Exploiting Parallelism in a QPALM-based Solver for Optimal Control

Pieter Pas*, Kristoffer Løwenstein†‡, Daniele Bernardini‡ and Panagiotis Patrinos*
*Department of Electrical Engineering (ESAT), KU Leuven, Kasteelpark Arenberg 10, 3001 Leuven, Belgium.
Email: {pieter.pas,panos.patrinos}@esat.kuleuven.be
†Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Via Ponzio 34/5, 20133, Milano, Italy.
Email: kristofferfink.loewenstein@polimi.it
‡ODYS S.r.l., Via Pastrengo 14, 20159, Milano, Italy.
Email: {kristoffer.lowenstein,daniele.bernardini}@odys.it

July 15, 2024

***Abstract*—We discuss the opportunities for parallelization in the recently proposed QPALM-OCP algorithm, a solver tailored to quadratic programs arising in optimal control. A significant part of the computational work can be carried out independently for the different stages in the optimal control problem. We exploit this specific structure to apply parallelization and vectorization techniques in an optimized C++ implementation of the method. Results for optimal control benchmark problems and comparisons to the original QPALM method are provided.**

I. Introduction

The need for the solution of linear-quadratic optimal control problems (OCPs) arises in popular applications such as linear model predictive control (MPC) and moving horizon estimation (MHE), as well as in sub-problems of solvers for their nonlinear counterparts. The real-time nature of these applications requires efficient quadratic programming (QP) solvers, that often need to perform well in constrained embedded environments. This work considers the parallelization of an optimized implementation of the recently proposed QPALM-OCP algorithm [6], which is a specialization of the augmented Lagrangian-based QPALM solver for (possibly nonconvex) quadratic programs [3]. The contribution in [6] is twofold: first, the QPALM algorithm is modified to handle linear equality constraints directly, and second, the specific structure of linear-quadratic optimal control problems (OCPs) is exploited in the solution of the linear systems in its inner semismooth Newton solver. Handling equality constraints directly reduces the overall number of iterations, and combined with more efficient linear solves in the inner solver, this can significantly reduce the solver run time.

This paper takes this one step further: it further explores the stage-wise structure of OCPs, and uses this to efficiently parallelize the algorithms described in [6]. Two levels of parallelism are used to take full advantage of the performance of modern hardware: (1) a compact storage format of the OCP matrices allows for efficient single instruction, multiple data (SIMD) operations across stages, and (2) OpenMP [8] is used to distribute the stages across multiple physical CPUs.

The remainder of this document is structured as follows: Section II introduces the optimal control problem formulation and notation; Section III summarizes the QPALM-OCP algorithm and highlights the opportunities for parallelization. Sections IV and V discuss data parallelism using vectorization and OpenMP parallelization. Section VI presents benchmark results, and Section VII concludes the paper with discussions of future work.

II. Problem formulation

We consider linear-quadratic optimal control problems:

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

The convex stage-wise and terminal cost functions are given by $\ell_j(x,u) = \frac{1}{2}\begin{pmatrix} x \\ u \end{pmatrix}^\top \begin{pmatrix} Q_j & S_j^\top \\ S_j & R_j \end{pmatrix} \begin{pmatrix} x \\ u \end{pmatrix} + \begin{pmatrix} x \\ u \end{pmatrix}^\top \begin{pmatrix} q^j \\ r^j \end{pmatrix}$ and $\ell_N(x) = \frac{1}{2}x^\top Q_N x + x^\top q^N$, with $Q_j \in \mathrm{Sym}_+(\mathbb{R}^{n_x})$, $S_j \in \mathbb{R}^{n_u \times n_x}$ and $R_j \in \mathrm{Sym}_+(\mathbb{R}^{n_u})$. The state variables $\mathbf{x} = [x^0 \cdots x^N] \in \mathbb{R}^{n_x \times (N+1)}$ and inputs $\mathbf{u} = [u^0 \cdots u^{N-1}] \in \mathbb{R}^{n_u \times N}$ 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 $c^j \in \mathbb{R}^{n_x}$. 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 $C_N \in \mathbb{R}^{n_{y^N} \times n_x}$ for the terminal constraints.

