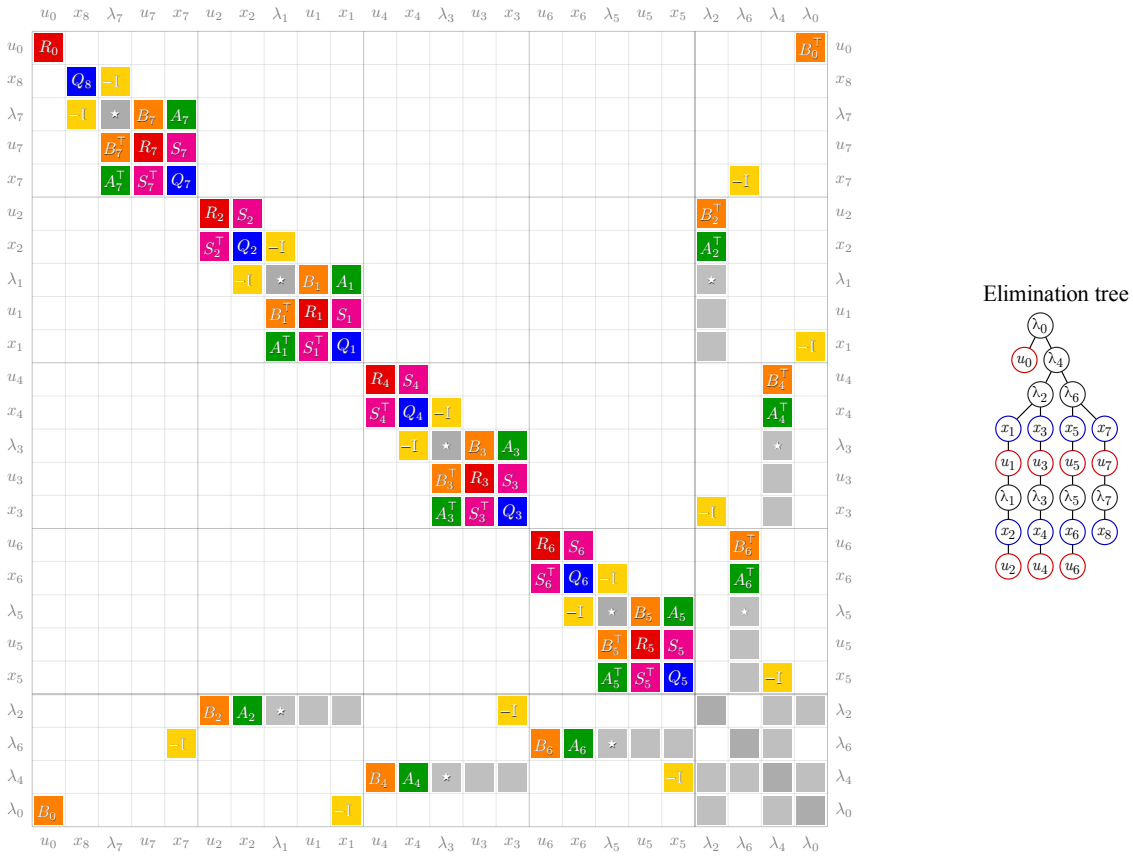

Figure 23. Variant of Figure 17 corresponding to the CYQLONE method from Section 4 with $P = 4$.

A Cyclic reduction

We list the complete CR algorithm [14, 16] for the specific case of *symmetric* block-tridiagonal systems in Algorithm 5. Compare this straightforward implementation to the CYQLONE implementation in Algorithm 2, which includes explicit multi-threading.

There are two important differences between CR and the parallel cyclic reduction (PCR) algorithm listed in Algorithm 6: 1. Each level of PCR processes the same number of matrices ($0 \leq k < N$), whereas the number of active processors in CR halves at each level; and 2. PCR is tail recursive, whereas CR requires an additional back substitution step at all levels. As discussed in Section 7.5.2, this enables vectorization of PCR with full vector lane utilization, and PCR speeds up the solution of the same linear system with different right-hand sides.

