A Parallell Linear System Solver for Optimal Control

Brian E. Jackson
Robotics Institute
Carnegie Mellon University
Pittsburgh, USA
brianjackson@cmu.edu

Zachary Manchester
Robotics Institute
Carnegie Mellon University
Pittsburgh, USA
zacm@cmu.edu

***Abstract*—Efficient linear system solvers are a critical component of numerical methods for solving optimal control problems. Currently, most direct methods for solving linear systems are serial, and only a few commercial parallel direct methods exist. We present a novel direct method that exploits the sparse, banded structure that arises in optimal control problems, including the prototypical LQR problem. Using a recursive application of Schur compliments, the algorithm has a theoretical $\mathcal{O}(\log(N))$ complexity in the time horizon, and maps well onto many-core processors. An open-source implementation demonstrates better performance than state-of-the-art commercial parallel direct solvers designed for general sparse symmetric linear systems.**

I. Motivation

Parallel computing has changed many aspects of how we approach algorithm development. With single-processor speeds reaching asymptotic improvements, it has become increasingly clear that substantial future improvements in computational performance must come through parallelization, custom silicon, or often a combination of both. While many algorithms in robotics are naturally parallelizable, especially the data-driven approaches that frequently appear in perception and reinforcement learning, many algorithms are still inherently serial and difficult to parallelize. Numerous problems in robotics require solving large, sparse, and nonlinearly-constrained optimization problems. Particularly in optimal control, these problems should ideally be solved online and onboard the robot, placing a high demand on computationally efficient algorithms.

While some approaches to motion planning and control are naturally parallelizable, such as sampling-based approaches like RRT [1], graph-based approaches that rely on discretization of the configuration space [2], or trajectory-sampling approaches like MPPI [3], they have many limitations. These algorithms often scale poorly with the dimension of the state space or the time horizon, have limited ability to deal with complicated and/or underactuated dynamics, or are limited by the types of constraints that can be imposed on the system, such as those used to ensure either physically-realistic or safe state and control trajectories. Optimization-based methods, on the other hand, offer many benefits over these approaches, and have received a lot of attention from both the academic and commercial robotics communities over the past few years.

Typical approaches to optimal control and the related subproblem of trajectory optimization usually fall into one of two categories: DDP-based approaches that rely on iteratively generating a local feedback law and simulating the system forward under the closed-loop feedback policy [4]–[10], or direct collocation methods that solve the problem using general-purpose nonlinear programming (NLP) solvers like SNOPT [11], Ipopt [12], or Knitro [13]–[18]. Both of these methods rely on forming a local approximation to the nonlinear system, which requires 1st or 2nd-order Taylor series expansions of the cost, dynamics, and constraints. The calculation of these expansions is trivially parallelizable over the time horizon. Parallelizing the optimization algorithm itself, however, is much more challenging.

To parallelize DDP-based methods, the trajectory is split into many segments and "glued" back together by additional constraints [8], [19], [20]. While this approach shows some promise, a recent study implementing DDP on a GPU found that increased parallelism led to decreased convergence rates, leading to a natural limit to the amount of parallelism that could be exploited [19]. Direct methods, on the other hand, are almost always dependent on general-purpose NLP solvers, all of which are currently serial and single-threaded. The most computationally demanding part of these algorithms is generally solving a large, sparse linear system of equations encoding the local optimality conditions for the nonlinear optimization problem. Efficient methods for parallelizing the solution of these linear systems are critical for unlocking the computational improvements necessary to find trajectories for high-dimensional systems and for long-horizon trajectories.

This paper presents a novel method for solving the LQR problem, whose solution can be found by solving a banded linear system of equations. This linear system is prototypical of those found when solving optimal control problems with methods like sequential quadratic programming (SQP). Using a hierarchical set of Schur compliments inspired by the nested-dissection algorithm used to solve problems for planar graphs [21], [22], we derive an algorithm with $\mathcal{O}(\log(N))$ complexity in the time horizon. With a modest amount of parallelism, this algorithm outperforms the benchmark $\mathcal{O}(N)$ Riccati recursion that underlies DDP-based algorithms, while being strictly more general in its applicability. We also show that the algorithm is faster and scales better with increased parallelism than commercial parallel linear-system solvers. Our algorithm is particularly well-suited to solving long-horizon problems. The ability to efficiently handle long time horizons is often critical in high-speed applications such as autonomous trucking, where a longer planning horizon has a direct impact on the safety and performance of the controller. Even when solving problems of-

fline, such as finding low-thrust trajectories for space systems [23], reducing solve times from hours to minutes or seconds will have a dramatic impact on the approaches we can take when searching for or designing such trajectories.

The remainder of this paper is structured as follows: In Section II we survey methods for solving the LQR problem and other related parallel solvers for optimal control. In Section III we review the linear quadratic regulator (LQR) problem and Schur compliments. We derive our algorithm in Section IV, demonstrating a novel use of recursive Schur compliments to solve the LQR problem, followed by considerations for adapting the algorithm to work well on a many-core processor. The theoretical performance of the algorithm is compared to the Riccati solution in Section V, which is followed up by a comparison of the actual open-source C implementation against a suite of state-of-the-art sparse matrix solvers in Section VI, with concluding remarks in Section VII.

II. Related Work

As one of the canonical problems in robotics, the LQR problem and its variants have received a tremendous amount of attention over the past 60 years. The work most related to the current one is a recent paper by Laine and Tomlin [24], which solves the LQR problem in parallel by splitting the trajectory into sub-trajectories and adding additional constraints, based on previous work applying MPC to distributed systems that used a nearly identical approach to decompose the distributed system into a set of smaller problems [25].

Recently, the traditional LQR problem has been extended to work with stage-wise equality constraints [26], [27]. The linear system (4a) associated with LQR can also be solved using any of the general techniques for solving sparse linear systems in parallel, including parallel QR [28], block-cyclic reduction [29], [30], multigrid methods [31], [32], and indirect Krylov methods such as preconditioned conjugate-gradient [33].

Other notable work related to parallelizing optimal control or trajectory optimization problems includes the qpDUNES solver [34] that parallelizes over the time horizon using block-cyclic reduction, a recent paper solving contact-aware problems on a GPU using a combination of indirect methods and block-cyclic reduction [35], an FPGA implementation of a linear MPC solver using the MINRES indirect method [36], and a variety of ADMM or augmented Lagrangian-based methods [37]–[41].

