

# Effective GPU Parallelization of Distributed and Localized Model Predictive Control

Carmen Amo Alonso and Shih-Hao Tseng

**Abstract**—To effectively control large-scale distributed systems online, model predictive control (MPC) has to swiftly solve the underlying high-dimensional optimization. There are multiple techniques applied to accelerate the solving process in the literature, mainly attributed to software-based algorithmic advancements and hardware-assisted computation enhancements. However, those methods focus on arithmetic accelerations and overlook the benefits of the underlying system’s structure. In particular, the existing decoupled software-hardware algorithm design that naively parallelizes the arithmetic operations by the hardware does not tackle the hardware overheads such as CPU-GPU and thread-to-thread communications in a principled manner. Also, the advantages of parallelizable subproblem decomposition in distributed MPC are not well recognized and exploited. As a result, we have not reached the full potential of hardware acceleration for MPC.

In this paper, we explore those opportunities by leveraging GPU to parallelize the distributed and localized MPC (DLMPC) algorithm. We exploit the locality constraints embedded in the DLMPC formulation to reduce the hardware-intrinsic communication overheads. Our parallel implementation achieves up to  $50\times$  faster runtime than its CPU counterparts under various parameters. Furthermore, we find that the locality-aware GPU parallelization could halve the optimization runtime comparing to the naive acceleration. Overall, our results demonstrate the performance gains brought by software-hardware co-design with the information exchange structure in mind.

## I. INTRODUCTION

The high computational demands of Model Predictive Control (MPC) have restricted its applicability to slow processes and low-dimensional systems [1]. Large-scale systems often require solving high-dimensional problems and algorithms scale poorly. Given the omnipresence of high-dimensional large-scale systems in real-world applications, efforts have been made to make MPC schemes more computationally efficient for these systems.

Multiple solutions have been proposed to accelerate MPC runtimes [2]. One popular approach relies on providing computational enhancements to the optimization solving algorithms, either by exploiting the sparsity in the matrices or by finding initial points for the optimization [3]–[5]. In this realm, recent works have also applied ideas from explicit MPC [6] to large-scale networks [7], [8] by moving most of the computational burden outside the optimization algorithms. The other direction is to take advantage of state-of-the-art hardware such as multi-core processors (CPUs), many-core processors (GPUs) or field programmable arrays (FPGA) to perform computations in parallel [9]–[15]. In

some instances these two approaches are combined, so efficient optimization algorithms are solved using multiple threads via hardware-specific implementations.

Although these methods provide promising avenues, most of their efforts are centered around achieving efficient computations by appropriately exploiting algorithmic features, and rely on hardware to simply parallelize mathematical operations. Hence, the hardware implementation of the algorithms is completely decoupled from the original system formulation, and therefore any hardware-intrinsic overhead can only be handled by using efficient programming practices. Yet, some branches of MPC directly encoding parallelization features in their formulation have received very little attention in this field.

For instance, the merits of distributed MPC<sup>1</sup> have been overlooked in parallel settings [2], despite the fact that distributed MPC formulations are very well-suited for parallelization. Moreover, some distributed MPC frameworks allow the information exchange constraints among different subsystems [17]. This feature is not only present in most widespread large-scale systems [18]–[21], but also these information exchange constraints resemble the hardware-intrinsic communication limitations and overheads encountered in MPC parallelization. Despite the great promise of these MPC frameworks to deal with parallelization and hardware-intrinsic overheads in a principled manner through the problem formulation, its full potential has not been realized in the literature.

In this paper we close this gap. We provide a principled parallel implementation and overhead analysis through an appropriate distributed MPC framework that allows for local communication constraints. We exploit the potential for parallelization of this scheme in a GPU, where the GPU is not used to parallelize arithmetic computations but rather each computing thread is tasked with computing the operations corresponding to a subsystem in the network. Moreover, we demonstrate how simply applying standard parallelization techniques to the algorithm incurs unnecessary overhead. And we show that communication exchange constraints embedded in the framework allow us to explicitly deal with these hardware-intrinsic communication overheads in a principled manner by means of longest-vector length, combined kernels and local memory.

In particular, we take advantage of the recent work in [17] and its extension [8], which provides a Distributed and

Carmen Amo Alonso and Shih-Hao Tseng are with the Division of Engineering and Applied Science, California Institute of Technology, Pasadena, CA 91125, USA. Emails: {camoalon, shtseng}@caltech.edu

<sup>1</sup>Distributed MPC ports the ideas of standard MPC to the distributed setting, where different subsystems have different subcontrollers that operate in parallel and can communicate with each other in a local fashion [16].

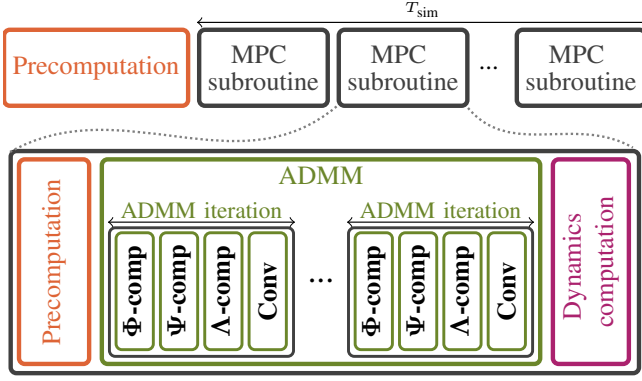


Fig. 1. The DLMPC algorithm consists of different computation steps. A precomputation step is carried out to compute necessary matrices that stay constant throughout all MPC iterations. Once this is completed, MPC iterations run sequentially (one per time-step) for a given number  $T_{\text{sim}}$  of time steps. Within a MPC iteration, a precomputation step precedes the ADMM algorithm. Once converged, we compute the next control input and the state according to the dynamics. Each iteration of ADMM is composed by the steps detailed in Algorithm 2.

