Speed-Up of Nonlinear Model Predictive Control for Robot Manipulators Using Task and Data Parallelism

Alejandro Astudillo, Joris Gillis, Goele Pipeleers, Wilm Decré and Jan Swevers
MECO Research Team, Department of Mechanical Engineering, KU Leuven, Belgium
DMMS lab, Flanders Make, Leuven, Belgium
{alejandro.astudillovigoya, joris.gillis, goele.pipeleers, wilm.decre, jan.swevers}@kuleuven.be

***Abstract*—The repetitive evaluation of computationally expensive functions in the objective and constraints represents a bottleneck in the solution of the underlying optimal control problem (OCP) of nonlinear model predictive controllers (MPC) for robot manipulators. We address this problem by exploiting the parallel evaluation of such functions within the execution of a first-order and a second-order OCP solution algorithm, such as the proximal averaged Newton-type method for optimal control (PANOC) and the sequential convex quadratic programming (SCQP) method, respectively. The use of task parallelism with multicore executions and data parallelism with single-instruction-multiple-data (SIMD) instructions is shown to effectively reduce the solution time of the underlying OCP so that the satisfaction of real-time constraints in the deployment of MPC for robot manipulators can be achieved.**

***Index Terms*—Robot manipulator, model predictive control, parallelization, vectorization, tunnel following**

I. Introduction

The deployment of advanced motion controllers for nonlinear systems requires the satisfaction of real-time constraints. For model predictive controllers (MPC), this implies that the underlying optimal control problem (OCP) must be solved in a time shorter than the sampling time of the control system. For highly nonlinear systems, however, the evaluation of expressions for dynamics and kinematics in the OCP objective or constraints is computationally expensive and may represent a bottleneck in the solution time of the OCP. This is the case for nonlinear MPC of robot manipulators, where the forward dynamics of the robot are imposed as equality constraints when torques are set as control inputs.

After transcribing the OCP into a nonlinear program (NLP), efficient algorithms such as the proximal averaged Newton-type method for optimal control (PANOC) [1] and the sequential convex quadratic programming (SCQP) method [2] can be used to solve such NLP in a shorter time compared to the widely used interior point (IP) method solver IPOPT [3] or the sequential quadratic programming (SQP) method with exact Hessian of the Lagrangian [4]. Nevertheless, the bottleneck in the evaluation of expressions for dynamics and kinematics of the robot manipulator is still present in such solution algorithms.

A. Approach and Contributions

The use of a prediction horizon with N time steps in the MPC setting involves numerous evaluations of functions in the objective and constraints of the OCP. In fact, every function in (i) the stage objective, i.e., the part of the objective that is added at each time step, and (ii) the path constraints, i.e., constraints that must be satisfied along the whole prediction horizon, is present N times in the objective (or constraints) function and must be evaluated the same amount of times for different arguments or inputs.

In this paper, we propose to exploit the repetitive presence of functions in the OCP objective and constraints to parallelize their evaluation and, consequently, to reduce the OCP solution time. Such parallelization is included within the automatic code generation feature in the numerical optimization framework CasADi [5]. This is done without modifying the NLP solution algorithms themselves nor their convergence properties. We explore task parallelism, i.e., multi-core parallelization, and data parallelism, i.e., single-instruction-multiple-data (SIMD), to efficiently evaluate multiple instances of the robot dynamics and kinematics in a tunnel-following MPC [6], [7] for a robot manipulator, where the underlying OCP is solved by using (i) the first-order method PANOC with augmented Lagrangian, and (ii) the second-order method SCQP. The effect of parallelization of function evaluations on both methods is also compared.

B. Notation

We denote $\langle v_1, \cdots, v_n \rangle := \begin{bmatrix} v_1^\top & \cdots & v_n^\top \end{bmatrix}^\top$ as the concatenation of multiple column vectors v_i, $i \subset \{1, ..., n\}$, $||v||_\mathbf{P} := \sqrt{v^\top \mathbf{P} v}$ as the weighted ℓ-2 norm of v, and $\mathbf{0}$ as the zero matrix.

C. Outline

The remainder of the paper is organized as follows. Section II introduces the tunnel-following MPC problem and the algorithms used to solve it. In Section III, the methodology for parallelizing the functions evaluation is presented. Numerical experiments are reported in Section IV. Section V finally details some concluding remarks and proposes future work.

*This work has been carried out within the framework of the Flanders Make SBO project MULTIROB: "Rigorous approach for programming and optimal control of multi-robot systems" and the FWO project G0A6917N of the Research Foundation - Flanders (FWO - Flanders). Flanders Make is the Flemish strategic research centre for the manufacturing industry.

II. Problem Formulation

This section briefly introduces the tunnel-following NMPC scheme, presented by van Duijkeren [6], which is the motion control scheme used in this study. We present the dynamics of the robot and the constraints imposed over it using the multiple-shooting method, and then introduce the two NLP solution algorithms that are evaluated.