Like [24], the proposed algorithm parallelizes the LQR problem over the time horizon, but instead of assuming an explicit discrete dynamics function, our algorithm can be trivially extended to work with implicit integrators such as the Hermite-Simpson method commonly found when solving optimal control problems with direct collocation. It can also easily handle stage-wise equality constraints.

While the proposed algorithm borrows ideas from the literature on solving symmetric sparse linear systems in parallel, it is, to the authors' best knowledge, a novel application and specialization of those ideas to the optimal control problem. We also provide a documented open-source parallelized implementation of our algorithm with a convenient wrapper written in a high-level programming language, where many of the previously cited works are purely theoretical or don't provide an open-source implementation.

III. Background

A. Notation

We use interval notation $j \in (a, b] := \{a+1, a+2, \ldots, b\}$, $j \in [a, b] := \{a, a+1, \ldots, b\}$, for $j, a, b \in \mathbb{N}$ to denote sets of consecutive integers. We use angle-bracket notation $\langle x, y \rangle$ to denote $x^T y$ for both vectors and matrix arguments.

B. The Linear Quadratic Regulator

The Linear Quadratic Regulator (LQR) problem is the canonical problem in optimal control, since it can be solved using a variety of methods and is amenable to theoretical analysis. It is also of practical significance since LQR problems arise when an unconstrained nonlinear optimal control problem is approximated using a 2nd-order Taylor expansion of the cost function and a linear approximation of the dynamics. The LQR problem, shown here with affine terms included, has the form:

$$\begin{aligned} \underset{x_{1:N},\, u_{1:N-1}}{\text{minimize}} \quad & \sum_{k=1}^{N} \frac{1}{2} x_k^T Q_k x_k + q_k^T x_k \\ & + \sum_{k=1}^{N-1} \frac{1}{2} u_k^T R_k u_k + r_k^T u_k \qquad (1) \\ \text{subject to} \quad & x_{k+1} = \hat{A}_k x_k + \hat{B}_k u_k + f_k = 0, \\ & x_1 = x_{\text{init}} \end{aligned}$$

where $x_k \in \mathbb{R}^n$ and $u_k \in \mathbb{R}^m$ are the state and control vectors at time step k, N is the number of time steps (also referred to as the time horizon), and $\hat{A} \in \mathbb{R}^{n \times n}, \hat{B} \in \mathbb{R}^{n \times m}, f \in \mathbb{R}^n, Q \in \mathbb{R}^{n \times n}, R \in \mathbb{R}^{m \times m}, q \in \mathbb{R}^n$, and $r \in \mathbb{R}^m$ come from the Taylor expansions of the cost and dynamics about some reference trajectory (or point).

The well-known first-order optimality, or KKT, conditions for (1) are [42]:

$$Q_k x_k + q_k + \hat{A}_k^T \lambda_{k+1} - \lambda_k = 0, \quad k \in [1, N) \qquad (2a)$$

$$R_k u_k + r_k + \hat{B}_k^T \lambda_{k+1} = 0, \quad k \in [1, N) \qquad (2b)$$

$$Q_N x_N + q_N - \lambda_N = 0 \qquad (2c)$$

$$x_{k+1} = \hat{A}_k x_k + \hat{B}_k u_k + f_k, \; k \in [1, N) \qquad (2d)$$

$$x_1 = x_{\text{init}} \qquad (2e)$$

which can be written in matrix form as the following linear system:

$$Hv = g \qquad (3)$$

where for $N = 4$:

$$H = \begin{bmatrix} Q_1 & & & & -I & \hat{A}_1^T & & & \\ & R_1 & & & & \hat{B}_1^T & & & \\ & & Q_2 & & & -I & \hat{A}_2^T & & \\ & & & R_2 & & & \hat{B}_2^T & & \\ & & & & Q_3 & & & -I & \hat{A} \\ & & & & & R_3 & & & \hat{B} \\ & & & & & & Q_4 & & -I \\ -I & & & & & & & & \\ \hat{A}_1 & \hat{B}_1 & -I & & & & & & \\ & & \hat{A}_2 & \hat{B}_2 & -I & & & & \\ & & & & \hat{A}_3 & \hat{B}_3 & -I & & \end{bmatrix} \tag{4a}$$

$$v = \begin{bmatrix} x_1^T & u_1^T & x_2^T & u_2^T & x_3^T & u_3^T & x_4^T & \lambda_1^T & \lambda_2^T & \lambda_3^T & \lambda_4^T \end{bmatrix}^T \tag{4b}$$

$$g = -\begin{bmatrix} q_1^T & r_1^T & q_2^T & r_2^T & q_3^T & r_3^T & q_4^T & x_{\text{init}}^T & f_1^T & f_2^T & f_3^T \end{bmatrix}^T \tag{4c}$$

Note that H is a symmetric, quasidefinite matrix (known as a KKT matrix) with $Nn+(N-1)m$ positive eigenvalues and Nn negative eigenvalues [42].

C. Schur Compliments

A Schur compliment is a common trick that reduces solving a single large linear system to solving a series of smaller ones. We will use the Schur compliment to solve symmetric 3x3 block-tridiagonal systems of the form:

$$\begin{bmatrix} A & D & \\ D^T & B & E^T \\ & E & C \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}. \tag{5}$$

We can solve the first and last rows of (5) for x and z:

$$x = A^{-1}(a - Dy) = \bar{a} - \bar{D}y, \tag{6a}$$

$$z = C^{-1}(c - Ey) = \bar{c} - \bar{E}y, \tag{6b}$$

where we calculate the following pieces numerically by factorizing A and C:

$$\bar{D} = A^{-1}E, \quad \bar{a} = A^{-1}a, \tag{7a}$$

$$\bar{E} = C^{-1}D, \quad \bar{c} = C^{-1}c. \tag{7b}$$

We can use (6) to eliminate x and z from the middle row, allowing us to solve for y:

$$\begin{aligned} y &= -(D^T\bar{D} + E^T\bar{E} - B)^{-1}(b - E^T\bar{c} - D^T\bar{a}) \\ &= -\bar{B}^{-1}\bar{b} \end{aligned} \tag{8}$$

After solving (8) numerically, we can now plug the result into (6) to get our completed solution vector.