By interleaving the states and inputs into a vector $x \in \mathbb{R}^{N(n_x+n_u)+n_x}$ and introducing the matrices H, M and G, the OCP can be written as a quadratic program in standard form, with $n \triangleq N(n_x+n_u)+n_x$ variables, $p \triangleq (N+1)n_x$ equality constraints, and $m \triangleq Nn_y + n_{y^N}$ inequality constraints:

$$\begin{aligned} \underset{x}{\textbf{minimize}} \quad & \tfrac{1}{2}x^\top Q x + x^\top q \\ \textbf{subject to} \quad & Mx = b \\ & b_l \le Gx \le b_u \end{aligned} \tag{2}$$

$$Q \triangleq \begin{pmatrix} Q_0 & S_0^\top & & & & \\ S_0 & R_0 & & & & \\ & & Q_1 & S_1^\top & & \\ & & S_1 & R_1 & & \\ & & & & \ddots & \\ & & & & & Q_N \end{pmatrix},\ x \triangleq \begin{pmatrix} x^0 \\ u^0 \\ x^1 \\ u^1 \\ \vdots \\ x^N \end{pmatrix},\quad q \triangleq \begin{pmatrix} q^0 \\ r^0 \\ q^1 \\ r^1 \\ \vdots \\ q^N \end{pmatrix}$$

$$M = \begin{pmatrix} \mathrm{I} & & & & & & \\ -A_0 & -B_0 & \mathrm{I} & & & & \\ & & -A_1 & -B_1 & \mathrm{I} & & \\ & & & & & \ddots & \\ & & & & -A_{N-1} & -B_{N-1} & \mathrm{I} \end{pmatrix},\quad b \triangleq \begin{pmatrix} x_{\text{init}} \\ c^0 \\ c^1 \\ \vdots \\ c^{N-1} \end{pmatrix}$$

$$G = \begin{pmatrix} C_0 & D_0 & & & & \\ & & C_1 & D_1 & & \\ & & & & \ddots & \\ & & & & & C_N \end{pmatrix},\quad b_l = \begin{pmatrix} b_l^0 \\ b_l^1 \\ \vdots \\ b_l^N \end{pmatrix},\quad b_u = \begin{pmatrix} b_u^0 \\ b_u^1 \\ \vdots \\ b_u^N \end{pmatrix}$$

III. QPALM-OCP

The QPALM-OCP algorithm is described by [6, Sec. III], and further details can be found in the main QPALM publication [3]. This section briefly summarizes the inner semismooth Newton solver, which is responsible for the main computational cost of the algorithm, and which will be optimized in the remainder of this paper.

A. The augmented Lagrangian inner problem

QPALM-OCP solves problem (2) by relaxing the inequality constraints $b_l \leq Gx \leq b_u$ using an augmented Lagrangian method (ALM). The inner problem at outer iteration k of this method amounts to the minimization of a piecewise quadratic augmented Lagrangian function, subject to equality constraints,

$$\begin{aligned} \underset{x}{\textbf{minimize}} \quad & \phi_k(x) \\ \textbf{subject to} \quad & Mx = b, \end{aligned} \tag{3}$$

where $\phi_k(x) \triangleq \frac{1}{2}x^\top Qx + x^\top q + \frac{1}{2}\,\mathbf{dist}^2_{\Sigma_{y,k}}\left(Gx + \Sigma_{y,k}^{-1}y^k, C\right) + \frac{1}{2}\left\|x - x^k\right\|^2_{\Sigma_{x,k}^{-1}}$. In this expression, $y^k \in \mathbb{R}^m$ represents the current estimate of the Lagrange multipliers corresponding to the inequality constraints, $x^k \in \mathbb{R}^n$ is the current estimate of the primal variables, $\Sigma_{y,k} \in \mathrm{Sym}_{++}(\mathbb{R}^m)$ is a diagonal matrix containing the ALM penalty weights, the set $C = \{z \in \mathbb{R}^m \mid b_l \leq z \leq b_u\}$ describes the feasible set for Gx, and $\Sigma_{x,k} \in \mathrm{Sym}_{++}(\mathbb{R}^n)$ adds primal regularization through a proximal term.