A. System Dynamics

Robot manipulators are chains of rigid bodies connected by joints, which constrain their movement with respect to their neighbouring bodies. For a manipulator with n_{dof} actuated joints, i.e., n_{dof} degrees of freedom (dof), its dynamics can be expressed by means of rigid body dynamics as [8]

$$M(\mathbf{q})\ddot{\mathbf{q}} + C(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}} + G(\mathbf{q}) = \boldsymbol{\tau} + \mathbf{J_c}(\mathbf{q})^\top \mathbf{f}^{\text{c}}, \tag{1}$$

where $M \in \mathbb{R}^{n_{\text{dof}} \times n_{\text{dof}}}$, $C \in \mathbb{R}^{n_{\text{dof}} \times n_{\text{dof}}}$ and $G \in \mathbb{R}^{n_{\text{dof}}}$ are the joint-space inertia, Coriolis and gravity matrices, respectively, $\mathbf{q} \in \mathbb{R}^{n_{\text{dof}}}$, $\dot{\mathbf{q}} \in \mathbb{R}^{n_{\text{dof}}}$, $\ddot{\mathbf{q}} \in \mathbb{R}^{n_{\text{dof}}}$ are the generalized joint position, velocity and acceleration vectors, $\boldsymbol{\tau} \in \mathbb{R}^{n_{\text{dof}}}$ is the generalized joint torque vector, $\mathbf{J_c} \in \mathbb{R}^{6 \times n_{\text{dof}}}$ is the contact Jacobian, and $\mathbf{f}^{\text{c}} \in \mathbb{R}^6$ is the stack of external forces applied to the robot. Solving (1) for $\ddot{\mathbf{q}}$, assuming there are no external forces affecting the robot, i.e., $\mathbf{f}^{\text{c}} = \mathbf{0}$, defines the forward dynamics (FD) function

$$\ddot{\mathbf{q}} = M^{-1}(\boldsymbol{\tau} - C\dot{\mathbf{q}} - G) =: \text{FD}(\mathbf{q}, \dot{\mathbf{q}}, \boldsymbol{\tau}). \tag{2}$$

which can be efficiently computed by using the articulated-body algorithm [9].

Remark 1: The forward dynamics function FD is highly nonlinear and its evaluation is computationally expensive.

Let us now introduce the state vector $x := \langle \mathbf{q}, \dot{\mathbf{q}} \rangle \in \mathbb{R}^{2n_{\text{dof}}}$ and the input vector $u := \boldsymbol{\tau} \in \mathbb{R}^{n_{\text{dof}}}$. Based on the definition of FD, the ordinary differential equation describing the nonlinear dynamics of the robot is defined as

$$\dot{x} = \langle \dot{\mathbf{q}}, \text{FD}(\mathbf{q}, \dot{\mathbf{q}}, \boldsymbol{\tau}) \rangle =: \xi(x, u). \tag{3}$$

For path-following-like motion control, it is useful to introduce virtual states and inputs that augment the system dynamics, to allow the optimizer to decide not only where the end-effector of the robot must be, but also when to be there [10]. Therefore, we define the path parameter $\mathbf{s} \in \mathcal{T} := [0, 1]$, which is assumed to have double-integrator dynamics [6]. These dynamics are used to augment the system dynamics ξ as

$$\dot{\hat{x}} = \langle \dot{\mathbf{q}}, \text{FD}(\mathbf{q}, \dot{\mathbf{q}}, \boldsymbol{\tau}), \dot{\mathbf{s}}, \ddot{\mathbf{s}} \rangle =: \hat{\xi}(\hat{x}, \hat{u}), \tag{4}$$

where $\hat{x} := \langle \mathbf{q}, \dot{\mathbf{q}}, \mathbf{s}, \dot{\mathbf{s}} \rangle \in \mathbb{R}^{2n_{\text{dof}}+2}$ and $\hat{u} := \langle \boldsymbol{\tau}, \ddot{\mathbf{s}} \rangle \in \mathbb{R}^{n_{\text{dof}}+1}$.

B. Multiple Shooting

The continuous dynamics (4) can be discretized for a sampling time $\delta_t \in \mathbb{R}_{>0}$ by means of a numerical integrator such as the Runge-Kutta fourth-order method described below

$$\begin{aligned} &k_1 = \hat{\xi}(\hat{x}_k, \hat{u}_k), && k_2 = \hat{\xi}(\hat{x}_k + 0.5\delta_t k_1, \hat{u}_k), \\ &k_3 = \hat{\xi}(\hat{x}_k + 0.5\delta_t k_2, \hat{u}_k), && k_4 = \hat{\xi}(\hat{x}_k + \delta_t k_3, \hat{u}_k), \\ &\hat{x}_{k+1} = \hat{x}_k + \frac{\delta_t}{6}(k_1 + 2k_2 + 2k_3 + k_4) =: \xi_d(\hat{x}_k, \hat{u}_k). \end{aligned} \tag{5}$$