IV. A Parallel LQR Algorithm

A. Recursive Schur Compliments

We now derive our parallelizable algorithm for solving linear systems of the form (3). First, we reorder the rows and columns of (4a) to get a banded matrix with a bandwidth of $2n + m - 1$ (see Fig. 1).

Note that, for clarity, we've ignored the special boundary conditions at the first and last time steps. From the coloring we see that the LQR problem takes the 3x3 block tridiagonal

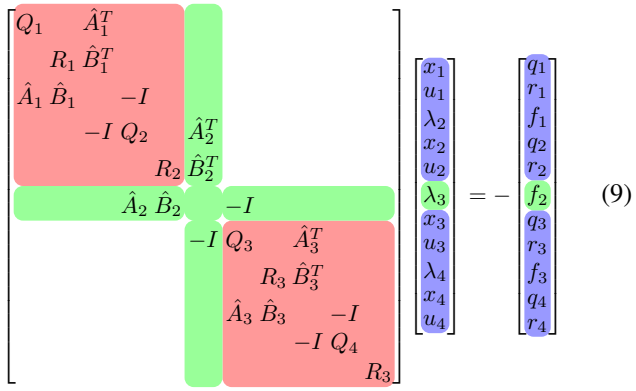


Fig. 1: The LQR linear system, rearranged and partitioned to appear in the form of (5). The red block in the upper left corner is A, the red lower-right block is C. The top vertical green block is D, and the bottom green vertical block is E. The solution and right-hand-side vectors have also been partitioned, with the top blue blocks equating to x and a, the middle green block to y and b, and the bottom blue blocks to z and c. We use these blocks to solve the LQR problem using the steps in Sec. III-C.

form of (5). When solving for $\bar{a}, \bar{c}, \bar{D}$, and $\bar{E}$ in (7) we make a critical observation:

$$\bar{D} = -\begin{bmatrix} Q_1 & \hat{A}_1^T & & \\ & R_1 & \hat{B}_1^T & \\ \hat{A}_1 & \hat{B}_1 & -I & \\ & & -I & Q_2 \\ & & & R_2 \end{bmatrix}^{-1} \begin{bmatrix} & \\ & \\ & \\ \hat{A}_2^T & \\ \hat{B}_2^T & \end{bmatrix}, \quad \bar{a} = -\begin{bmatrix} Q_1 & \hat{A}_1^T & & \\ & R_1 & \hat{B}_1^T & \\ \hat{A}_1 & \hat{B}_1 & -I & \\ & & -I & Q_2 \\ & & & R_2 \end{bmatrix}^{-1} \begin{bmatrix} q_1 \\ r_1 \\ f_1 \\ q_2 \\ r_2 \end{bmatrix}, \tag{10a}$$

$$\bar{E} = -\begin{bmatrix} Q_3 & \hat{A}_3^T & & \\ & R_3 & \hat{B}_3^T & \\ \hat{A}_3 & \hat{B}_3 & -I & \\ & & -I & Q_4 \\ & & & R_4 \end{bmatrix}^{-1} \begin{bmatrix} -I & \\ & \\ & \\ & \\ & \end{bmatrix}, \quad \bar{c} = -\begin{bmatrix} Q_3 & \hat{A}_3^T & & \\ & R_3 & \hat{B}_3^T & \\ \hat{A}_3 & \hat{B}_3 & -I & \\ & & -I & Q_4 \\ & & & R_4 \end{bmatrix}^{-1} \begin{bmatrix} q_3 \\ r_3 \\ f_3 \\ q_4 \\ r_4 \end{bmatrix}. \tag{10b}$$

Due to the banded structure of the matrix, each of these smaller linear systems also takes the form of (5). Note that, for each diagonal A or C block, we solve two linear systems, one with right-hand-side data from the original right-hand-side vector (shown in blue), and one with right-hand-side data from the green columns in Fig. 1.

If we apply the same technique to each of these 4 linear systems, we'd get 16 total linear systems to solve. However, after eliminating duplicates, we're left with 12 unique small linear systems,

$$\bar{D}_1^{(1)} = -\begin{bmatrix} Q_1 & \\ & R_1 \end{bmatrix}^{-1} \begin{bmatrix} \hat{A}_1^T \\ \hat{B}_1^T \end{bmatrix}, \quad \bar{a}_1^{(1,2)} = -\begin{bmatrix} Q_1 & \\ & R_1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \bar{a}_1^{(1,3)} = -\begin{bmatrix} Q_1 & \\ & R_1 \end{bmatrix}^{-1} \begin{bmatrix} q_1 \\ r_1 \end{bmatrix}, \tag{11a}$$

$$\bar{E}_1^{(1)} = -\begin{bmatrix} Q_2 & \\ & R_2 \end{bmatrix}^{-1} \begin{bmatrix} -I \\ \end{bmatrix}, \quad \bar{c}_1^{(1,2)} = -\begin{bmatrix} Q_2 & \\ & R_2 \end{bmatrix}^{-1} \begin{bmatrix} \hat{A}_2^T \\ \hat{B}_2^T \end{bmatrix}, \quad \bar{c}_1^{(1,3)} = -\begin{bmatrix} Q_2 & \\ & R_2 \end{bmatrix}^{-1} \begin{bmatrix} q_2 \\ r_2 \end{bmatrix}, \tag{11b}$$

$$\bar{D}_2^{(1)} = -\begin{bmatrix} Q_3 & \\ & R_3 \end{bmatrix}^{-1} \begin{bmatrix} \hat{A}_3^T \\ \hat{B}_3^T \end{bmatrix}, \quad \bar{a}_2^{(1,2)} = -\begin{bmatrix} Q_3 & \\ & R_3 \end{bmatrix}^{-1} \begin{bmatrix} -I \\ \end{bmatrix}, \quad \bar{a}_2^{(1,3)} = -\begin{bmatrix} Q_3 & \\ & R_3 \end{bmatrix}^{-1} \begin{bmatrix} q_3 \\ r_3 \end{bmatrix}, \tag{11c}$$

$$\bar{E}_2^{(1)} = -\begin{bmatrix} Q_4 & \\ & R_4 \end{bmatrix}^{-1} \begin{bmatrix} -I \\ \end{bmatrix}, \quad \bar{c}_2^{(1,2)} = -\begin{bmatrix} Q_4 & \\ & R_4 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \bar{c}_2^{(1,3)} = -\begin{bmatrix} Q_4 & \\ & R_4 \end{bmatrix}^{-1} \begin{bmatrix} q_4 \\ r_4 \end{bmatrix}, \tag{11d}$$

where (11a) and (11b) come from (10a) and (11c) and (11d) come from (10b). We also introduce some notation, since we