Because of the squared distance, the function ϕ_k is not twice continuously differentiable. Therefore, QPALM applies a semismooth Newton method, with a generalized Hessian matrix of ϕ_k that is given by [6, Eq. 15]

$$H_k(x) = Q + G^\top \Sigma_{y,k}^{\mathcal{J}(x)} G, \tag{4}$$

where we defined the diagonal matrix $\Sigma_{y,k}^{\mathcal{J}(x)}$ as

$$\left(\Sigma_{y,k}^{\mathcal{J}(x)}\right)_{ii} \triangleq \begin{cases} 0 & \text{if } \left(Gx + \Sigma_{y,k}^{-1}y^k\right)_i \in [(b_l)_i, (b_u)_i], \\ \left(\Sigma_{y,k}\right)_{ii} & \text{otherwise.} \end{cases} \tag{5}$$

Therefore, the semismooth Newton direction $(\Delta x, \Delta\lambda)$ for inner problem (3) is the solution to

$$\begin{pmatrix} H_k(x) & M^\top \\ M & 0 \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta\lambda \end{pmatrix} = -\begin{pmatrix} \nabla\phi_k(x) + M^\top\lambda \\ Mx - b \end{pmatrix}. \tag{6}$$

B. Solving the semismooth Newton system

We use the approach described in [6, Sec. IV] to solve (6) in a way that preserves the block structure of the original OCP matrices Q, M and G:

$$\begin{aligned} H_k(x)v &= \nabla\phi_k(x) + M^\top\lambda, \\ -MH_k^{-1}(x)M^\top\Delta\lambda &= Mv - (Mx - b), \\ -H_k(x)\Delta x &= \nabla\phi_k(x) + M^\top(\lambda + \Delta\lambda). \end{aligned} \tag{7}$$

Thanks to the structure of the matrices in (2), the matrix $H_k(x)$ is block-diagonal, with blocks of the form $H_{k,j}(x) = \begin{pmatrix} Q_j & S_j^\top \\ S_j & R_j \end{pmatrix} + \begin{pmatrix} C_j^\top \\ D_j^\top \end{pmatrix} \Sigma_{y,k,j}^{\mathcal{J}(x)} \begin{pmatrix} C_j & D_j \end{pmatrix}$. This structure allows the evaluation and factorization of $H_{k,j}(x)$ to be carried out in parallel across stages. The Cholesky factorization $L_H L_H^\top$ of $H_{k,j}(x)$ can then be used to evaluate the intermediate matrices V_j and W_j that produce the matrix $\Psi_k(x) \triangleq MH_k^{-1}(x)M^\top$. $\Psi_k(x)$ is block-tridiagonal, with block-bidiagonal Cholesky factors consisting of $N+1$ diagonal blocks of size $n_x \times n_x$, and N coupling blocks between stages along the subdiagonal. For the sake of readability, we will from now on omit the dependence on x and the iteration number k. The following equation lists the stage-wise parallelizable operations and the corresponding BLAS/LAPACK operations:

$$\begin{aligned} H_j &= \begin{pmatrix} Q_j & S_j^\top \\ S_j & R_j \end{pmatrix} + \begin{pmatrix} C_j^\top \\ D_j^\top \end{pmatrix} \Sigma_{y,j}^{\mathcal{J}} \begin{pmatrix} C_j & D_j \end{pmatrix} && \texttt{(syrk)} \\ L_{H,j}L_{H,j}^\top &= H_j && \texttt{(potrf)} \\ V_j &= \begin{pmatrix} A_j & B_j \end{pmatrix} L_{H,j}^{-\top} && \texttt{(trsm)} \\ W_j &= \begin{pmatrix} \mathrm{I}_{n_x} & 0_{n_x \times n_u} \end{pmatrix} L_{H,j}^{-\top} && \texttt{(trtri,gemm,trsm)} \\ \Psi_{0,0} &= W_0 W_0^\top && \texttt{(syrk)} \\ \Psi_{j+1,j+1} &= V_j V_j^\top + W_{j+1}W_{j+1}^\top && \texttt{(syrk)} \\ \Psi_{j+1,j} &= -V_j W_j^\top && \texttt{(gemm)} \end{aligned}$$