Instead of discretizing (4) as a single simulation along the prediction horizon $N \in \mathbb{N}^+$ of an OCP, which would worsen the nonlinearity of the already highly nonlinear robot dynamics, such discretization can be performed over N intervals by using the multiple-shooting method [11]. This is a direct method that introduces grid state variables $X_g := \{\hat{x}_k \ : \ k \in [0, N]\}$ and grid input variables $U_g := \{\hat{u}_k \ : \ k \in [0, N-1]\}$ as decision variables of the OCP. The trajectory of $\hat{x}_k$ at the end of each interval k will not necessarily coincide with the trajectory of $\hat{x}_{k+1}$ at the start of the interval $k+1$. This discontinuity is known as the multiple-shooting gap or defect, and defines the multiple-shooting equality constraints

$$\hat{x}_{k+1} - \xi_d(\hat{x}_k, \hat{u}_k) = \mathbf{0}, \quad k = 0, \ldots, N-1. \tag{6}$$

The grid variables X_g and U_g allow the formulation of N independent dynamic equality constraints (6), one per discretization interval. This enables the parallel evaluation of the set of multiple-shooting constraints which, given the computational complexity of FD, will contribute to the reduction of the OCP solution time.

C. Tunnel-Following Nonlinear MPC

Unlike path-following MPC, tunnel-following MPC [6], [7] does not require the position $p_{\text{ee}}(\mathbf{q}) \in \mathbb{R}^3$ of the end-effector of a robot manipulator to follow a Cartesian geometric path $p_{\text{ref}}(\mathbf{s}) \in \mathbb{R}^3$ exactly. Instead, it allows a certain deviation from the path within a ρ-neighborhood, i.e., a sphere of radius $\rho \in \mathbb{R}_{\geq 0}$, for every time instant. The union of all ρ-neighborhoods forms a tunnel around the reference path p_{ref} (see Fig. 1).

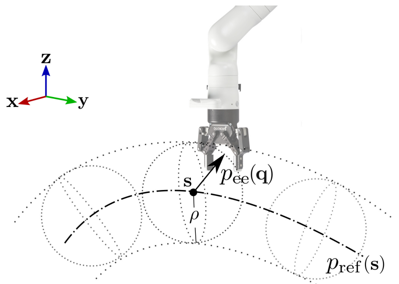

Fig. 1. Illustration of spheres of radius ρ (ρ-neighborhood) which form the tunnel around the reference path p_{ref}.

The tunnel-following MPC scheme presented in [6] depends on the following assumption.

Assumption 1: A position reference $p_{\text{ref}}(\mathbf{s})$ and a path-velocity reference $\dot{\mathbf{s}}_{\text{ref}}(\mathbf{s})$ are known a priori, while the end-effector position $p_{\text{ee}}(\mathbf{q})$ is obtained from the forward kinematics of the robot.

Tunnel-following MPC imposes a path constraint that sets an upper bound ρ^2 to the squared ℓ-2 norm of the position error $e_{\mathcal{P}}(\mathbf{q}, \mathbf{s}) := p_{\text{ee}}(\mathbf{q}) - p_{\text{ref}}(\mathbf{s})$, i.e., imposes a maximum distance ρ between p_{ee} and p_{ref}. This constraint, however, can be relaxed by means of a slack variable $l_k \in \mathbb{R}_{\geq 0}$ to guarantee the feasibility of the solution of the underlying OCP when the end-effector cannot stay within the ρ-neighborhood. Thus, the tunnel-following constraint is defined as

$$||e_{\mathcal{P}}(\mathbf{q}_k, \mathbf{s}_k)||^2 - l_k \leq \rho^2, \quad k = 0, \dots, N-1. \tag{7}$$

Let us now define the underlying OCP of the tunnel-following MPC scheme as follows

$$\begin{aligned}
\min_{w \in W} \quad & V_N(\hat{x}_N) + \sum_{k=0}^{N-1} \left[V(\hat{x}_k, \hat{u}_k) + \alpha l_k \right] && \text{(8a)}\\
\text{s.t.} \quad & \hat{x}_0 - \chi = \mathbf{0}, && \text{(8b)}\\
& \text{(6)}, && \text{(8c)}\\
& \text{(7)}, && \text{(8d)}\\
& \zeta(\hat{x}_k, \hat{u}_k) \leq \mathbf{0}, \quad k = 0, \dots, N-1, && \text{(8e)}
\end{aligned}$$

where $w := \langle \hat{x}_0, \hat{u}_0, l_0, \cdots \hat{u}_{N-1}, l_{N-1}, \hat{x}_N \rangle$ is the vector of decision variables in the set $W := \{w : w_{\min} \leq w \leq w_{\max}\}$,

$$V_N(\hat{x}_N) := || \langle e_{\mathcal{P}}(\mathbf{q}_N, \mathbf{s}_N), \mathbf{s}_N - 1, \hat{x}_N \rangle ||^2_{\mathbf{P}} \tag{9}$$

is the terminal cost with weight matrix $\mathbf{P} \succeq \mathbf{0}$,

$$V(\hat{x}_k, \hat{u}_k) := || \langle e_{\dot{\mathbf{s}}}(\mathbf{s}_k, \dot{\mathbf{s}}_k), e_{\mathcal{P}}(\mathbf{q}_k, \mathbf{s}_k), \hat{x}_k, \hat{u}_k \rangle ||^2_{\mathbf{Q}} \tag{10}$$