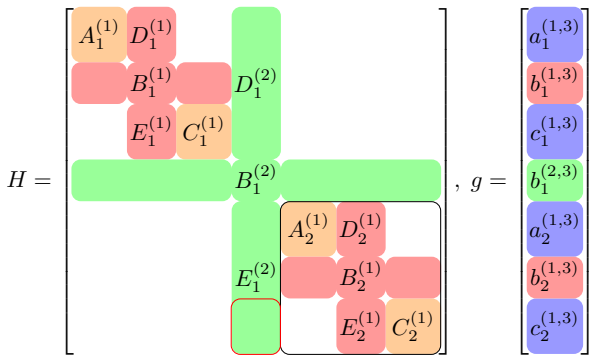


Fig. 2: The LQR KKT system after recursively partitioning with Schur compliments with a depth of $K = 2 = \log_2(N)$ with $N = 4$. The labels follow the notation used in Algorithm 1. Levels are enumerated starting at $j = 1$ for the deepest recursion level (the orange and red blocks), up to $j = K$ (the green blocks). We assign the right-hand-side vector a level of $K + 1$ (the blue blocks). We use $A_i^{(j)}$ and $C_i^{(j)}$ to denote the ith A and C blocks at level j, e.g. the block outlined in black is $C_1^{(2)}$. We use the same notation for $D_i^{(j)}$ and $E_i^{(j)}$, as shown. We use $a_i^{(j,p)}$ and $c_i^{(j,p)}$ to refer to the block with the rows of $D_i^{(j)}$ and $E_i^{(j)}$, respectively, and the columns of either $D_i^{(p)}$ or $E_i^{(p)}$. For example, the block outlined in red is $c_2^{(1,2)}$ since it corresponds to the rows of the red block $E_2^{(1)}$ but taken from the green data / columns of level $p = 2$. Since the right-hand-side vector g is given a level of $K + 1$, its blocks all have $p = 3$. We've partitioned the g vector from the lowest level, coloring the blocks corresponding to the $B_i^{(j)}$ blocks to match the level j.

need to distinguish between all of the different $\bar{a}, \bar{c}, \bar{D}$, and $\bar{E}$ blocks. We use $\bar{a}_i^{(j,p)}$ and $\bar{c}_i^{(j,p)}$ to denote the ith $\bar{a}$ or $\bar{c}$ at a recursion depth of j using right-hand-side data from a "parent" depth of p, where depth is measured from the bottom. We also note that, as suggested by (11), $\bar{a}_i^{(j,j)} = \bar{D}_i^{(j)}$ and $\bar{c}_i^{(j,j)} = \bar{E}_i^{(j)}$. The notation is further clarified in Fig. 2. Since the linear systems in (10) all involve block-diagonal matrices, we can solve all of these systems directly using e.g. a dense Cholesky decomposition. With these pieces we use (8) to get all the y's, followed by (6) to get all the x's and z's, which concatentated form the $\bar{a}$, $\bar{c}$, $\bar{D}$, and $\bar{E}$ at the next level, i.e. (10). These are then used to solve the top-level Schur compliment for the solution vector using the same procedure.

It should be clear that the resulting algorithm is naturally recursive and results in a binary tree of Schur compliments, since each 3x3 Schur compliment requires solving another Schur compliment problem for both A and C. Borrowing terminology from tree graphs, we will often refer to e.g. $D_i^{(j)}$ as the D for the ith "leaf" of "level" j. While a naïve recursive implementation of this algorithm turns out to be computationally inefficient and hard to parallelize, we can "unroll" the recursion following the steps of the previous paragraphs. The resulting algorithm is summarized in Algorithm 1.

As we saw in the preceding paragaphs, our algorithm starts at the lowest level by factorizing all of the orange blocks along the diagonal, and then using those factorizations to solve all of the systems in (11), corresponding to lines 1-3. It's important to note that we get different right-hand-side data for each of the "upper" levels, corresponding to the data from the red, green, and blue blocks from the same rows as the diagonal block. After this initial computation, in lines 7-18 we loop over each of the levels, using the pieces from the previous level to calculate a y and then x and z for each of the upper levels to get all the $\bar{a}$, $\bar{c}$, $\bar{D}$, or $\bar{E}$ for the next level. In Algorithm 1, the right-hand-side vector is represented as the highest level, $K+1$ where $K = \log_2 N$ is the number of levels for a problem with a horizon length of N (see Fig. 2 for more details).

The following section describes further considerations and modifications that are needed to efficiently implement the algorithm on a many-core processor.

Algorithm 1 Recursive Schur Compliments

1: **for** $i \in (0, 2^{K-1}]$ **do**
2: **for** $p \in (0, K+1]$ **do**
3: Solve $A_i^{(1)} \bar{a}_i^{(1,p)} = a_i^{(1,p)}$ using Cholesky
4: Solve $C_i^{(1)} \bar{c}_i^{(1,p)} = c_i^{(1,p)}$ using Cholesky
5: **end for**
6: **end for**
7: **for** $j \in (0, K]$ **do**
8: **for** $i \in (0, 2^{K-j}]$ **do**
9: $\bar{B}_i^{(j)} = \langle D_i^{(j)}, \bar{D}_i^{(j)} \rangle + \langle E_i^{(j)}, \bar{E}_i^{(j)} \rangle - B_i^{(j)}$
10: Factorize $\bar{B}_i^{(j)}$
11: **for** $p \in (j, K+1]$ **do**
12: $\bar{b}_i^{(j,p)} \leftarrow b_i^{(j,p)} - \langle D_i^{(j)}, \bar{a}_i^{(j,p)} \rangle - \langle E_i^{(j)}, \bar{c}_i^{(j,p)} \rangle$
13: Solve $-\bar{B}_i^{(j)} y_i^{(j,p)} = \bar{b}_i^{(j,p)}$ using Cholesky
14: $x_i^{(j,p)} \leftarrow \bar{a}_i^{(j,p)} - \bar{D}_i^{(j)} y_i^{(j,p)}$
15: $z_i^{(j,p)} \leftarrow \bar{c}_i^{(j,p)} - \bar{E}_i^{(j)} y_i^{(j,p)}$
16: **end for**
17: **end for**
18: **end for**