Localized MPC (DLMPC) controller capable of encoding the communication structure of the network. It also provides a distributed algorithm to compute the DLMPC controller, where algorithmic iterations result in basic arithmetic operations that are scalable independently of the size of the network. We note that the limitations in communication among the GPU computing threads resemble the communication scheme in control systems for large-networks, and we take advantage of the local communication constraints that are already included in the DLMPC algorithm. We demonstrate through simulations the effectiveness of our method.

For the remainder of the paper, we present brief overviews of the DLMPC algorithm [17], its explicit form [8], and GPU parallelization in Section II. In Section III we analyze the GPU implementation and reduce the overhead via several concepts, such as longest-vector length, combined kernels and local memory. We demonstrate the usefulness of these improvements through simulations in Section IV. Finally, we conclude the paper in Section V along with some future research opportunities and directions.

## II. PRELIMINARIES

In this section we introduce the preliminary concepts necessary for the GPU implementation of DLMPC. We start with a brief introduction of the DLMPC algorithm [8], [17]. We then follow with an overview of GPU parallelization.

### A. Distributed and Localized Model Predictive Control

Consider a large-scale discrete-time linear time invariant (LTI) system with dynamic matrices  $(A, B)$ . The system is composed of  $N$  subsystems interconnected according to an unweighted graph  $\mathcal{G}_{(A,B)}$ . The control input for the system is computed through a MPC controller, where at each time step an optimal control problem is solved over the finite time horizon  $T$  using the current state as initial condition for the

ADMM variables:

$$\Phi = \begin{bmatrix} \triangle & \triangle & 0 \\ 0 & \triangle & \triangle \end{bmatrix}, \Psi = \begin{bmatrix} \bullet & \bullet & 0 \\ 0 & \bullet & \bullet \end{bmatrix}, \Lambda = \begin{bmatrix} \blacksquare & \blacksquare & 0 \\ 0 & \blacksquare & \blacksquare \end{bmatrix}$$

ADMM iteration:

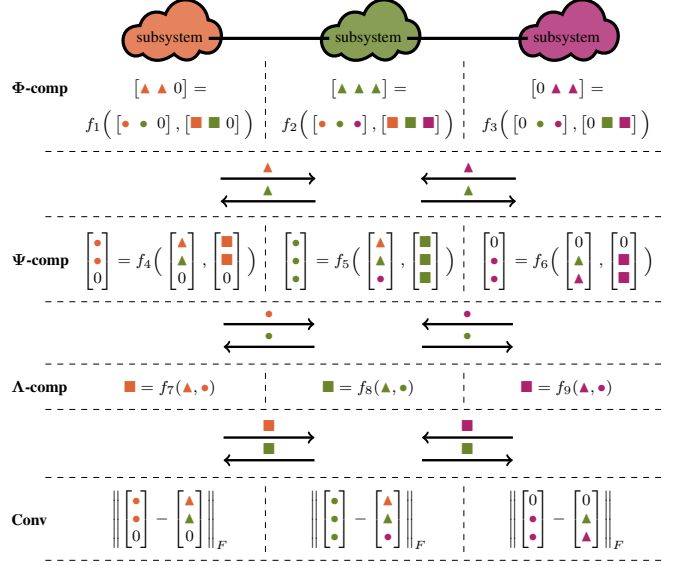


Fig. 2. An example showing how Algorithm 2 distributes decision variables across a 3-node network. In each iteration,  **$\Phi$ -comp** distributes  $\Phi$  row-wisely, and then  **$\Psi$ -comp** distributes  $\Psi$  column-wisely. After that,  **$\Lambda$ -comp** updates  $\Lambda$  based on the new  $\Phi$  and  $\Psi$  and we determine if ADMM converges according to some criterions in the **Conv** step. Between each computation step, each subsystem shares the updated variable with its local neighborhood (in particular, the set of its  $d$ -hop neighbors).

dynamics. Hence, at time step  $\tau$ :

$$\begin{aligned} \min \quad & \sum_{t=0}^{T-1} x_t^T Q_t x_t + u_t^T R_t u_t + x_T^T Q_T x_T \\ \text{s.t.} \quad & x_0 = x(\tau), \quad x_{t+1} = Ax_t + Bu_t, \\ & x_t^{\min} \leq x_t \leq x_t^{\max}, \quad x_T^{\min} \leq x_T \leq x_T^{\max}, \\ & u_t^{\min} \leq u_t \leq u_t^{\max} \quad \forall t \in \{0, \dots, T-1\}, \\ & d\text{-locality constraints according to } \mathcal{G}_{(A,B)}, \end{aligned} \quad (1)$$

where  $x_t \in \mathbb{R}^{N_x}$  is the state and  $u_t \in \mathbb{R}^{N_u}$  the control input at time  $t$  in the MPC problem.  $x(\tau)$  represents the measured state at time step  $\tau$ , used as an initial condition for the MPC problem. We require the states and inputs to be bounded above and below (marked by the superscript  $\max$  and  $\min$  respectively).

The  $d$ -locality constraints restrict the information exchange among subsystems to occur at a local scale, i.e. subsystem  $i$  can only share information with subsystems within distance  $d \ll N$ , as measured per the interconnection graph  $\mathcal{G}_{(A,B)}$ . Hence, the closed loop control policy for subcontroller  $i$  can be computed using only states, control actions, and system models collected from  $d$ -hop neighbors. For a formal definition of this constraint, interested readers are referred to [17].