Algorithm 5: Solution of a symmetric block-tridiagonal system using cyclic reduction (CR)

Input: N: number of diagonal blocks (power of two)
Input: $M^{(0)} = \{M_k\}_{k=0}^{N-1}$,: diagonal blocks of the block-tridiagonal matrix
Input: $K^{(0)} = \{K_k\}_{k=0}^{N-2}$: subdiagonal blocks of the block-tridiagonal matrix
Input: $b_{(0)} = \{b^k\}_{k=0}^{N-1}$: right-hand side of the linear system
Output: $\{x^k\}_{k=0}^{N-1}$: solution of the linear system
Output: $\{L_k\}_{k=0}^{N-1}, \{U_k\}_{k=1}^{N-1}, \{Y_k\}_{k=1}^{N-1}$: blocks of the Cholesky factor of a permuted matrix (§3.2.2)

1 CYCLIC–REDUCTION($M^{(0)}$, $K^{(0)}$, $b_{(0)}$, 0)
2 **Function** CYCLIC–REDUCTION($M^{(l)}$, $K^{(l)}$, $b_{(l)}$, l)
3 **if** $2^l = N$ ▷ Base case (single block remaining)
4 $L_0 = \mathrm{chol}(M_0^{(l)})$
5 $x^0 = L_0^{-\top} L_0^{-1} b_{(l)}^0$
6 **return**
7 $p = N/2^{l+1}$ ▷ Number of active processors
8 $K_{N-2^l}^{(l)} = 0$
9 **for** $i = 0, 1, \ldots, p-1$ ▷ Factor odd equations (10)
10 $k = 2^{l+1} i + 2^l$
11 $L_k = \mathrm{chol}(M_k^{(l)})$
12 $Y_k = K_k^{(l)} L_k^{-\top}$
13 $U_k = K_{k-2^l}^{(l)\top} L_k^{-\top}$
14 $\tilde{b}^k = L_k^{-1} b_{(l)}^k$
15 $M_0^{(l+1)} = M_0^{(l)} - U_{2^l} U_{2^l}^\top$ ▷ Update even equations (11)
16 $b_{(l+1)}^0 = b_{(l)}^0 - U_{2^l} \tilde{b}_{2^l}$
17 **for** $i = 1, \ldots, p-1$
18 $k = 2^{l+1} i$
19 $M_k^{(l+1)} = M_k^{(l)} - Y_{k-2^l} Y_{k-2^l}^\top - U_{k+2^l} U_{k+2^l}^\top$
20 $K_{k-2^{l+1}}^{(l+1)} = -Y_{k-2^l} U_{k-2^l}^\top$
21 $b_{(l+1)}^k = b_{(l)}^k - Y_{k-2^l} \tilde{b}^{k-2^l} - U_{k+2^l} \tilde{b}^{k+2^l}$
22 CYCLIC–REDUCTION($M^{(l+1)}$, $K^{(l+1)}$, $b_{(l+1)}$, $l+1$) ▷ Factor & solve even equations (recursively)
23 **for** $i = 0, 1, \ldots, p-1$ ▷ Solve odd equations (12)
24 $k = 2^{l+1} i + 2^l$
25 $x^k = L_k^{-\top} \left(\tilde{b}^k - Y_k^\top x^{k+2^l} - U_k^\top x^{k-2^l} \right)$

Algorithm 6: Solution of a symmetric block-tridiagonal system using parallel cyclic reduction (PCR)

Input: N: number of diagonal blocks (power of two)
Input: $M^{(0)} = \{M_k\}_{k=0}^{N-1}$,: diagonal blocks of the block-tridiagonal matrix
Input: $K^{(0)} = \{K_k\}_{k=0}^{N-2}$: subdiagonal blocks of the block-tridiagonal matrix, $K_{N-1}^{(0)} = 0$
Input: $b_{(0)} = \{b^k\}_{k=0}^{N-1}$: right-hand side of the linear system
Output: $\{x^k\}_{k=0}^{N-1}$: solution of the linear system
Output: $\{L_k^{(l)}\}_{k=0}^{N-1}, \{U_k^{(l)}\}_{k=1}^{N-1}, \{Y_k^{(l)}\}_{k=1}^{N-1}$: blocks of the Cholesky factors of permuted matrices