B. *Parallelization*

We now proceed with the derivation of our final algorithm, a "flattened" version of Algorithm 1 to make it amenable to implementation on a many-core processor using e.g. OpenMP. By carefully analyzing the dependency graph of Algorithm 1 we identified the critical path, which we used to determine the synchronization points for the algorithm. This section provides the key highlights of the final algorithm; interested readers should consult the provided open-source implementation for the full details of the algorithm.

In lines 1-6 of Algorithm 1 we compute $\bar{a}, \bar{c}, \bar{D}$, and $\bar{E}$ at the bottom level, using right-hand-side data from each of the upper levels. It should be obvious looking at Fig. 1 that at most two levels plus the right-hand-side at level $K + 1$ will have non-zero data, since any row containing a Q_k or

R_k has at most two other entries. We can replace these lines with a loop over time indices k that, for each Q_k and R_k, solves $-Q_k^{-1}q_k$, $Q_k^{-1}\hat{A}_k^T$, $Q_k^{-1}(-I)$, $-R_k^{-1}r_k$, and $R_k^{-1}\hat{B}_k^T$, with special-casing applied to the initial and final time steps. We denote the function that handles all of this for each time step SOLVELEAF(k).

Examining the main loop of Algorithm 1, we notice that we need to calculate many terms of the form

$$\langle D, \bar{a} \rangle \text{ or } \langle E, \bar{c} \rangle . \tag{12}$$

where the $\bar{a}$ and $\bar{c}$ terms come from each of the upper levels (line 12), as well as the current one (line 9). We can calculate all of these terms in parallel since they are completely independent. While the inner dimension of these inner products gets larger at higher levels, they all have the same computational cost since the D and E for each level all have the same form:

$$D = \begin{bmatrix} \vdots \\ \hat{A}_k^T \\ \hat{B}_k^T \end{bmatrix} \quad E = \begin{bmatrix} -I \\ 0 \\ \vdots \end{bmatrix} \tag{13}$$

Leveraging this structure allows us to compute inner products solely on the blocks corresponding to the states and controls at the current and next time steps. We denote the function that computes $\bar{b}$ for level j, leaf i (which includes calculating $\bar{B}$ since $\bar{b}_i^{(j,j)} = \bar{B}_i^{(j)}$) and parent level p as INNERPRODUCTS(i, j, p).

After calculating all the $\bar{b}$ and $\bar{B}$ terms we compute the Cholesky factorization of $\bar{B}$ and solve for $y = \bar{B}^{-1}\bar{b}$ for all leaves i and upper levels p. We denote the functions that do each of these operations FACTORIZEBBAR(i, j) and SOLVEBBAR(i, j, p).

The last step is to calculate x and z (lines 14-15 in Algorithm 1). This step updates all of the $\bar{a}, \bar{c}, \bar{D}$, and $\bar{E}$ terms for the next level. We can see from Fig. 2 that this affects every row of the columns associated with the upper levels, except those which correspond to a B for the current or upper levels. Since our updates are of the form:

$$\bar{a} \leftarrow \bar{a} - \bar{D}y \tag{14a}$$

$$\bar{c} \leftarrow \bar{c} - \bar{E}y, \tag{14b}$$

each row can be updated in parallel. We perform these updates blockwise, parallelizing over knot points and upper levels. We can write down a function UPATESCHUR(k, j, p) that performs (14) based on the knot point index k, level j, and upper level p.

Putting this all together, our algorithm is summarized in Algorithm 2, which we refer to as rsLQR in the following sections. As shown by the **parfor** loops in Algorithm 2, most of the computation can be done in parallel. Each **parfor** carries an implicit synchronization before continuing to the next loop, and corresponds with the synchronization steps needed along the critical path. The next section analyses the theoretical computational properties of this algorithm.

Algorithm 2 Recursive Schur LQR (rsLQR)

1: **parfor** k = 1:N **do**
2: SOLVELEAF(k)
3: **end parfor**
4: **for** $j \in (0, D]$ **do**
5: $L = 2^{K-j}$ (number of leaves)
6: **parfor** $i \in (0, L], p \in [j, K+1]$ **do**
7: INNERPRODUCTS(i, j, p)
8: **end parfor**
9: **parfor** $i \in (0, L]$ **do**
10: FACTORIZEBBAR(i, j)
11: **end parfor**
12: **parfor** $i \in (0, L], p \in (j, K+1]$ **do**
13: SOLVEBBAR(i, j, p)
14: **end parfor**
15: **parfor** $k \in (0, N], p \in (j, K+1]$ **do**
16: UPDATESCHUR(k, j, p)
17: **end parfor**
18: **end for**

V. THEORETICAL COMPLEXITY

We compare the theoretical complexity of our algorithm against Riccati recursion, an extremely efficient but inherently serial algorithm for solving LQR problems. We approximated the floating point operations for each algorithm using the approximate floating point complexity for the fundamental linear algebra operations given in Table I. To account for parallelization, the total number of operations of each of the parallel loops was divided by the number of processors, thresholding when the number of processors exceeded the number of individual tasks. Only parallelization of the parallel for-loops in Algorithm 2 is considered: low-level parallelization of the linear algebra via SIMD instructions, instruction-level-parallelism, or multithreading was ignored for both algorithms. While this could offer significant performance improvements, achieving an implementation that efficiently scales with both levels of parallelism is nontrival and left for future work.