is the stage cost with weight matrix $\mathbf{Q} \succeq \mathbf{0}$, $e_{\dot{\mathbf{s}}}(\mathbf{s}_k, \dot{\mathbf{s}}_k) := \dot{\mathbf{s}}_k - \dot{\mathbf{s}}_{\text{ref}}(\mathbf{s}_k)$ is the time-tracking error, l_k are slack variables that are heavily penalized with a weight $\alpha \in \mathbb{R}_{\gg 0}$, χ is an estimate of the state vector, and $\zeta(\hat{x}_k, \hat{u}_k)$ are general path constraints. Here, the time-tracking error $e_{\dot{\mathbf{s}}}(\mathbf{s}_k, \dot{\mathbf{s}}_k)$ is heavily penalized in V with weight $w_{\dot{\mathbf{s}}} \gg 0$ to make the system follow the path-velocity reference $\dot{\mathbf{s}}_{\text{ref}}(\mathbf{s})$ with minimum error, while the other elements in V are regularization terms with weight $w_r : 0 < w_r \ll 1$. The term $\mathbf{s}_N - 1$ is penalized in the terminal cost with weight w_r to create attraction of the state $\mathbf{s}$ to its maximum value $\mathbf{s}_{\max} = 1$.

D. Solution of the Optimal Control Problem

Let us now briefly introduce the two NLP solution algorithms that will be evaluated in this study.

1) PANOC with augmented Lagrangian: Following a direct optimal control method for parametric optimization, OCP (8) can be transcribed into an NLP of the type

$$\begin{aligned}
\min_{w \in W} \quad & f(w) && \text{(11a)}\\
\text{s.t.} \quad & g(w, \chi) \in \mathcal{Z}. && \text{(11b)}
\end{aligned}$$

where $f(w)$ and $g(w, \chi)$ are nonconvex smooth functions, and $\mathcal{Z}$ is a closed convex set. This NLP can be solved by using a first-order method like the proximal averaged Newton-type method for optimal control (PANOC) with augmented Lagrangian [1], [12]. This method solves an inner optimization problem with PANOC and applies the augmented Lagrangian method with an outer iterative procedure.

The NLP (11) is casted into the inner optimization problem

$$\min_{w \in W} \quad f(w) + \frac{\varsigma}{2} \mathbf{dist}^2_{\mathcal{Z}}(g(w, \chi) + \varsigma^{-1}\lambda), \tag{12}$$

where $\mathbf{dist}_{\mathcal{Z}}(v) := \mathbf{inf}_{\beta \in \mathcal{Z}} ||\beta - v||$ is the distance between a vector v and the set $\mathcal{Z}$, λ is the vector of Lagrange multipliers corresponding to $g(w, \chi)$, and $\varsigma \in \mathbb{R}_{>0}$ is a penalty parameter. The problem (12) is then solved by PANOC using the projected gradient operator and the forward-backward envelope [13]. The Lagrange multipliers λ and the penalty parameter ς are updated by the outer iterative procedure. The interested reader is referred to [1] for additional information on PANOC and to [12] for more information on the augmented Lagrangian method used in the outer iterative procedure.

2) SCQP: Let us introduce a compressed representation of OCP (8) as the following NLP

$$\begin{aligned}
\min_{w} \quad & \phi_0(c_0(w)) && \text{(13a)}\\
\text{s.t.} \quad & h_i(w, \chi) = \mathbf{0}, \quad i = 1, ..., n_h, && \text{(13b)}\\
& \phi_i(c_i(w)) \leq \mathbf{0}, \quad i = 1, ..., n_g, && \text{(13c)}
\end{aligned}$$

where $\phi_i(c_i(w))$, $i \subset \{0, \dots, n_g\}$, are *convex-over-nonlinear* functions, with a convex outer part $\phi_i(c)$ and a nonlinear inner part $c_i(w)$. The sequential convex quadratic programming (SCQP) method [2] is a variation of the SQP method applied to NLPs of the type (13), where the exact Hessian of the Lagrangian B^{SQP} is replaced, for every QP subproblem, by an approximation B^{SCQP} that (i) ignores the second-order derivatives of inner nonlinear functions $c_i(w)$ in the objective and constraints, (ii) is cheaper to evaluate, and (iii) is guaranteed to be positive semidefinite. Such approximation is computed as

$$\begin{aligned}
B_k^{\text{SCQP}}(w, \mu) := & \frac{\partial c_0}{\partial w}(w)^\top \nabla_c^2 \phi_0(c_0(w)) \frac{\partial c_0}{\partial w}(w) \\
& + \sum_{i=1}^{n_g} \mu_i \frac{\partial c_i}{\partial w}(w)^\top \nabla_c^2 \phi_i(c_i(w)) \frac{\partial c_i}{\partial w}(w), \quad (14)
\end{aligned}$$

where μ_i are the Lagrange multipliers corresponding to the inequality constraints (13c). For a full discussion on the SCQP method, the reader is referred to [2].