PARALLEL–CYCLIC–REDUCTION($M^{(0)}$, $K^{(0)}$, $b_{(0)}$, 0)
Function PARALLEL–CYCLIC–REDUCTION($M^{(l)}$, $K^{(l)}$, $b_{(l)}, l$)
  $L_k^{(l)} = \mathrm{chol}\left(M_k^{(l)}\right)$ $(0 \le k < N)$
  if $2^l = N$ ▷ Base case (N individual blocks remaining)
    $x^k = L_k^{(l)\,-\top} L_k^{(l)\,-1} b_{(l)}^k$ $(0 \le k < N)$
    return
  $Y_k^{(l)} = K_k^{(l)} L_k^{(l)\,-\top}$ $(0 \le k < N)$
  $U_k^{(l)} = K_{k-2^l}^{(l)\top} L_k^{(l)\,-\top}$ $(0 \le k < N)$
  $\tilde{b}^k = L_k^{(l)\,-1} b_{(l)}^k$ $(0 \le k < N)$
  $M_k^{(l+1)} = M_k^{(l)} - Y_{k-2^l}^{(l)} Y_{k-2^l}^{(l)\top} - U_{k+2^l}^{(l)} U_{k+2^l}^{(l)\top}$ $(0 \le k < N)$
  $K_k^{(l+1)} = -Y_{k+2^l}^{(l)} U_{k+2^l}^{(l)\top}$ $(0 \le k < N)$
  $b_{(l+1)}^k = b_{(l)}^k - Y_{k-2^l}^{(l)} \tilde{b}^{k-2^l} - U_{k+2^l}^{(l)} \tilde{b}^{k+2^l}$ $(0 \le k < N)$
  PARALLEL–CYCLIC–REDUCTION($M^{(l+1)}$, $K^{(l+1)}$, $b_{(l+1)}$, $l+1$) ▷ Reduce all equations recursively
For the sake of readability, subscripts $k \pm 2^l$ are implicitly reduced modulo N.

References

[1] Dario A. Bini, Stefano Massei and Leonardo Robol. "On the decay of the off-diagonal singular values in cyclic reduction". In: *Linear Algebra and its Applications* 519 (Apr. 2017), pp. 27–53. ISSN: 0024-3795. DOI: 10.1016/j.laa.2016.12.027.

[2] Dario A. Bini and Beatrice Meini. "The cyclic reduction algorithm: from Poisson equation to stochastic processes and beyond". In: *Numerical Algorithms* 51.1 (May 2009), pp. 23–60. ISSN: 1572-9265. DOI: 10.1007/s11075-008-9253-0.

[3] Xueyi Bu and Brian Plancher. "Symmetric Stair Preconditioning of Linear Systems for Parallel Trajectory Optimization". In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. May 2024, pp. 9779–9786. DOI: 10.1109/ICRA57147.2024.10610386.

[4] Sandra Catalán, Adrián Castelló, Francisco D. Igual, Rafael Rodríguez-Sánchez and Enrique S. Quintana-Ortí. "Programming parallel dense matrix factorizations with look-ahead and OpenMP". In: *Cluster Computing* 23.1 (Mar. 2020), pp. 359–375. ISSN: 1573-7543. DOI: 10.1007/s10586-019-02927-z.

[5] Cris Cecka and Vijay Thakkar. *CuTe Layouts – NVIDIA/cutlass 3.6.0*. URL: https://github.com/NVIDIA/cutlass/blob/24f991e87930e1159f1f5a47e329d43bcfbd76b9/media/docs/cute/01_layout.md (visited on 08/01/2025).

[6] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms. Society for Industrial and Applied Mathematics, Jan. 2006. ISBN: 978-0-89871-613-9. DOI: 10.1137/1.9780898718881.

[7] Alexander Domahidi, Aldo U. Zgraggen, Melanie N. Zeilinger, Manfred Morari and Colin N. Jones. "Efficient interior point methods for multistage problems arising in receding horizon control". In: *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*. Maui, HI, USA: IEEE, Dec. 2012, pp. 668–674. ISBN: 978-1-4673-2066-5 978-1-4673-2065-8 978-1-4673-2063-4 978-1-4673-2064-1. DOI: 10.1109/CDC.2012.6426855.