In order to properly deal with the local communication constraints, we resort to the System Level Synthesis (SLS) parametrization [22], [23]. In SLS, the decision variables

of the MPC problem (1) are replaced by a matrix  $\Phi \in \mathbb{R}^{(N_x T + N_u(T-1)) \times N_x}$ . Each row of  $\Phi$  corresponds to a state or an input in the system at a certain time step, so each state and input in the system occupy  $T$  and  $T-1$  rows of  $\Phi$ , respectively. Locality constraints are translated into some suitable sparsity requirements on  $\Phi$ . In particular, the element of  $\Phi$  at the  $i^{\text{th}}$  row and the  $j^{\text{th}}$  column is non-zero only if the distance between the subsystems corresponding to the  $i^{\text{th}}$  row and the  $j^{\text{th}}$  column is smaller than  $d$  on  $\mathcal{G}_{(A,B)}$ . Hence, the number of non-zero elements that each subsystem solves for is  $O(d)$ . For a formal definition of locality constraints, see [22], [24].

Once the MPC subroutine (1) is expressed in SLS, it is possible to distribute the optimization by means of the Alternating Direction Method of the Multipliers (ADMM) [25]. The ADMM algorithm consists of three steps. At each step, the required computations are distributed across the subsystems. Given the locality constraints, the dimension of the distributed subproblems will be dominated by  $d$  as opposed to  $N$ . [8] shows that all three steps can be efficiently computed: Two steps admit a closed-form solution, and the other one has an explicit solution. Precomputations are necessary to enjoy the benefits of closed-forms and explicit solutions. A sketch of the DLMPC algorithm is presented in Algorithm 1 and Algorithm 2, and we visualize the process in Fig. 1 and Fig. 2.

---

**Algorithm 1** DLMPC algorithm sketch

---

- 1: Precompute necessities of closed-form solutions.
  - 2: Initialize  $x(0)$ .
  - 3: **for**  $t = 0$  **to**  $T_{\text{sim}}$  **do**
  - 4:   Precompute the explicit solution using  $x(t)$ .
  - 5:   Perform ADMM (see Algorithm 2 for details).
  - 6:   Compute  $u(t)$  from  $\Phi$  and obtain  $x(t+1)$ .
  - 7: **end for**
- 

---

**Algorithm 2** ADMM computations sketch for subsystem  $i$

---

- 1:  $\text{conv} \leftarrow \text{false}$ .
  - 2: **while**  $\text{conv}$  is **false** **do**
  - 3:    **$\Phi$ -comp**: Compute rows of  $\Phi$  via the explicit solution in [8].
  - 4:   Share  $\Phi$  with its  $d$ -hop local neighbors.
  - 5:    **$\Psi$ -comp**: Compute columns of  $\Psi$ .
  - 6:   Share  $\Psi$  with its  $d$ -hop local neighbors.
  - 7:    **$\Lambda$ -comp**:  $\Lambda \leftarrow \Lambda + \Phi - \Psi$ .
  - 8:   Share  $\Lambda$  with its  $d$ -hop local neighbors.
  - 9:   **Conv**: Check the convergence criterion and save the result in  $\text{conv}$ .
  - 10: **end while**
- 

### B. GPU Parallelization Overview

GPU differs from CPU in computation and memory, which have a profound impact on programming and implementa-

tions. We elaborate on those differences below.

**Computing threads:** A thread is the smallest independent sequence of instructions in a computing process. CPU is able to handle complex tasks using a limited number of threads in the order of  $O(10)$ . In contrast, GPU has the capacity of running thousands to millions of threads in parallel, but each is capable of simpler operations. A GPU computing process is referred as *kernel*.

**Sharing memory:** Comparing to single-thread tasks in CPU, memory access is much more involved under a multi-thread scenario like GPU. When more than one thread access a sharing memory location, a race condition can occur when they both attempt to modify the content, and their access order determines the outcome. As a result, the consistency and correctness of the results are not guaranteed without special treatments. To avoid a race condition, we should either explicitly enforce some order among the threads or avoid concurrent access to the same memory locations. The former option is not preferred as it undermines the benefits of parallelism. On the other hand, memory sharing restriction curbs inter-thread information exchange. As a result, an algorithm needs to avoid information exchange among its parallel components to achieve high performance on GPU.

Given the characteristics, although GPU has great potential to boost algorithms' performance through parallelization, a GPU-parallelized algorithm is subject to two kinds of communication overheads: (i) *CPU-GPU* and (ii) *thread-to-thread*. CPU-GPU communication incurs an overhead on copying large volume of data between the memory systems of CPU and GPU, and thread-to-thread communication imposes an overhead on handling coupled memory accesses among parallel threads. GPU local memory<sup>2</sup> offers a limited resource to mitigate these overheads. It provides a narrow memory space accessible to a *local* group of threads with synchronization barriers, and hence allows local groups of threads to exchange information through these shared memory locations with consistency guarantee. Meanwhile, local memory is limited to a local scale, and a single thread cannot belong to two different groups.

### III. GPU PARALLELIZATION

In this section, we provide the implementation of the DLMPC algorithm in a computer system equipped with both a CPU and a GPU. First, we take advantage of the parallelization potential of the algorithm and perform a *naive parallelization* of the ADMM steps in GPU. We then provide enhancements to reduce overheads, such as reducing setup complexity by using *longest-vector length*, and reducing CPU-GPU communication by setting *combined kernels*. Lastly, we present how the locality constraints allow to effectively use *local memory* to deal with the thread-to-thread coupling.