III. Parallelization of Functions Evaluation

In this section, we introduce the methodology used to parallelize the evaluation of functions in the objective and constraints within the solution of the OCP (8).

Since the multiple instances of expressions $V(\hat{x}_k, \hat{u}_k)$ and l_k in (8a), $\hat{x}_{k+1} - \xi_d(\hat{x}_k, \hat{u}_k)$ in (8c), and $||e_{\mathcal{P}}(\mathbf{q}_k, \mathbf{s}_k)||^2 - l_k$ in (8d) are independent from each other along the prediction horizon, the optimization framework used to construct and solve OCP (8) can be instructed to parallelize the evaluation of such instances in up to N processing units, instead of evaluating them sequentially. In this work, we use CasADi [5] as numerical optimization and algorithmic differentiation framework (i) to define the expressions needed to construct the OCP (8), (ii) to differentiate such expressions by means of algorithmic differentiation, and (iii) to generate self-contained C-code of the expressions and their derivatives, which can then be compiled for efficient evaluation.

A. Task parallelism

Many computers today have powerful processors with multiple cores, which can be leveraged to execute multiple tasks concurrently using task parallelism. This form of parallelization executes multiple functions (or multiple instances of one function) simultaneously in separate cores.

CasADi implements a *map* construct that can be applied to expressions that need to be evaluated multiple times with different arguments. To perform such evaluations in multiple cores, the *map* construct makes use of the Open Multi-Processing API (OpenMP) to instruct task parallelization to such expressions and to extend such instruction to the evaluation of their derivatives. For a CasADi function F that is needed to be evaluated N times, task parallelism is instructed by using the command `F = F.map(N, "openmp")`.

B. Data parallelism

Additional to task parallelism, some modern processors allow an additional type of parallelization within each of their cores. This type of parallelization is known as data parallelism, vectorization or single-instruction-multiple-data (SIMD), and relies on internal processing units within a core to evaluate the same expression or function on multiple data arguments simultaneously. This indicates that data parallelism can be used to parallelize the evaluation of functions in OCP (8).

Data parallelism is generally applied in modern computers by means of SIMD instructions sets. These instruction sets are, in turn, based on the so-called AVX registers, which can perform simultaneous operations on multiple data. These registers can hold 128 bits, 256 bits, or 512 bits of data depending on the SIMD instruction set available for the processor (SSE4, AVX2 or AVX512, respectively).

Modern compilers, e.g., GCC and ICC, automatically identify expressions that are suitable for vectorization in the generated C-code [14]. However, many of such expressions are ignored by the compiler since their data arguments lack two conditions that allow vectorization: (i) data structure alignment and (ii) space locality of data. Data structure alignment refers to the placement in memory of each argument of an expression such that its base address is a multiple of the AVX register size. In addition, preserving the spatial locality of the data refers to locating the initial element of successive data arrays in contiguous memory locations, i.e., preserving a unit-stride.

We have explicitly dealt with the satisfaction of the two aforementioned conditions by (i) inserting padding to the data argument structures to impose an element size equal to the size of the AVX register used, and (ii) reordering the arguments of expressions to preserve unit-stride during execution. Without these implementations, automatic vectorization in the compiler would not recognize some functions as parallelizable with AVX2 or AVX512 instructions, and would rely mainly on SSE4 instructions to vectorize operations were possible.

The *map* construct in CasADi also allows to evaluate an expression multiple times in a serial or sequential manner, i.e., using just one core. This can be instructed by using the command `F = F.map(N, "serial")`.

However, we have extended the serial option of *map* to use OpenMP to instruct the use of SIMD in the evaluation of multiple instances of an expression within one core.

IV. Numerical Example

The proposed methodology has been implemented in CasADi[1] and tested in a simulated environment. The tests were executed on a PC running Ubuntu 18.04 with a 10-cores Intel i9-9900X CPU. This CPU is compatible with the SSE4, AVX2 and AVX512 instruction sets. All code-generated functions are compiled using GCC 9.1.0 with compilation flags `-O3`, `-march=native` and `-fopenmp` unless stated otherwise.

Efficient CasADi functions for forward dynamics FD and kinematics p_{ee} of the robot were generated with the interface [15] of the rigid-body dynamics library Pinocchio [16].

The tunnel-following nonlinear MPC is applied to a 7-dof *Kinova Gen3* robot following a lemniscate-shaped path. We use a prediction horizon of $N = 16$, a sampling time $\delta_t = 0.005\ s$, a tunnel radius $\rho = 0.01\ m$, a maximum $\dot{\mathbf{s}}_{\mathrm{ref}}(\mathbf{s})$ of 0.1, and weights $\alpha = 100$, $w_r = 10^{-3}$, and $w_{\dot{\mathbf{s}}} = 10$.