[8] Agner Fog. *The microarchitecture of Intel, AMD, and VIA CPUs*. May 2024. URL: https://www.agner.org/optimize/microarchitecture.pdf (visited on 06/08/2024).

[9] Janick V. Frasch, Sebastian Sager and Moritz Diehl. "A parallel quadratic programming method for dynamic optimization problems". In: *Mathematical Programming Computation* 7.3 (Sept. 2015), pp. 289–329. ISSN: 1867-2949, 1867-2957. DOI: 10.1007/s12532-015-0081-7.

[10] Gianluca Frison. "Algorithms and Methods for High-Performance Model Predictive Control". DTU Compute PHD-2015 No. 402. PhD thesis. Technical University of Denmark, 2016.

[11] Gianluca Frison and Moritz Diehl. "HPIPM: a high-performance quadratic programming framework for model predictive control". In: *IFAC-PapersOnLine*. 21st IFAC World Congress 53.2 (Jan. 2020), pp. 6563–6569. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2020.12.073.

[12] Gianluca Frison and John Bagterp Jørgensen. "Efficient implementation of the Riccati recursion for solving linear-quadratic control problems". In: *2013 IEEE International Conference on Control Applications (CCA)*. ISSN: 1085-1992. Aug. 2013, pp. 1117–1122. DOI: 10.1109/CCA.2013.6662901.

[13] Gianluca Frison, Dimitris Kouzoupis, Tommaso Sartor, Andrea Zanelli and Moritz Diehl. "BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization". In: *ACM Transactions on Mathematical Software* 44.4 (Dec. 2018), pp. 1–30. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/3210754.

[14] Walter Gander and Gene H. Golub. "Cyclic reduction-history and applications". In: *Scientific Computing: Proceedings of the Workshop, 10-12 March 1997, Hong Kong*. Springer Science & Business Media, 1998, p. 73.

[15] P. E. Gill, G. H. Golub, W. Murray and M. A. Saunders. "Methods for modifying matrix factorizations". In: *Mathematics of Computation* 28.126 (1974), pp. 505–535. ISSN: 0025-5718, 1088-6842. DOI: 10.1090/S0025-5718-1974-0343558-6.

[16] Don Heller. "Some Aspects of the Cyclic Reduction Algorithm for Block Tridiagonal Linear Systems". In: *SIAM Journal on Numerical Analysis* 13.4 (Sept. 1976). Publisher: Society for Industrial and Applied Mathematics, pp. 484–496. ISSN: 0036-1429. DOI: 10.1137/0713042.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Aug. 2011. ISBN: 978-0-12-383872-8.

[18] Ben Hermans, Andreas Themelis and Panagiotis Patrinos. "QPALM: A Newton-type Proximal Augmented Lagrangian Method for Quadratic Programs". In: *2019 IEEE 58th Conference on Decision and Control (CDC)* (Dec. 2019), pp. 4325–4330. DOI: 10.1109/CDC40024.2019.9030211.

[19] Ben Hermans, Andreas Themelis and Panagiotis Patrinos. "QPALM: a proximal augmented lagrangian method for nonconvex quadratic programs". In: *Mathematical Programming Computation* 14.3 (Sept. 2022), pp. 497–541. ISSN: 1867-2957. DOI: 10.1007/s12532-022-00218-0.

[20] S.P. Hirshman, K.S. Perumalla, V.E. Lynch and R. Sanchez. "BCYCLIC: A parallel block tridiagonal matrix cyclic solver". In: *Journal of Computational Physics* 229.18 (Sept. 2010), pp. 6392–6404. ISSN: 00219991. DOI: 10.1016/j.jcp.2010.04.049.

[21] Hiroshi Inoue. "How SIMD width affects energy efficiency: A case study on sorting". In: *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*. Apr. 2016, pp. 1–3. DOI: 10.1109/CoolChips.2016.7503679.

[22] Wilson Jallet, Ewen Dantec, Etienne Arlaud, Justin Carpentier and Nicolas Mansard. "Parallel and Proximal Constrained Linear-Quadratic Methods for Real-Time Nonlinear MPC". In: *Proceedings of the 2024 Robitics: Science and Systems (RSS) conference*. Delft, July 2024. DOI: 10.48550/arXiv.2405.09197.