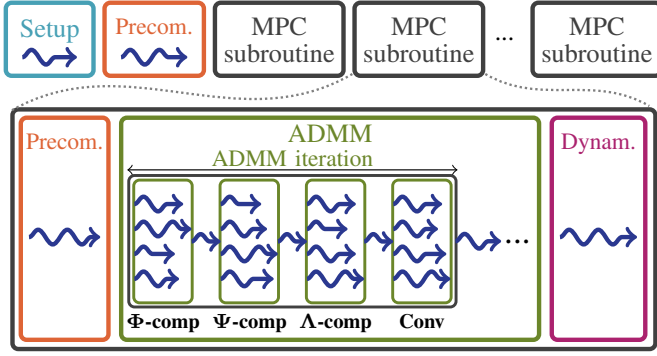


Fig. 3. Naive parallelization implementation. Boxed components represent the same computation steps as in Fig. 1. An additional Setup step has been introduced at the beginning of the implementation for GPU setup. Computing threads are denoted with blue wavy arrows: A single arrow represents a single-thread computation, and multiple arrows within a computation step represent a multi-thread computation. The length of the different arrows in multi-thread processes represents runtime for each of the threads.

#### A. Naive Parallelization

We start with a naive parallelization of DLMPC algorithm by parallelizing ADMM steps in Algorithm 2. For each ADMM step, we assign each of the subproblems below a single thread in GPU:

- For  $\Phi$ -comp, each thread computes one row of  $\Phi$ .
- For  $\Psi$ -comp, each thread computes one column of  $\Psi$ .
- For  $\Lambda$ -comp, each thread computes one element of  $\Lambda$ .
- For Conv, each thread evaluates the convergence criterion against one column.

Notice that each thread in GPU is tasked with performing all necessary arithmetic operations leading to the assigned row/column/element. In this implementation we are not parallelizing the arithmetic computations in GPU, but rather treating each GPU thread as a subsystem of the distributed MPC framework.

According to the ADMM algorithm, abundant information sharing is required after each computation in order to perform the next computation. Due to the limitations of GPU communication among threads, in this naive parallelization we perform the information sharing in the form of memory accesses in CPU. Therefore, after each parallelized computation we return to CPU to exchange results and set up the next one. We illustrate this implementation in Fig. 3, where we represent the computing threads with an arrow so one can distinguish the steps that are computed in CPU (single thread) and the ones that are computed in GPU (multi-thread). Notice that an additional setup step required to launch the GPU kernels is also represented.

This naive parallelization of the DLMPC algorithm suffers from several overheads. First, the threads have different runtime due to the various lengths of the rows and columns they compute. Such various lengths result in significant setup overhead before computation. Second, there are several CPU-GPU switches per ADMM iteration to exchange information across different threads for rows and columns. This incurs CPU-GPU communication overhead. Those overheads imply

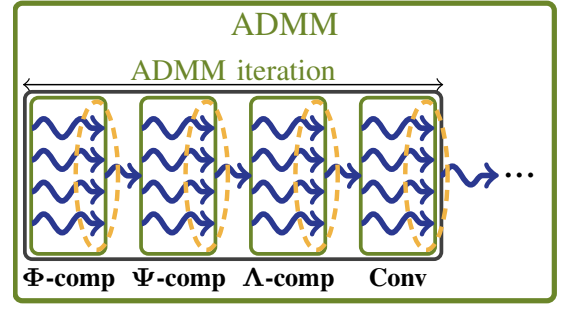


Fig. 4. Since the computation time of parallelized threads is dominated by the slowest one, we could omit computing the exact input vector size for each thread and feed all threads with same-sized (the longest-sized) input vectors, which results in roughly the same runtime for all the threads. This simplifies both the setup and per-thread computation and hence reduces the overhead.

that a naive parallelization of a distributed and localized MPC scheme such as DLMPC is not necessarily efficient, and additional considerations are needed to fully exploit the GPU potential. In what follows, we analyze these overheads and provide effective solutions based on hardware-specific considerations and the presence of locality constraints. We build upon these solutions until an optimal GPU implementation of the ADMM steps is presented.

#### B. Longest-Vector Length

Threads have different runtime in Fig. 3 since they process different sizes of input vectors. Feeding each thread a different-sized input vector imposes a setup overhead – we need to compute, store, and pass as parameters of the threads the sizes of each input vector. Such an overhead was justifiable in a single thread CPU version like [26] as we want to avoid unnecessary computations. In particular, only non-zero elements are needed when computing on a single thread, and filtering out non-zero elements pays off as fewer inputs imply faster computation under sequential processing. The situation changes in GPU parallelization. Since the computation time of parallelized threads is determined by the slowest one, and the kernel does not return until *all* threads have finished the computations, it is no longer beneficial to trim off zero elements unless they are processed by the slowest thread.

Accordingly, we can save the efforts of attaining exact different-sized input vectors for parallelization. Instead, we only need to ensure the input vector is long enough to cover the non-zero elements and find the minimum upper bound on the length, which is the maximum number of non-zero elements in the rows and columns of  $\Phi$  and  $\Psi$ , respectively, or the *longest-vector length* for short. We denote by  $D_{\text{row}}$  and  $D_{\text{col}}$  the longest-vector lengths of  $\Phi$  and  $\Psi$ , respectively, and establish below that by virtue of the locality constraints,  $D_{\text{row}}, D_{\text{col}} \ll N$  and the number of elements that a thread solves for is much smaller than the size of the network.