A. Evaluation of robot dynamics

Let us first evaluate the effects of parallelization on one of the most computationally expensive functions used to define OCP (8), the multiple-shooting constraints (6). As already mention in Section II, by introducing grid state and input variables the multiple-shooting method defines $N = 16$ independent instances of the expression $\hat{x}_{k+1} - \xi_d(\hat{x}_k, \hat{u}_k)$, which must be evaluated for 16 pairs of arguments $(\hat{x}_k, \hat{u}_k)$.

Remark 2: As baseline for comparison, we execute the generated code without the explicit instruction of any type of parallelization with the *map* function, and without assisting the satisfaction of data structure alignment and space locality of data. This baseline is labeled as NO-PAR hereafter.

In Table I, we show the average evaluation time of the group of 16 instances of (6) instructed to be evaluated using (i) task parallelism with 2, 4 and 8 cores, and (ii) data parallelism using the compilation flag `-mprefer-vector-width=SIZE`, with `SIZE`=128 for SSE4, `SIZE`=256 for AVX2 and `SIZE`=512 for AVX512 vectorization.

The evaluation of FD (2) requires evaluating the sine and cosine of the joint positions $\mathbf{q}_j,\ j \subset \{1, ..., n_{\mathrm{dof}}\}$. Due to the compilation flag `-O3`, this evaluation is transformed by the compiler into a call to the $\mathrm{sincos}(\mathbf{q}_j)$ subroutine, which performs the operations $\sin(\mathbf{q}_j)$ and $\cos(\mathbf{q}_j)$ simultaneously. However, the sincos subroutine is not automatically vectorized by the GCC compiler. Therefore, for data parallelism we also use the compilation flag `-fdisable-tree-sincos` to disable the conversion of $\{\sin(\mathbf{q}_j),\ \cos(\mathbf{q}_j)\}$ into a call to $\mathrm{sincos}(\mathbf{q}_j)$.

As shown in Table I, data parallelism speeds up the evaluation of the instances of (6) by a factor greater and closer to the theoretical speedup than the actual speed-up achieved with task parallelism. An explanation for the

[1]Source code available at *https://github.com/casadi/casadi/tree/vectorize3*

TABLE I
AVERAGE EVALUATION TIME OF A GROUP $N = 16$ INSTANCES OF THE MULTIPLE-SHOOTING CONSTRAINTS (6) FROM 10^6 SAMPLES.

Parallelization method	Theoretical speed-up	Evaluation time (Actual speed-up)
NO-PAR	–	$25.102\ \mu s$ $(-)$
2 cores	2	$14.345\ \mu s$ (1.75)
4 cores	4	$8.995\ \mu s$ (2.79)
8 cores	8	$6.256\ \mu s$ (4.01)
SSE4	2	$14.197\ \mu s$ (1.77)
AVX2	4	$6.873\ \mu s$ (3.65)
AVX512	8	$3.761\ \mu s$ (6.67)

difference between actual and theoretical speed-up would involve the presence of communication overhead in task-parallel executions and the fact that SIMD instructions are affected by frequency scaling or downclocking, i.e., the maximum operation frequency of the core is reduced while executing SIMD instructions [17]. The communication overhead in task-parallel execution would also explain the poor performance achieved with 4 and 8 cores, since such overhead cannot be neglected for executions in the order of the microseconds.

B. Solution of the optimal control problem

As we mentioned previously in Section III, we instructed CasADi to parallelize the N independent instances of the expressions $V(\hat{x}_k, \hat{u}_k)$ and l_k in (8a), $\hat{x}_{k+1} - \xi_d(\hat{x}_k, \hat{u}_k)$ in (8c), and $||e_{\mathcal{P}}(\mathbf{q}_k, \mathbf{s}_k)||^2 - l_k$ in (8d). In OCP (8), the terminal cost $V_N(\hat{x}_N)$ in (8a) and the initial state constraint (8b) cannot be parallelized. However, they are not as computationally expensive as the functions that are instructed to be parallelized and we assume that the effect of not evaluating these functions in parallel is negligible.

According to Amdahl's law [18] applied to parallel computing, the theoretical speed-up that can be achieved in a process with parallel executions is upper-bounded by the percentage of the process that cannot be parallelized. Therefore, let us first compare the NLP solution methods described in Section II (PANOC and SCQP), besides the widely used SQP and IP methods, in terms of (i) the percentage of time required to evaluate the functions in the OCP, and (ii) the average multiple-shooting constraint violation along the execution of the tunnel-following task. For this test, we use the NO-PAR baseline with no explicit parallel instructions. Only one quadratic subprogram is solved for every SCQP execution, following the real-time iteration scheme. Note that the IP solver IPOPT is not suitable for code-generation. Therefore, the results shown for solver evaluation time in IP are not obtained using the compilation flags mentioned above.

Table II presents the results of the comparison of the NLP solution methods. The data in this table confirms that PANOC and SCQP have a lower solution time compared to the SQP method with exact Hessian and the IP method, being PANOC the method that would benefit the most from parallel evaluation of the functions, since a greater percentage of its solution time is spent on functions evaluation. Nevertheless, one major drawback of PANOC is the average multiple-shooting constraint violation of 4.206×10^{-3}. This indicates that multiple-shooting constraints are not properly satisfied with PANOC for this highly nonlinear system, which restricts its implementation on a real robot.