[23] David Kershaw. "Solution of Single Tridiagonal Linear Systems and Vectorization of the ICCG Algorithm on the Cray-1". In: *Parallel Computations*. Ed. by Garry Rodrigue. Academic Press, Jan. 1982, pp. 85–99. ISBN: 978-0-12-592101-5. DOI: 10.1016/B978-0-12-592101-5.50008-7.

[24] Arno Krechel, Hans-Joachim Plum and Klaus Stüben. "Parallelization and vectorization aspects of the solution of tridiagonal linear systems". In: *Parallel Computing* 14.1 (May 1990), pp. 31–49. ISSN: 0167-8191. DOI: 10.1016/0167-8191(90)90094-P.

[25] Jules J. Lambiotte and Robert G. Voigt. "The Solution of Tridiagonal Linear Systems on the CDC STAR 100 Computer". In: *ACM Trans. Math. Softw.* 1.4 (Dec. 1975), pp. 308–329. ISSN: 0098-3500. DOI: 10.1145/355656.355658.

[26] Kristoffer Fink Lowenstein, Daniele Bernardini and Panagiotis Patrinos. "QPALM-OCP: A Newton-Type Proximal Augmented Lagrangian Solver Tailored for Quadratic Programs Arising in Model Predictive Control". In: *IEEE Control Systems Letters* (2024), pp. 1349–1354. ISSN: 2475-1456. DOI: 10.1109/LCSYS.2024.3410638.

[27] Conrado Martínez, Markus Nebel and Sebastian Wild. "Sesquickselect: One and a half pivots for cache-efficient selection". In: *2019 Proceedings of the Meeting on Analytic Algorithmics and Combinatorics (ANALCO)*. Society for Industrial and Applied Mathematics, Jan. 2019, pp. 54–66. DOI: 10.1137/1.9781611975505.6.

[28] Olga Moldovanova and Mikhail Kurnosov. "Automatic SIMD Vectorization of Loops: Issues, Energy Efficiency and Performance on Intel Processors". In: *Supercomputing*. Ed. by Vladimir Voevodin and Sergey Sobolev. Cham: Springer International Publishing, 2017, pp. 388–399. ISBN: 978-3-319-71255-0. DOI: 10.1007/978-3-319-71255-0_31.

[29] David R. Musser. "Introspective Sorting and Selection Algorithms". In: *Software: Practice and Experience* 27.8 (1997), pp. 983–993. ISSN: 1097-024X. DOI: 10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-%23.

[30] Bethany L. Nicholson, Wei Wan, Shivakumar Kameswaran and Lorenz T. Biegler. "Parallel cyclic reduction strategies for linear systems that arise in dynamic optimization problems". In: *Computational Optimization and Applications* 70.2 (June 2018), pp. 321–350. ISSN: 1573-2894. DOI: 10.1007/s10589-018-0001-7.

[31] Isak Nielsen. *On Structure Exploiting Numerical Algorithms for Model Predictive Control*. Linköping University Electronic Press, Oct. 2015. ISBN: 978-91-7685-965-0. DOI: 10.3384/lic.diva-121711.

[32] Isak Nielsen and Daniel Axehill. "A parallel structure exploiting factorization algorithm with applications to Model Predictive Control". In: *2015 54th IEEE Conference on Decision and Control (CDC)*. Osaka: IEEE, Dec. 2015, pp. 3932–3938. ISBN: 978-1-4799-7886-1. DOI: 10.1109/CDC.2015.7402830.

[33] Pieter Pas. *BATMAT: Fast linear algebra routines for batches of small matrices*. 2025. URL: https://github.com/tttapa/batmat.

[34] Pieter Pas. *CYQLONE: A parallel quadratic programming solver for optimal control*. 2025. URL: https://github.com/kul-optec/cyqlone.

[35] Pieter Pas, Kristoffer Løwenstein, Daniele Bernardini and Panagiotis Patrinos. "Exploiting Parallelism in a QPALM-based Solver for Optimal Control". In: *RSS 2024 Workshop: Frontiers of optimization for robotics*. Delft, The Netherlands, July 2024.

[36] Pieter Pas and Panagiotis Patrinos. "Blocked Cholesky factorization updates of the Riccati recursion using hyperbolic Householder transformations". In: Accepted for publication in the Proceedings of CDC 2025. arXiv, Mar. 2025. DOI: 10.48550/arXiv.2503.15372.