The state, control, and horizon complexity for Riccati and our algorithm rsLQR with an increasing number of processors is shown in Fig. 3. The maximum number of processors, 4096, was chosen such that all parallelization that can be exploited is exploited for the longest horizon length of 512. The number of processors needed to fully exploit the parallelism available with N time steps is given by:

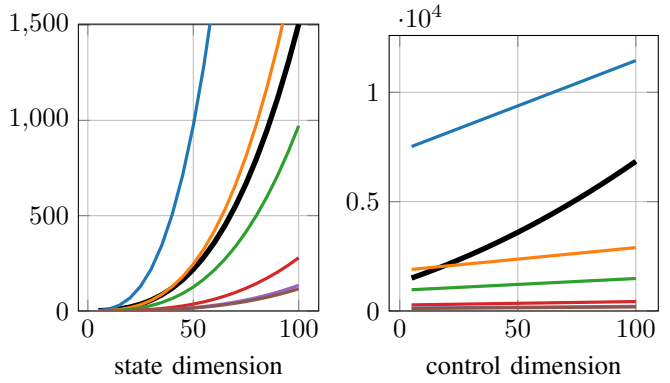
$$P_{\text{max}} = N\left(\log_2(N) - 1\right) \tag{15}$$

As shown, the proposed algorithm beats Riccati recursion with 32-64 cores. With 64 cores—currently the high end of what is available on modern multicore CPUs—the proposed algorithm is 2-5x faster than standard Riccati recursion. When fully exploiting the given parallelism (again, ignoring low-level parallelism in the linear algebra), we achieve the expected $\log(N)$ complexity, with performance about 45x faster than Riccati at a horizon length of 512. For problems with

TABLE I: Theoretical Complexity for Linear Algebra

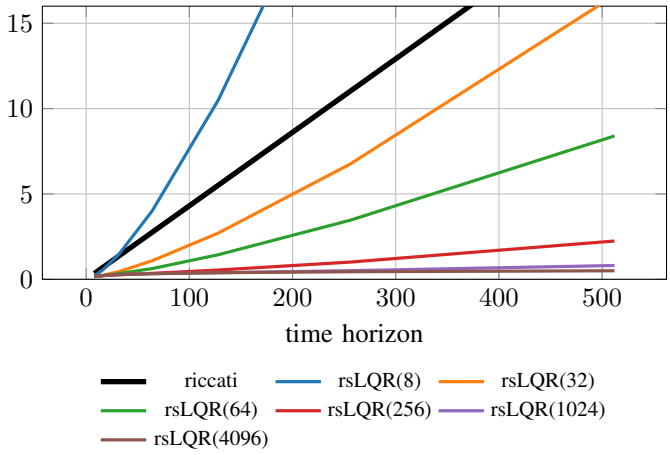
Method	Complexity
Matrix Multiplication	$2nmp$
Matrix Multiplication w/ addition	$np(3m+1)$
Cholesky factorization	$\frac{n(n+1)(n-1)}{3} + n + \frac{n(n+1)}{2}$
Cholesky solve	$2pn^2$

For matrix multiplication $C = AB$, $A \in \mathbb{R}^{n\times m}$, $B \in \mathbb{R}^{m\times p}$. For cholesky factorization, $A \in \mathbb{R}^{n\times n}$, and for a Cholesky solve $Ax = b$, $b \in \mathbb{R}^{n\times p}$.



(a) State complexity with $m = 5$ controls and a horizon of $N = 256$.

(b) Control complexity with $n = 100$ states and a horizon of $N = 256$.



(c) Horizon complexity with $n = 14$ states and $m = 7$ controls (the dimensions of a standard 7DOF robot arm).

Fig. 3: Theoretical complexity vs Riccati for an increasing number of processors. The legend entry rsLQR(P) represents the rsLQR algorithm run with P processors. All results are in units of millions of floating-point operations.

thousands of knot points (e.g. 4096), such as those found in space trajectory design [23], our algorithm is over 250x faster with $n = 14$ and $m = 7$.

VI. Computational Results

A. Implementation Details

The algorithm was implemented in pure C, using OpenMP for shared-memory parallelization. A Julia wrapper is also provided. While the algorithm may also map well to a distributed-memory architecture, given its relatively low communication requirements between workers, this is left for future work. Using OpenMP, a single threadpool was maintained for the entire algorithm, and work was statically divided amongst the threads by dividing the work into equal-sized portions.

Since the algorithm only requires basic element-wise matrix operations, matrix multiplication, and Cholesky decomposition, a custom linear algebra library was written in pure C, which provided better scaling with the number of cores than Eigen [43], OpenBLAS [44], or Intel MKL [45]. The memory required by the algorithm is allocated in one large chunk, which is subsequently assigned in consecutive blocks to the blocks required by the algorithm, increasing data locality and reducing the likelihood of cache misses.

The code for the examples was compiled in Release mode (i.e. full optimizations) using Clang 13 on a desktop with an AMD 3990X processor with 64 cores running Pop!_OS 21.10. The results for Riccati recursion are based on an implementation in pure C using the exact same linear algebra routines and build system as the rsLQR algorithm. To guarantee that all linear systems were strictly quasidefinite, a small about of regularization was added to the dual variables for all problems. The code is freely available at https://github.com/bjack205/rsLQR.

B. Numerical Results

As shown in Fig. 4, actual performance closely matched theoretical predictions once the horizon was long enough to offset the overhead of launching a significant number of worker threads. At a horizon length of 512 with 64 cores, our algorithm is 50% faster than Riccati recursion.

Figure 5 compares the computation time versus horizon length for several solvers. All solvers were called from Julia, and solve times are the median over 100 samples. As shown, the proposed algorithm performs significantly better on long horizons than parallelized commercial solvers for symmetric sparse systems, with a 183% improvement over Pardiso 6.0 [46] and an 84% improvement over MA86 [47] at a horizon length of $N = 512$. In our tests, neither of these solvers scaled very well with increased parallelism, often only providing marginal improvements. Our algorithm demonstrated performance on-par with with the SuiteSparse package, and was only beat by the single-threaded QDLDL algorithm [48], whose performance on these KKT systems is significantly better than all other solvers.