The resulting path of the end-effector of the *Kinova Gen3* robot achieved during the simulated task is shown in Fig. 2.

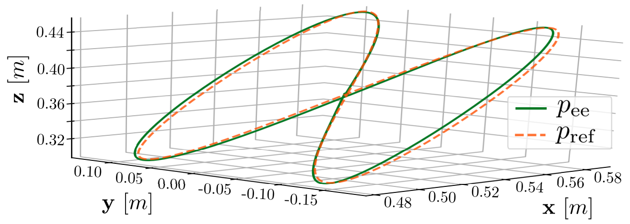

Fig. 2. Comparison of the path-followed by the end-effector p_{ee} and the reference path p_{ref} during the tunnel-following task solved with SCQP. The results obtained with PANOC, SQP and IP are not shown for the sake of simplicity since they overlap the resulting p_{ee} of SCQP.

Figure 3 shows the excursion of the ℓ-2 norm of the position error along the simulated task using PANOC and SCQP. What can be clearly seen in this figure is the satisfaction of the tunnel-constraints $\forall\ \mathbf{s} \in \mathcal{T}$ without requiring the slack variable l_k to be greater than zero. The slack variable l_k would become important for experiments with a lower tunnel radius ρ and a greater maximum path-velocity reference $\dot{\mathbf{s}}_{\text{ref}}$. However, PANOC could not solve OCP (8) with such modified values.

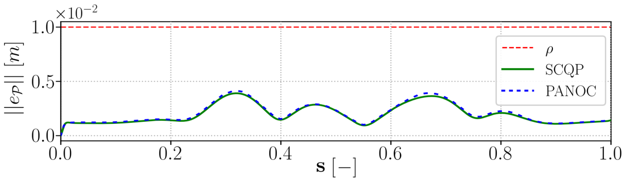

Fig. 3. Excursion of the ℓ-2 norm of $e_{\mathcal{P}}$ along the tunnel-following execution. Errors obtained with IP and SQP are not shown in the figure for the sake of simplicity since their difference with respect to the plots already shown is negligible.

Let us now evaluate the application of task and data parallelism on the evaluation of functions in PANOC and SCQP. Figure 4 illustrates the different solution times achieved for PANOC and SCQP with (i) no parallelization of the functions evaluation (NO-PAR), (ii) task parallelization with 2 cores, 4 cores and 8 cores, and (iii) data parallelization with SSE4, AVX2 and AVX512 instructions.

What stands out from this figure is the speed-up achieved with task parallelism on PANOC, with a maximum of 3.29 by using 8 cores. This speed-up, however, may have been increased by automatic vectorization (with SSE4) of the code executed in each core. The SCQP method also benefits from task parallelism, but to a lesser extent, as anticipated from Table II. The speed-up achieved with 4 and 8 cores is also

TABLE II

COMPARISON OF THE NLP SOLUTION METHODS APPLIED TO THE TUNNEL-FOLLOWING MPC PROBLEM WITHOUT EXPLICIT PARALLELIZATION

Solution method	Solution time	Function evaluation time	Solver evaluation time	Average of the multiple-shooting constraint violation
PANOC	$\mathbf{5.079\ ms}$	$4.627\ ms$ (91.11%)	$0.452\ ms$ (8.89%)	$\mathbf{4.206 \times 10^{-3}}$
SCQP	$\mathbf{5.157\ ms}$	$2.344\ ms$ (45.46%)	$2.813\ ms$ (54.54%)	3.217×10^{-6}
SQP	$25.140\ ms$	$22.293\ ms$ (88.68%)	$2.847\ ms$ (11.32%)	1.838×10^{-7}
IP	$1036.711\ ms$	$968.196\ ms$ (93.39%)	$68.515\ ms$ (6.61%)	0.422×10^{-7}

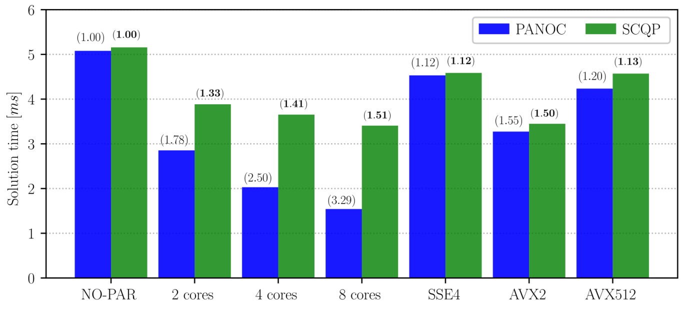

Fig. 4. Comparison of the solution times achieved with task and data parallelism over the evaluation of functions of OCP (8) solved with PANOC and SCQP. The speed-up relative to the corresponding NO-PAR execution is shown on top of each bar.

affected by communication overhead, which limits the benefit obtained form task parallelism for this test. Data parallelism does not achieve the same performance as task parallelism, with a maximum speed-up of 1.55 with PANOC using AVX2. This is justified by the frequency scaling of the processor during SIMD executions, which is more noticeable with AVX512 instructions. It should be noted that such frequency scaling affects not only the evaluation of the functions that are instructed to be parallelized, but also all other processes running on the processor, including the evaluation of (i) the rest of the functions in the OCP, (ii) their derivatives and (iii) the solver itself, which slows down the OCP solution. Nevertheless, these results suggest that a combination of task and data parallelism can be exploited to further extend the benefits of parallelization in PANOC and SCQP.