[37] Pieter Pas, Andreas Themelis and Panagiotis Patrinos. "Gauss–Newton meets PANOC: A fast and globally convergent algorithm for nonlinear optimal control". In: *IFAC-PapersOnLine* 56.2 (2023), pp. 4852–4857. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2023.10.1254.

[38] James Blake Rawlings, David Q. Mayne and Moritz Diehl. *Model predictive control: theory, computation, and design*. 2nd edition. Madison, Wisconsin: Nob Hill Publishing, 2017. ISBN: 978-0-9759377-3-0.

[39] Fabio Reale. "A tridiagonal solver for massively parallel computer systems". In: *Parallel Computing* 16.2 (Dec. 1990), pp. 361–368. ISSN: 0167-8191. DOI: 10.1016/0167-8191(90)90073-I.

[40] Tobias Ribizel and Hartwig Anzt. "Parallel selection on GPUs". In: *Parallel Computing* 91 (Mar. 2020), p. 102588. ISSN: 01678191. DOI: 10.1016/j.parco.2019.102588.

[41] Roland Schwan, Daniel Kuhn and Colin N. Jones. *Exploiting Multistage Optimization Structure in Proximal Solvers*. (Preprint). Mar. 2025. DOI: 10.48550/arXiv.2503.12664.

[42] Christian Siebert. "Scalable and Efficient Parallel Selection". In: *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski and Jerzy Waśniewski. Berlin, Heidelberg: Springer, 2014, pp. 202–213. ISBN: 978-3-642-55224-3. DOI: 10.1007/978-3-642-55224-3_20.

[43] Tyler M. Smith, Robert Van De Geijn, Mikhail Smelyanskiy, Jeff R. Hammond and Field G. Van Zee. "Anatomy of High-Performance Many-Threaded Matrix Multiplication". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. Phoenix, AZ, USA: IEEE, May 2014, pp. 1049–1059. ISBN: 978-1-4799-3800-1 978-1-4799-3799-8. DOI: 10.1109/IPDPS.2014.110.

[44] Field G. Van Zee and Robert A. Van De Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". In: *ACM Transactions on Mathematical Software* 41.3 (June 2015), pp. 1–33. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/2764454.

[45] Lander Vanroye, Ajay Sathya, Joris De Schutter and Wilm Decré. "FATROP: A Fast Constrained Optimal Control Problem Solver for Robot Trajectory Optimization and Control". In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. ISSN: 2153-0866. Oct. 2023, pp. 10036–10043. DOI: 10.1109/IROS55552.2023.10342336.

[46] Yang Wang and Stephen Boyd. "Fast Model Predictive Control Using Online Optimization". In: *IFAC Proceedings Volumes*. 17th IFAC World Congress 41.2 (Jan. 2008), pp. 6974–6979. ISSN: 1474-6670. DOI: 10.3182/20080706-5-KR-1001.01182.

[47] Yang Wang and Stephen Boyd. "Performance bounds for linear stochastic control". In: *Systems & Control Letters* 58.3 (Mar. 2009), pp. 178–182. ISSN: 0167-6911. DOI: 10.1016/j.sysconle.2008.10.004.

[48] Michael Weiss. "Strip mining on SIMD architectures". In: *Proceedings of the 5th international conference on Supercomputing*. ICS '91. New York, NY, USA: Association for Computing Machinery, June 1991, pp. 234–243. ISBN: 978-0-89791-434-5. DOI: 10.1145/109025.109083.

[49] Stephen J. Wright. "Partitioned Dynamic Programming for Optimal Control". In: *SIAM Journal on Optimization* 1.4 (Nov. 1991). Publisher: Society for Industrial and Applied Mathematics, pp. 620–642. ISSN: 1052-6234. DOI: 10.1137/0801037.

[50] Yao Zhang, Jonathan Cohen and John D. Owens. "Fast tridiagonal solvers on the GPU". In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '10. New York, NY, USA: Association for Computing Machinery, Jan. 2010, pp. 127–136. ISBN: 978-1-60558-877-3. DOI: 10.1145/1693453.1693472.