Next, the factorization of Ψ is carried out recursively:

$$\begin{aligned} L_{\Psi,0,0}L_{\Psi,0,0}^\top &= \Psi_{0,0} && \texttt{(potrf)} \\ L_{\Psi,j+1,j} &= \Psi_{j+1,j}L_{\Psi,j,j}^{-\top} && \texttt{(trsm)} \\ \tilde{\Psi}_{j+1,j+1} &= \Psi_{j+1,j+1} - L_{\Psi,j+1,j}L_{\Psi,j+1,j}^\top && \texttt{(syrk)} \\ L_{\Psi,j+1,j+1}L_{\Psi,j+1,j+1}^\top &= \tilde{\Psi}_{j+1,j+1} && \texttt{(potrf)} \end{aligned}$$

Finally, the vectors v, $\Delta\lambda$ and Δx from (7) can be computed using forward and back substitution of L_H and L_Ψ.

All computations up to the factorization of Ψ are fully independent and will be parallelized in the following sections. The factorization of Ψ can still make use of the parallel nature of linear algebra operations inside of each $n_x \times n_x$ block.

IV. Vectorization

Many of the operations in QPALM-OCP can be expressed stage-wise. This includes part of the solution of (7) as discussed in the previous section, but also operations such as the matrix-vector product Mx, where one needs to evaluate the stage-wise products $A_j x^j$. These smaller products are fully independent and can be carried out in parallel. Since the same operation is applied to different matrices, we can use single instruction, multiple data (SIMD) programming techniques across the multiple stages.

A. Compact storage format

To enable this data parallelism across stages, the full OCP horizon N is subdivided into blocks with the same size as the hardware's vector length (2, 4 or 8 for modern CPUs), and all stages within a block are processed simultaneously. For example, using a vector length of 2, the products A_0x^0 and A_1x^1 are carried out at the same time, and so are the products A_2x^2 and A_3x^3. In order to perform such SIMD operations efficiently, the corresponding elements $(A_0)_{ij}$ and $(A_1)_{ij}$ need to be located at adjacent memory addresses. To this end, we store the matrices in a "compact" format, where matrices of different stages are interleaved. Figure 1 visualizes this format.

In our implementation of QPALM-OCP, all OCP matrices (A_j, B_j, C_j, Q_j, R_j, S_j) are stored in a compact format, and so are the corresponding vectors (x, Σ_y, y, etc.), in order to minimize the number of storage conversions required over the course of the algorithm.

When the horizon length N is not a multiple of the vector length, zero or identity matrices are added as appropriate, and the final stage is padded with $S^N = 0, R^N = \mathrm{I}, D^N = 0$.

B. Vectorized linear algebra routines for matrices in compact storage

Even though most linear algebra libraries will readily use SIMD instructions for ordinary matrices, using SIMD *across* multiple matrices rather than *within* a single matrix can result in higher performance, especially for smaller matrices [5]. For this reason, a selection of linear algebra routines that operate on batches of matrices in compact storage format are available in the Intel Math Kernel Library (MKL) [7]. In our benchmarks, these MKL routines resulted in some avoidable overhead, so we implemented some of these performance-critical routines ourselves. The architecture is roughly based on the BLIS framework [11] [10, Fig. 1], with multiple layers of loops around optimized GEMM and TRSM-GEMM micro-kernels. These micro-kernels keep a small $3\times3\times d$ or $5\times5\times d$ block of the result matrices in the CPU's vector registers as an accumulator, and perform some highly optimized multiply-accumulate operations. The loops around the micro-kernel perform blocking (and optionally packing) to optimize cache usage. The implementation is built on top of the experimental `std::simd` library [4] and makes use of C++ function templates for handling different block sizes and data types. The practical effectiveness of the storage format and the optimized linear algebra routines will be discussed in Section VI.