VII. Discussion

We have demonstrated a new parallelizable algorithm for solving the sparse linear systems that arise when solving trajectory optimization and other related optimal-control problems. This algorithm offers greater generality than other LQR-based approaches, allowing straightforward adaptations to handle implicit integration methods and constraints. With 64 parallel threads, the proposed algorithm offers a 50% improvement over Riccati recursion at a horizon length of 512,

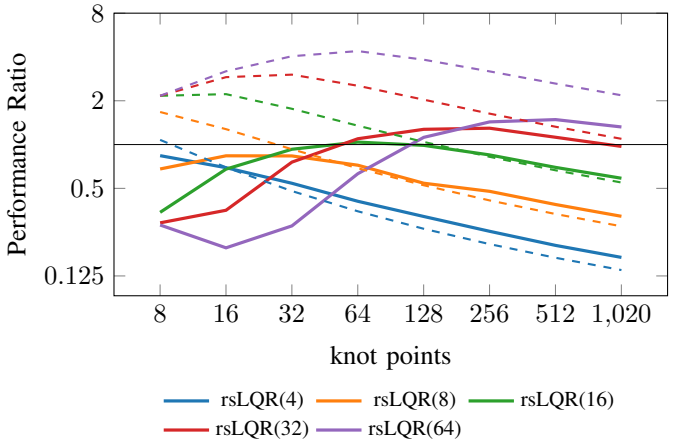


Fig. 4: Comparison of actual (solid lines) and theoretical (dashed lines) speedup of rsLQR versus Riccati recursion for varying horizon lengths for a system with $n = 14$ states and $m = 7$ controls. Values greater than 1 mean rsLQR was faster than Riccati.

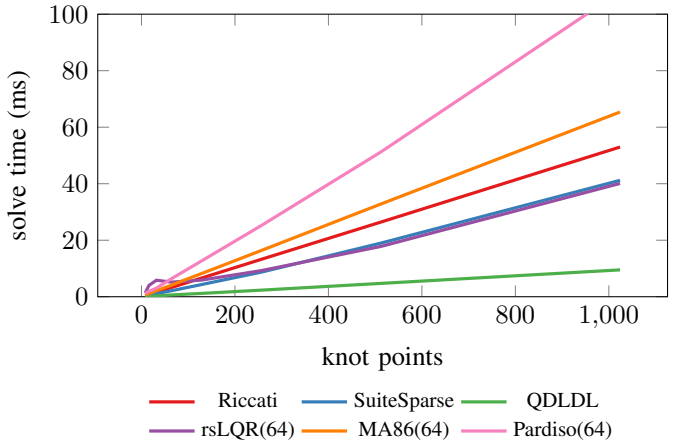


Fig. 5: Comparison of solve time vs horizon length for several state-of-the-art sparse matrix solvers. Parallelizable methods are followed by a number in parentheses denoting the number of cores used to compute the solution.

while again being strictly more general in its applicability. It also beats existing commerical parallelized sparse symmetric linear system solvers such as HSL MA86 (by about 80%) and Pardiso 6.0 (by about 180%) at a horizon length of 512, and scales much better with increased parallelism.

Substantial areas for future work remain, such as adapting the implementation to work on distributed-memory architectures and comparisons with distributed memory linear algebra packages such as ScaLAPACK [49]. While our algorithm's performance was only matched or beaten by SuiteSparse and QDLDL, these algorithms don't work on large-scale distributed-memory architectures. Future work may also investigate GPU or FPGA implementations; however, while modern GPUs have thousands of "cores", mapping the current algorithm onto the SIMD-style parallelism of a GPU would require significant modification to maintain high performance due its communication and synchronization requirements. Future work will also investigate the performance benefits of integrating the proposed method into nonlinear trajectory optimization methods such as direct collocation via sequential quadratic or convex programming.

References

[1] S. M. LaValle, J. J. Kuffner, B. Donald, *et al.*, "Rapidly-exploring random trees: Progress and prospects," *Algorithmic and computational robotics: new directions*, vol. 5, pp. 293–308, 2001.

[2] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.

[3] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, "Aggressive driving with model predictive path integral control," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 1433–1440.

[4] D. Mayne, "A Second-order Gradient Method for Determining Optimal Trajectories of Non-linear Discrete-time Systems," *International Journal of Control*, vol. 3, no. 1, pp. 85–95, Jan. 1, 1966.

[5] Y. Tassa, N. Mansard, and E. Todorov, "Control-limited differential dynamic programming," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, May 2014, pp. 1168–1175.

[6] T. A. Howell, B. E. Jackson, and Z. Manchester, "ALTRO: A Fast Solver for Constrained Trajectory Optimization," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Macau, China, Nov. 2019.

[7] B. E. Jackson, T. Punnoose, D. Neamati, K. Tracy, and R. Jitosho, "ALTRO-C: A Fast Solver for Conic Model-Predictive Control," presented at the International Conference on Robotics and Automation (ICRA), Xi'an, China, 2021, p. 8.

[8] M. Giftthaler, M. Neunert, M. Stäuble, J. Buchli, and M. Diehl. "A Family of Iterative Gauss-Newton Shooting Methods for Nonlinear Optimal Control." (Dec. 11, 2017), [Online]. Available: http : / / arxiv . org / abs / 1711 . 11006 (visited on 01/05/2021).

[9] C. Mastalli, R. Budhiraja, W. Merkt, *et al.* "Crocoddyl: An Efficient and Versatile Framework for Multi-Contact Optimal Control." (Sep. 11, 2019).

[10] F. Farshidian, M. Neunert, A. W. Winkler, G. Rey, and J. Buchli, "An efficient optimal planning and control framework for quadrupedal locomotion," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 93–100.

[11] P. E. Gill, W. Murray, and M. A. Saunders, "SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization," *SIAM Review*, vol. 47, no. 1, pp. 99–131, Jan. 2005.

[12] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, Mar. 2006.

[13] C. Hargraves and S. Paris, "Direct trajectory optimization using nonlinear programming and collocation," *Journal of Guidance, Control, and Dynamics*, vol. 10, no. 4, pp. 338–342, Jul. 1, 1987.

[14] M. Posa, S. Kuindersma, and R. Tedrake, "Optimization and stabilization of trajectories for constrained dynamical systems," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2016, pp. 1366–1373.

[15] J. T. Betts, *Practical methods for optimal control and estimation using nonlinear programming*. SIAM, 2010.

[16] A. Hereid and A. D. Ames, "Frost*: Fast robot optimization and simulation toolkit," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 719–726.