V. CONCLUSIONS

This paper showed the benefits of exploiting task and data parallelism in the evaluation of functions for the real-time implementation of nonlinear MPC of robot manipulators. We assisted automatic vectorization by modifying data structures in CasADi to ease AVX2 and AVX512 implementations on code-generated functions. A first-order solution method (PANOC) and a second-order method (SCQP) were compared against the commonly used SQP and IP methods, showing a speed-up of up to 3.29x for PANOC and 1.51x for SCQP, both achieved with task parallelism on 8 cores with respect to the nonparallelized implementation. Future work will investigate the simultaneous use of task and data parallelism on function evaluation, as well as the application of the results of this work in experiments with a real robot.

REFERENCES

[1] L. Stella, A. Themelis, P. Sopasakis, and P. Patrinos, “A simple and efficient algorithm for nonlinear model predictive control,” in *2017 IEEE 56th Annu. Conf. Decis. Control*. IEEE, dec 2017, pp. 1939–1944.

[2] R. Verschueren, N. van Duijkeren, R. Quirynen, and M. Diehl, “Exploiting convexity in direct Optimal Control: a sequential convex quadratic programming method,” in *2016 IEEE 55th Conf. Decis. Control*, dec 2016, pp. 1099–1104.

[3] A. Wächter and L. T. Biegler, “On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming,” *Math. Program.*, vol. 106, no. 1, pp. 25–57, 2006.

[4] R. B. Wilson, “A simplicial method for convex programming,” Ph.D. dissertation, Harvard University, 1963.

[5] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, “CasADi: a software framework for nonlinear optimization and optimal control,” *Math. Program. Comput.*, vol. 11, no. 1, pp. 1–36, mar 2019.

[6] N. Van Duijkeren, “Online Motion Control in Virtual Corridors - For Fast Robotic Systems,” Ph.D. dissertation, KU Leuven, 2019.

[7] F. Debrouwere, W. Van Loock, G. Pipeleers, and J. Swevers, “Optimal Tube Following for Robotic Manipulators,” *IFAC Proc. Vol.*, vol. 47, no. 3, pp. 305–310, 2014.

[8] R. M. Murray, S. S. Sastry, and L. Zexiang, *A Mathematical Introduction to Robotic Manipulation*, 1st ed. CRC Press, 1994.

[9] R. Featherstone, *Rigid Body Dynamics Algorithms*. Boston, MA: Springer US, 2008.

[10] T. Faulwasser and R. Findeisen, “Nonlinear Model Predictive Control for Constrained Output Path Following,” *IEEE Trans. Automat. Contr.*, vol. 61, no. 4, pp. 1026–1039, apr 2016.

[11] H. Bock and K. Plitt, “A Multiple Shooting Algorithm for Direct Solution of Optimal Control Problems,” *IFAC Proc. Vol.*, vol. 17, no. 2, pp. 1603–1608, jul 1984.

[12] P. Sopasakis, E. Fresk, and P. Patrinos, “Open: Code generation for embedded nonconvex optimization,” *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 6548–6554, 2020.

[13] A. Themelis, L. Stella, and P. Patrinos, “Forward-Backward Envelope for the Sum of Two Nonconvex Functions: Further Properties and Nonmonotone Linesearch Algorithms,” *SIAM J. Optim.*, vol. 28, no. 3, pp. 2274–2303, jan 2018.

[14] V. Porpodas and T. M. Jones, “Throttling Automatic Vectorization: When Less is More,” in *2015 Int. Conf. Parallel Archit. Compil.* IEEE, oct 2015, pp. 432–444.

[15] A. Astudillo, J. Carpentier, J. Gillis, G. Pipeleers, and J. Swevers, “Mixed Use of Analytical Derivatives and Algorithmic Differentiation for NMPC of Robot Manipulators,” in *Model. Estim. Control Conf.*, Austin, Texas, 2021.

[16] J. Carpentier, G. Saurel, G. Buondonno, J. Mirabel, F. Lamiraux, O. Stasse, and N. Mansard, “The Pinocchio C++ library : A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives,” in *2019 IEEE/SICE Int. Symp. Syst. Integr.* IEEE, jan 2019, pp. 614–619.

[17] M. Gottschlag and F. Bellosa, “Reducing AVX-Induced Frequency Variation With Core Specialization,” in *9th Work. Syst. Multi-core Heterog. Archit.*, Dresden, Germany, 2019.

[18] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proc. spring Jt. Comput. Conf. - AFIPS ’67*. New York: ACM Press, 1967, pp. 483–485.