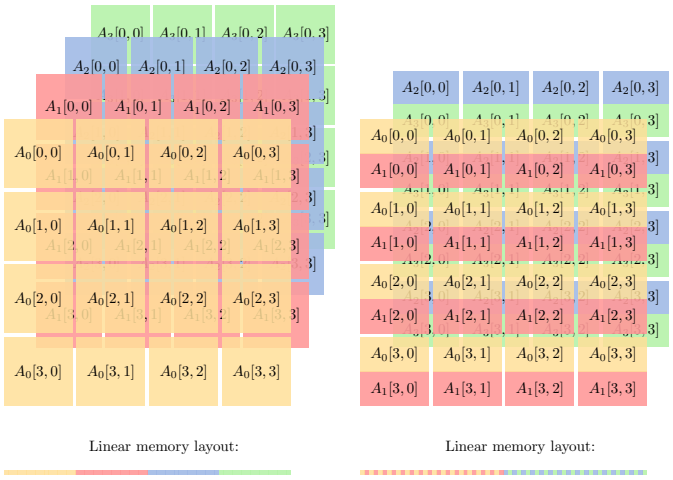

Fig. 1. Comparison between "naive" (left) and "compact" storage (right) of matrices A_j for a problem of size $N = 4 = n_x$ and a vector length of $d = 2$. In the naive format, each matrix A_j is stored contiguously in memory (e.g. column major order), and matrix A_{j+1} of the next stage is stored after A_j. In the compact format, matrices A_{2k} and A_{2k+1} are interleaved in such a way that their elements $(A_{2k})_{ij}$ and $(A_{2k+1})_{ij}$ are stored next to each other. Alternatively, the naive format can be seen as a $n_x \times n_x \times N$ tensor where the stage number $j < N$ is the index with the largest stride. In contrast, the compact format would be a $n_x \times n_x \times \lceil N/d \rceil$ tensor where each element is a tuple of d elements.

V. Parallelization

It is unlikely that the full horizon length N fits inside of a single vector register. Therefore we can further parallelize the $\lceil N/d \rceil$ blocks of stages introduced in the previous section by distributing them across the different CPUs in modern multi-core hardware. Frameworks like OpenMP [8] provide an easy way to express such parallelism.

VI. Numerical examples

This section reports the performance of a C++ implementation of QPALM-OCP using the optimizations described in the previous sections. Løwenstein et al. [6] used the classical spring-mass benchmark from [12] to compare the performance of their prototype implementation of QPALM-OCP to QPALM, OSQP and HPIPM. We keep QPALM as a baseline, and also include the recent PIQP solver [9].

Furthermore, we also investigate the effectiveness of our parallelization efforts, and conclude with four test problems from the `qpsolvers/mpc_qpbenchmark` repository [1].

A. Setup

All benchmarks were carried out using an octa-core Intel Core i7-11700 @ 2.50 GHz with dynamic CPU frequency scaling disabled for deterministic results. Native code was compiled using GCC 14.1 with AVX-512 support enabled at optimization level `-O3`. The absolute tolerances for all solvers are set to 10^{-8}, and the relative tolerances are set to zero.

B. Spring-mass benchmark

We use the same setup as described in [6, Sec. V]: The number of masses M is varied from 10 to 70, and the horizon

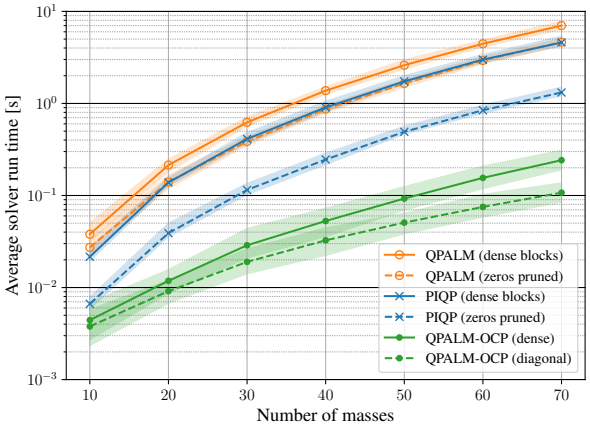

Fig. 2. Average solver run times for different numbers of masses, horizon $N = 15$. Shaded areas indicate best and worst run times.