[17] Z. Manchester and S. Kuindersma, "DIRTREL: Robust Trajectory Optimization with Ellipsoidal Disturbances and LQR Feedback," in *Robotics: Science and Systems XIII*, Robotics: Science and Systems Foundation, Jul. 12, 2017.

[18] T. Howell, C. Fu, and Z. Manchester, "Direct Policy Optimization using Deterministic Sampling and Collocation," *IEEE Robotics and Automation Letters*, pp. 1–1, 2021.

[19] B. Plancher and S. Kuindersma, "A performance analysis of parallel differential dynamic programming on a gpu.," in *WAFR*, 2018, pp. 656–672.

[20] F. Farshidian, E. Jelavic, A. Satapathy, M. Giftthaler, and J. Buchli, "Real-time motion planning of legged robots: A model predictive control approach," in *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, Nov. 2017, pp. 577–584.

[21] A. George, "Nested Dissection of a Regular Finite Element Mesh," *SIAM Journal on Numerical Analysis*, vol. 10, no. 2, pp. 345–363, Apr. 1, 1973.

[22] M. S. Khaira, G. L. Miller, and T. J. Sheffler, "Nested Dissection: A survey and comparison of various nested dissection algorithms," p. 29,

[23] K. Tracy and Z. Manchester, "Low-thrust trajectory optimization using the kustannheimo-stiefel transformation," in *AAS/AIAA Astrodynamics Specialist Conference*, 2021, pp. 1–12.

[24] F. Laine and C. Tomlin, "Parallelizing lqr computation through endpoint-explicit riccati recursion," in *2019 IEEE 58th Conference on Decision and Control (CDC)*, IEEE, 2019, pp. 1395–1402.

[25] D. Soudbakhsh and A. M. Annaswamy, "Parallelized model predictive control," in *2013 American Control Conference*, IEEE, 2013, pp. 1715–1720.

[26] J. Brüdigam and Z. Manchester, "Linear-quadratic optimal control in maximal coordinates," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2021, pp. 9775–9781.

[27] S. Yang, G. Chen, Y. Zhang, F. Dellaert, and H. Choset, "Equality constrained linear optimal control with factor graphs," *arXiv preprint arXiv:2011.01360*, 2020.

[28] S. N. Yeralan, T. A. Davis, W. M. Sid-Lakhdar, and S. Ranka, "Algorithm 980: Sparse qr factorization on the gpu," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 2, pp. 1–29, 2017.

[29] S. P. Hirshman, K. S. Perumalla, V. E. Lynch, and R. Sanchez, "Bcyclic: A parallel block tridiagonal matrix cyclic solver," *Journal of Computational Physics*, vol. 229, no. 18, pp. 6392–6404, 2010.

[30] W. Gander and G. H. Golub, "Cyclic reduction—history and applications," *Scientific computing (Hong Kong, 1997)*, vol. 7385, 1997.

[31] U. M. Yang *et al.*, "Boomeramg: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002.

[32] J. E. Jones and S. F. McCormick, "Parallel multigrid methods," in *Parallel Numerical Algorithms*, Springer, 1997, pp. 203–224.

[33] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler, "Preconditioned krylov solvers on gpus," *Parallel Computing*, vol. 68, pp. 32–44, 2017.

[34] J. V. Frasch, S. Sager, and M. Diehl, "A parallel quadratic programming method for dynamic optimization problems," *Mathematical Programming Computation*, vol. 7, no. 3, pp. 289–329, Sep. 1, 2015.

[35] Z. Pan, B. Ren, and D. Manocha, "Gpu-based contact-aware trajectory optimization using a smooth force model," in *Proceedings of the 18th annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2019, pp. 1–12.

[36] J. L. Jerez, G. A. Constantinides, E. C. Kerrigan, and K.-V. Ling, "Parallel mpc for real-time fpga-based implementation," *IFAC Proceedings Volumes*, vol. 44, no. 1, pp. 1338–1343, 2011.

[37] D. Kouzoupis, R. Quirynen, B. Houska, and M. Diehl, "A block based aladin scheme for highly parallelizable direct optimal control," in *2016 American Control Conference (ACC)*, IEEE, 2016, pp. 1124–1129.

[38] L. Yu, A. Goldsmith, and S. Di Cairano, "Efficient convex optimization on gpus for embedded model predictive control," in *Proceedings of the General Purpose GPUs*, 2017, pp. 12–21.

[39] C. A. Alonso and S.-H. Tseng, "Effective gpu parallelization of distributed and localized model predictive control," *arXiv preprint arXiv:2103.14990*, 2021.

[40] Z. Zhou and Y. Zhao, "Accelerated admm based trajectory optimization for legged locomotion with coupled rigid body dynamics," in *2020 American Control Conference (ACC)*, IEEE, 2020, pp. 5082–5089.

[41] V. Sindhwani, R. Roelofs, and M. Kalakrishnan, "Sequential operator splitting for constrained nonlinear optimal control," in *2017 American Control Conference (ACC)*, IEEE, 2017, pp. 4864–4871.

[42] S. Boyd, S. P. Boyd, and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[43] G. Guennebaud, B. Jacob, *et al.*, *Eigen v3*, http://eigen.tuxfamily.org, 2010.

[44] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 blas performance optimization on loongson 3a processor," in *2012 IEEE 18th international conference on parallel and distributed systems*, IEEE, 2012, pp. 684–691.

[45] Intel, *Intel mkl*, https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html, 2022.

[46] M. Bollhöfer, O. Schenk, R. Janalik, S. Hamm, and K. Gullapalli, "State-of-the-art sparse direct solvers," in *Parallel Algorithms in Computational Science and Engineering*, Springer, 2020, pp. 3–33.

[47] J. Hogg and J. Scott, "An indefinite sparse direct solver for multicore machines," Tech. Rep. TR-RAL-2010-011, Rutherford Appleton Laboratory, Chilton, Oxfordshire, UK., Tech. Rep., 2010.

[48] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: An operator splitting solver for quadratic programs," *Mathematical Programming Computation*, vol. 12, no. 4, pp. 637–672, Dec. 2020.

[49] U. of Tennessee, B. University of California, U. of Colorado Denver, and N. Ltd., *ScaLAPACK - scalable linear algebra PACKage*, http://www.netlib.org/scalapack/.