**Lemma 1.** *Let  $s$  be the maximum number of states or control inputs per subsystem in the network, and  $l$  the maximum degree of nodes in  $\mathcal{G}_{(A,B)}$ . Suppose  $\mathcal{G}_{(A,B)}$  is subject to  $d$ -locality constraints and the MPC time horizon is  $T$ , then*

<sup>2</sup>We use OpenCL terminology, also referred as *shared memory* in CUDA.

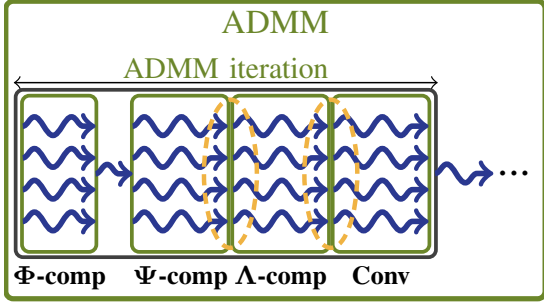


Fig. 5. To reduce CPU-GPU communication overhead, we remove the single-thread computation in between computations (circled in yellow) by combining the computations into per-column threads. This enhances the parallelization by combining kernels. Such a combination does not apply to the information exchange between  $\Phi$ -comp and  $\Psi$ -comp as we shift from per-row to per-column computation.

$D_{row}$  and  $D_{col}$  are bounded by

$$D_{row} \leq \frac{s(l^{d+1} - 1)}{l - 1}, \quad D_{col} \leq \frac{(2T - 1)s(l^{d+1} - 1)}{l - 1}.$$

*Proof.* Each row in  $\Phi$  represents a state/input in a subsystem, and hence  $D_{row}$  is the number of states that it can receive information from. We can establish a bound on  $D_{row}$  by bounding the number of nodes within  $d$ -hops. By definition, we know that there are at most  $l^k$  nodes that are  $k$ -hops away from a node, so within  $d$ -hops, there are at most

$$1 + l + l^2 + \dots + l^d = \frac{l^{d+1} - 1}{l - 1}$$

nodes, each has at most  $s$  states, which shows the bound.

On the other hand,  $D_{col}$  is the number of states and inputs, among all horizon  $T$ , a state can impact. Similarly, by  $d$ -locality constraints, we can use the bound on  $D_{row}$  as an estimate of the states/inputs a state would impact at each time. Since there are, in total,  $T$  states and  $T - 1$  inputs in  $\Psi$  per column, we can bound  $D_{col}$  by  $(2T - 1)$  times of the above bound on  $D_{row}$ , which yields the desired result.  $\square$

Given a sparse system, the maximum degree  $l$  is expected to be small, as is the maximum number of subsystem states/inputs  $s$ . Given the assumption  $d \ll N$ , the above lemma suggests  $D_{row}, D_{col} \ll N$

### C. Combined Kernels

We then tackle the CPU-GPU communication overhead. In the naive parallelization, CPU-GPU communication are necessary to properly exchange information in between computations. The reason is that each computation occurs according to a different distribution of the elements of  $\Phi$ ,  $\Psi$ ,  $\Lambda$ , i.e. row-wise, column-wise, and element-wise. Although this particular distribution of elements might be the most efficient for each of the computations isolated, the additional GPU-CPU overhead steaming from the information sharing in between computations makes this option is suboptimal.

To reduce the CPU-GPU communication overhead, we proposed the use of combined kernels. In particular, the last three computation steps in the ADMM iteration can be combined in the same kernel by parallelizing  $\Lambda$ -comp in a

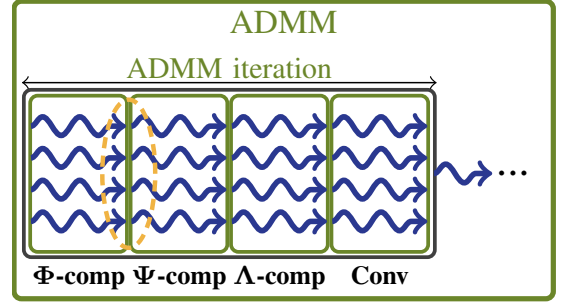


Fig. 6. Conducting  $\Phi$ -comp and  $\Psi$ -comp consecutively in GPU requires the per-row threads to exchange information with per-column threads, which results in thread-to-thread communication. Using local memory, we can avoid such a thread-to-thread communication by creating column patches, which duplicate row-wise computation threads to decouple the per-column local memory groups and synchronize results entirely within GPU.

column-wise fashion (as opposed to element-wise). By distributing the computations in this manner, each of the threads has sufficient information to sequentially perform  $\Psi$ -comp,  $\Lambda$ -comp, and  $\text{Conv}$  without the need for communication among threads. This reduces the CPU-GPU communication overhead since only one exchange between CPU and GPU is necessary for each ADMM iteration (for the transformation from row-wise to column-wise). However, the treatment herein slightly degrades the parallelization benefits of  $\Lambda$ -comp, since by having only a thread per column, each thread now has to loop over the elements in its column sequentially. This additional overhead is very modest because due to the locality constraints, the number of relevant elements per thread is  $D_{col}$ , as opposed to a CPU-GPU memory-copying operation, where the variables handled are of order  $N$ .

### D. Local Memory and Column Patch

The information exchange between  $\Phi$ -comp and  $\Psi$ -comp involves the transformation from row-wise to column-wise representation and hence is not easily combined into one kernel. The key difficulty is that the row-wise results should be passed down to per-column threads, which results in thread-to-thread communication/synchronization issues. We could realize synchronized thread-to-thread communications through local memory. However, local memory is shallow – It would not fit all the threads in – and it is exclusive – One thread can only belong to one group. These properties make the bipartite information exchange pattern difficult to enforce: The row-wise result might be required by multiple per-column threads, while each thread may need multiple row-wise results. As a result, to leverage local memory to save CPU-GPU communication overhead, we need to group the threads smartly.