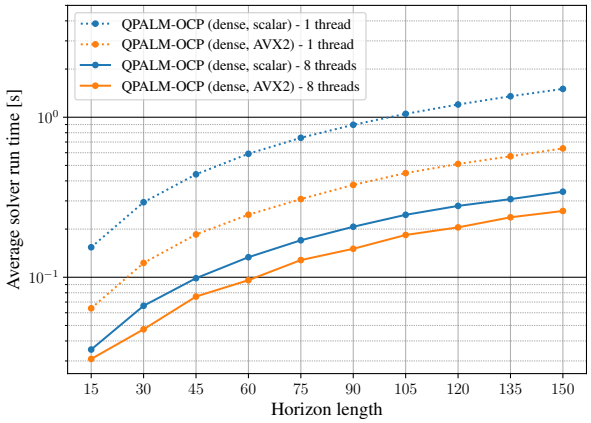

Fig. 3. Effect of parallelization and vectorization for different horizon length, masses $M = 30$.

is fixed at $N = 15$. For each configuration, we solve 100 OCPs with randomly generated initial conditions as in [6], and the geometric means of the solver run times are reported.

Since this specific problem has diagonal cost matrices Q_j and R_j, as well as diagonal constraint matrices $\begin{pmatrix} C_j & D_j \end{pmatrix}$ (box constraints), the resulting matrices $H_{k,j}(x)$ are diagonal as well, significantly simplifying the solution of (7). We therefore compare two different implementations of the method described in Section III-B: One general implementation where all problem data is represented by full, dense matrices, and one implementation where we exploit the diagonal structure of $H_{k,j}$ as much as possible.

For a fair comparison, we also include two versions of the sparse QPALM and PIQP solvers: one where we include all OCP matrices as dense blocks in the sparse matrix (even though they contain numerical zeros), and one where we eliminate all zeros from the sparse matrices. In the case of PIQP, we also provide the constraints as bounds on the variables rather than using general linear constraints for the second scenario.

The results of this experiment are shown in Figure 2. Both optimized versions of QPALM-OCP outperform the other solvers on this benchmark. For the largest problem (3275 primal variables), the dense version of QPALM-OCP is around 29 times faster than QPALM with dense blocks, and over 19 times faster than QPALM with all numerical zeros pruned. The diagonal version of QPALM-OCP is around 65 times faster than QPALM with dense blocks, and over 43 times faster than QPALM with all numerical zeros pruned.

C. Effectiveness of parallelization

In this experiment, we fix the number of masses to $M = 30$ and run four variants of the solver for different horizon lengths. The dense QPALM-OCP solver is used with both AVX2 vectorization (vector length 4), and without vectorization (no compact storage). These two configurations are repeated for a single OpenMP thread, and 8 OpenMP threads. The geometric means of the run times for 10 random instances of the spring-mass benchmark are reported. Figure 3 summarizes the results: In the single-threaded case, vectorization results in an overall speedup by a factor of around 2.3. By using multiple threads, the performance increases further, but it is limited by cache and bandwidth effects as well as the sequential parts such as the factorization of Ψ.

D. MPC qpbenchmark

Finally, we briefly report the performance on the four `QUADCMPC*` quadruped locomotion benchmarks from the `qpsolvers/mpc_qpbenchmark` repository [1] in Table I. Even though the problem is very sparse (yet not diagonal), the dense QPALM-OCP solver significantly outperforms the sparse QPALM solver. On the `LIPMWALK*` benchmarks, the performance difference is less pronounced, since these problems are very small ($n_x = 3, n_u = 1, N = 16$). With OpenMP disabled (it adds overhead for these small problems), QPALM-OCP achieves a shifted geometric mean run time of 0.43 ms, versus 0.46 ms for standard QPALM.

Problem	QPALM (pruned)	QPALM-OCP (dense)
`QUADCMPC1`	21.2 ms	5.1 ms
`QUADCMPC2`	9.9 ms	2.7 ms
`QUADCMPC3`	3.0 ms	2.0 ms
`QUADCMPC4`	3.5 ms	0.7 ms

TABLE I
Performance comparison between QPALM and QPALM-OCP on the `qpsolvers/mpc_qpbenchmark` `QUADCMPC*` problems.

VII. Conclusion

In this paper, we have shown that the recent QPALM-OCP method can effectively be parallelized, and that it outperforms other state-of-the-art solvers on a set of benchmark problems. Future work includes efficient offline packing of the matrix storage [2, 11], and the implementation of factorization update routines to avoid complete refactorizations when a small number of constraints or penalty factors change.

References

[1] Stéphane Caron, Akram Zaki, Pavel Otta, Daniel Arnström, and Justin Carpentier. qpbenchmark: Benchmark for quadratic programming solvers available in Python, January 2024. URL https://github.com/qpsolvers/qpbenchmark.

[2] Gianluca Frison, Dimitris Kouzoupis, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization. *ACM Trans. Math. Softw.*, 44(4), jul 2018. doi: 10.1145/3210754.

[3] Ben Hermans, Andreas Themelis, and Panagiotis Patrinos. QPALM: a proximal augmented Lagrangian method for nonconvex quadratic programs. *Mathematical Programming Computation*, 14(3):497–541, September 2022. doi: 10.1007/s12532-022-00218-0.

[4] Jared Hoberock. Working Draft, C++ Extensions for Parallelism Version 2. Technical Report N4808, ISO/IEC, March 2019.

[5] Intel® Math Kernel Library. Introducing Vectorized Compact Routines, August 2017. URL https://www.intel.com/content/www/us/en/developer/articles/technical/intelr-math-kernel-library-introducing-vectorized-compact-routines.html.

[6] Kristoffer Fink Løwenstein, Daniele Bernardini, and Panagiotis Patrinos. QPALM-OCP: A Newton-Type Proximal Augmented Lagrangian Solver Tailored for Quadratic Programs Arising in Model Predictive Control. *IEEE Control Systems Letters*, pages 1–1, 2024. doi: 10.1109/LCSYS.2024.3410638.

[7] Intel® oneAPI Math Kernel Library for C. Compact BLAS and LAPACK Functions, 2024. URL https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-2/compact-blas-and-lapack-functions.html.

[8] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 5.2, November 2021.

[9] Roland Schwan, Yuning Jiang, Daniel Kuhn, and Colin N. Jones. PIQP: A Proximal Interior-Point Quadratic Programming Solver. In *2023 62nd IEEE Conference on Decision and Control (CDC)*, pages 1088–1093, 2023.

[10] Field G. Van Zee and Tyler M. Smith. Implementing High-performance Complex Matrix Multiplication via the 3m and 4m Methods. *ACM Trans. Math. Softw.*, 44(1), July 2017. doi: 10.1145/3086466. URL https://doi.org/10.1145/3086466.

[11] Field G. Van Zee and Robert A. van de Geijn. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.*, 41(3), June 2015. URL https://doi.org/10.1145/2764454.

[12] Yang Wang and Stephen Boyd. Fast Model Predictive Control Using Online Optimization. *IFAC Proceedings Volumes*, 41(2):6974–6979, 2008. 17th IFAC World Congress.

Acknowledgments

This work was supported by Fonds Wetenschappelijk Onderzoek (FWO) PhD grant 11M9523N; FWO research projects G081222N, G033822N and G0A0920N; the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 953348; and EuroHPC Project 101118139 Inno4Scale.