Our approach is to group each per-column thread in  $\Psi$ -comp with row-wise computation threads in  $\Phi$ -comp, referred to as *column patch*. That is, for the  $i^{\text{th}}$  column of  $\Psi$ , we launch a column patch to solve for the  $j^{\text{th}}$  row where  $j \in \{j : \Phi(j, i) \neq 0\}$ . Once this is done, the row-wise results are saved in local memory in GPU, and one of those threads can proceed with the column-wise computations of  $\Psi$  and  $\Lambda$  as described in the previous subsection without returning to CPU.



Thanks to the locality constraints, each column patch only has  $D_{\text{col}}$   **$\Phi$ -comp** threads to include and fit their results in the shallow local memory. On the other hand, since each row has several non-zero elements, we could potentially have multiple threads in different column patches that compute the same row-wise result. But it is fine as GPU has plenty of threads to launch, and using multiple threads to compute the same result in parallel does not incur additional runtime overhead. Therefore, we ensure synchronization without the need for information sharing across threads - since we can repeat relevant computations in different local groups - or computing units - since local synchronization is all is needed. This was only possible by exploiting the GPU architecture together with the locality constraints directly encoded in the DLMPC formulation.

We highlight the roles of the locality constraints in our enhancement techniques. Locality constraints can facilitate desirable trade-offs between computational resources and information exchange across threads: Longest-vector length incurs additional precomputation steps, and combined kernels sacrifice parallelization potential of  **$\Lambda$ -comp**. Those trade-offs are justified by the small dimensions derived from the locality constraints  $D_{\text{row}}, D_{\text{col}} \ll N$ . Meanwhile, for the local memory and column patch technique, locality constraints allow us to decouple the row-wise threads without harming the runtime (in essence, locality allow us to pay in the spatial space to decouple the row-wise threads without temporal performance degradation).

#### IV. EVALUATION

Through simulations, we study two aspects of the GPU-parallelized DLMPC. First, we compare the scalability of our implementation with other methods. We then analyze the overhead of the implementations for future enhancements.

##### A. Setup

We implement our GPU-parallelized DLMPC in Python and OpenCL.

We compare the scalability of the four proposed GPU implementations in Section III against two CPU variations - a Python replica of the single-threaded DLMPC version in [26] and an optimization-based approach under the SLSpy framework [27] with CVXPY [28] as the solver. The results are measured on a desktop with AMD Ryzen 7 3700X processor (16 logical cores), 32 GB DDR4 memory, and AMD Radeon RX 550/550X GPU. For each evaluated scenario, we simulate 100 different initial conditions and present the average and the standard deviation of the measurements. The synthetic dynamics is chosen the same as in [8], which is a chain-like network with two-state nodes as the subsystems. Each subsystem  $i$  evolves according to

$$[x(t+1)]_i = [A]_{ii}[x(t)]_i + \sum_{j \in \mathbf{in}_i(d)} [A]_{ij}[x(t)]_j + [B]_{ii}[u(t)]_i,$$

where  $\mathbf{in}_i(d)$  contain the  $d$ -hop neighbors of node  $i$  and

$$[A]_{ii} = \begin{bmatrix} 1 & 0.1 \\ -0.3 & 0.7 \end{bmatrix}, [A]_{ij} = \begin{bmatrix} 0 & 0 \\ 0.1 & 0.1 \end{bmatrix}, [B]_{ii} = I.$$

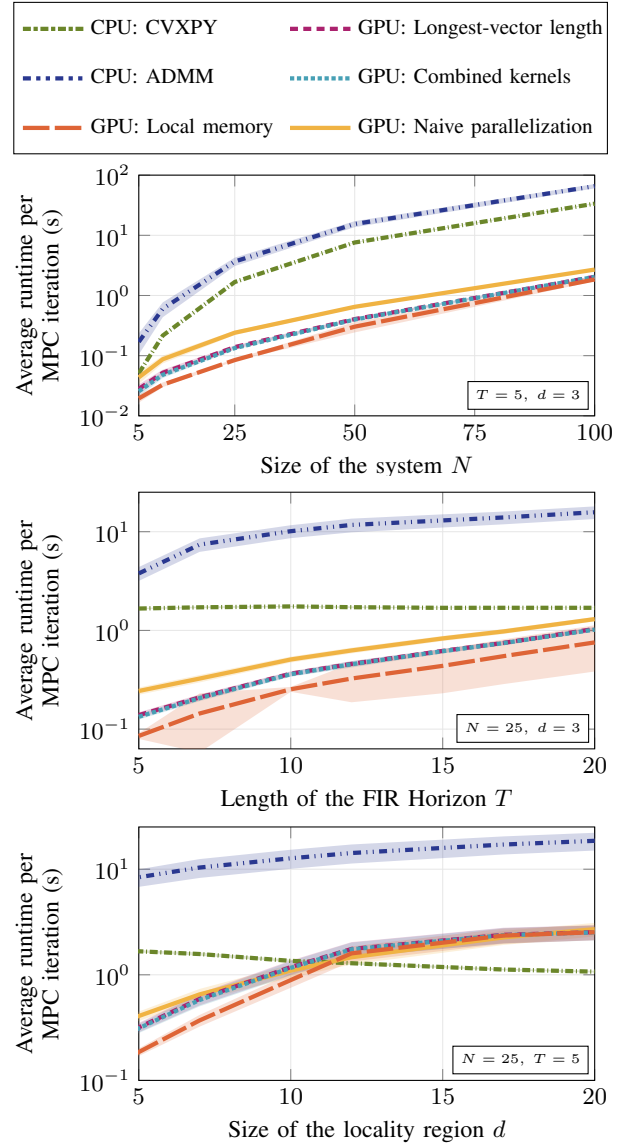


Fig. 7. Comparison of the runtimes obtained by different computing strategies for different parameter regimes. The lines are the mean values and the shaded areas show the values within one standard deviation. We observe that GPU computation strategies scale much better with the size of the network than CPU implementations. This trend also holds for ADMM CPU implementation over all time horizons and locality region sizes, while GPU implementations only outperform CVXPY on small locality regions and CVXPY seems to scale well over all simulated time horizons.

The state is subject to upper and lower bounds:

$$-0.2 \leq [x(t)]_{i,1} \leq 1.2 \quad \text{for } t = 1, \dots, T,$$

where  $[x]_{i,1}$  is the first state in the two-state subsystem  $i$ .

We perform the MPC with  $T_{\text{sim}} = 20$  subroutine iterations with the cost function

$$f(x, u) = \sum_{i=1}^N \sum_{t=1}^{T-1} \|[x(t)]_i\|_2^2 + \|[u(t)]_i\|_2^2 + \|[x(T)]_i\|_2^2.$$

Note that the number of states and inputs in this plant is  $3N$ , since each of the  $N$  subsystems has 2 states and 1 input.

### B. Scalability

To evaluate the scalability of the methods, we run the simulations with varying system size  $N$ , MPC time horizon  $T$ , and locality region size  $d$ . We measure the average runtime per MPC iteration, i.e., the total runtime divided by the number of MPC subroutine iterations  $T_{\text{sim}}$ , and summarize the results in Fig. 7. We remark that the runtime is measured for the *whole* simulation rather than merely the ADMM portion of the algorithm.

In Fig. 7, the GPU implementations scale much better with the network size  $N$  than the CPU implementations. Moreover, the runtime differences grow from an order of magnitude to several orders of magnitude as  $N$  increases. This is as expected since GPU implementations can parallelize the subsystem computations through multi-threads whereas the CPU implementations cannot. Remarkably, the ADMM implementation in CPU is consistently worse than when solved via CVXPY, which emphasizes the need for a parallel implementation such as the one presented in this paper to fully take advantage of the DLMPC algorithm. Among the GPU methods, local memory has superior performance ( $50\times$  faster than ADMM in CPU for  $N = 25$  and  $35\times$  for  $N = 100$ ), followed by combined kernels and longest-vector length ( $15\times$  faster than ADMM in CPU for  $N = 25$  and  $25\times$  for  $N = 100$ ). Naive parallelization is  $3\times$  slower than local memory for small  $N$ . In fact, a smaller  $N$  leads to a bigger performance difference among GPU implementations, which indicates that the CPU overhead of the implementations outweighs the improvements made by GPU when the network scales.

For time horizon  $T$  and locality region size  $d$ , the runtime scales accordingly for all the ADMM implementations. This can be seen from a simple analysis of the optimization variables: Since larger  $T$  and  $d$  introduce more non-zero elements in the matrix  $\Phi$ , the corresponding decomposed row and column vectors become longer and the runtime increases. However, this trend does not apply to the SLSPY-based CPU implementation solved by CVXPY, where the runtime stays pretty much the same or even decreases when  $T$  and  $d$  increases. Nevertheless, the GPU implementations still outperform CVXPY by up to one order of magnitude for  $d \leq 10$  and all simulated  $T$ . In other words, the advantage of GPU parallelization is significant when the locality region is small. Hence, in light of the results, the ADMM implementation of DLMPC is best for large network consisting of subsystems with relatively small local neighborhood.

### C. Overhead Analysis

To better understand how different methods scale in runtime with the network size, we break down the runtime into four DLMPC phases as in Fig. 3, which are

**Setup** refers to all the computational steps necessary to setup the GPU computations. It refers to the compilation of the kernels themselves, but also to all the extra computations necessary to implement the different GPU schemes, such as

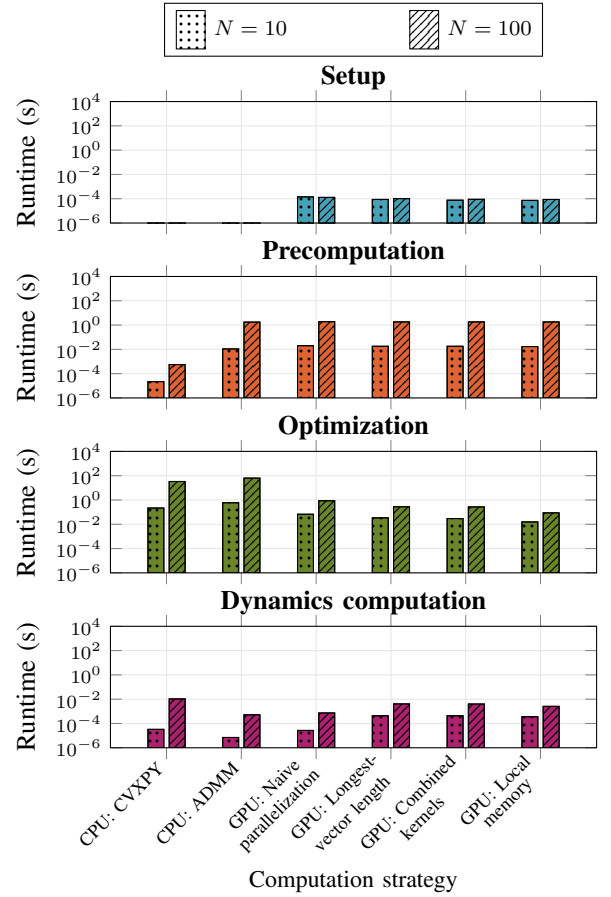


Fig. 8. Runtime breakdown for the different computing phases: precompilation, precomputation, optimization and dynamics computation for two different network sizes  $N = 10$  and  $N = 100$ . Colors for the different phases are the same as in Fig. 3. For CPU implementations, the optimization phase is the bottleneck, while the bottleneck shifts to the precomputation for the GPU implementations.

the computation of  $D_{\text{row}}$  and  $D_{\text{col}}$ , etc.

**Precomputation** refers to computing the necessary matrices and vectors that multiply the decision variables  $\Phi$ ,  $\Psi$ ,  $\Lambda$ . We note here that this phase appears twice in the implementation: before starting any of the MPC computations, and before starting each of the MPC subroutines.

**Optimization** refers to the computational steps necessary to solve the optimization problem (1). In this case, it refers to the steps taken by ADMM or CVXPY to find a value for  $\Phi$ .

**Dynamics computation** refers to the computation of the next state using the control input given by MPC.

We measure the runtime breakdown for network size  $N = 10$  and  $N = 100$  and present the results in Fig. 8. From Fig. 8, we can observe that the setup and dynamics computation phases are relatively fast, and the runtime is dominated by the optimization phase when  $N = 10$  and the precomputation phase when  $N = 100$ . As we adopt more sophisticated techniques described in Section III to GPU implementations, the optimization phase speeds up significantly. However, the precomputation phase does not benefit from the techniques and it becomes the bottleneck when the network size scales. Since an increase in  $d$  or  $T$

mostly burdens the setup and precomputation phases, the results also explain why the GPU implementations scale poorer when  $d$  or  $T$  is large.

Another important takeaway from Fig. 8 is that for CPU implementations, the optimization phase is the bottleneck, while the bottleneck shifts to the precomputation for the GPU implementations. This justifies our approaches in this paper to accelerate the CPU optimization phase by GPU parallelization. Meanwhile, the results also suggest that to further speed up the computation, future research should focus on faster precomputation techniques.

## V. CONCLUSION AND FUTURE DIRECTIONS

We develop effective GPU parallelized DLMPC for large-scale distributed system control. Our results show that although a naive GPU implementation does improve the performance by  $15 - 25\times$ , we can still get up to  $50\times$  performance improvement by taking into account the hardware-intrinsic limitations. We overcome these limitations by taking advantage of the local communication constraints in the formulation, and developing longest-vector length, combined kernels and local memory implementations. With extensive experiments, we demonstrate that the DLMPC algorithm is suitable for GPU parallelization, and that its full potential is only realized when the local communication constraints are taken into account in the GPU implementation. We demonstrate the scalability of the method for large network sizes, and noticed that most of the computational overhead in the GPU computations was due to the precomputations being performed in CPU.

As discussed in the overhead analysis, precomputation becomes the new overhead after our GPU parallelizations. Therefore, a future direction would be to effectively parallelize the precomputation for higher performance. In addition, there are some other parts of DLMPC that we can parallelize further, such as a better initial point for ADMM to converge faster as well as a better parallelized dynamics computation. Another future direction is to include more than one ADMM iteration into one kernel, to avoid CPU-GPU data exchanges. One might also be interested in extending the parallelization in this work to a fully distributed setting where the processing units scatter over a network. As the distributed setting introduces new challenges such as robustness to communication dropouts, synchronization, and delay, it would be interesting to see how locality constraints could improve robustness or performance.

## REFERENCES

- [1] S. Qin and T. A. Badgwell, "A survey of industrial model predictive control technology," *Control Eng. Pract.*, vol. 11, no. 7, pp. 733–764, 2003.
- [2] K. M. Abughalieh and S. G. Alawneh, "A survey of parallel implementations for model predictive control," *IEEE Access*, vol. 7, pp. 34 348–34 360, 2019.
- [3] H. Jonson and T. Glad, "A method for state and control constrained linear quadratic control problems," in *Proc. 9th IFAC World Cong., Budapest, Hungary*, 2–6 July, 1984, pp. 229–233.
- [4] S. Richter, C. N. Jones, and M. Morari, "Real-time input-constrained MPC using fast gradient methods," in *Proc. IEEE CDC*, 2009, pp. 7387–7393.
- [5] F. Ke, Z. Li, H. Xiao, and X. Zhang, "Visual servoing of constrained mobile robots based on model predictive control," *IEEE Trans. on Syst., Man, and Cybern.: Syst.*, vol. 47, no. 7, pp. 1428–1438, 2017.
- [6] A. Bemporad, M. Morari, V. Dua, and E. N. Pistikopoulos, "The explicit linear quadratic regulator for constrained systems," *Automatica*, vol. 38, no. 1, pp. 3–20, 2002.
- [7] J. Oravec, Y. Jiang, B. Houska, and M. Kvasnica, "Parallel explicit MPC for hardware with limited memory," in *20th IFAC World Cong., Toulouse, France*, 9–14 July, 2017, pp. 3301–3306.
- [8] C. Amo Alonso, N. Matni, and J. Anderson, "Explicit distributed and localized model predictive control via system level synthesis," in *Proc. IEEE CDC*, 2020, pp. 5606–5613.
- [9] M. Kogel and R. Findeisen, "Parallel solution of model predictive control using the alternating direction multiplier method," in *Proc. 4th IFAC Conf. on Nonlinear Model Predictive Control*, 2012, pp. 369–374.
- [10] D.-K. Phung, B. Hérissé, J. Marzat, and S. Bertrand, "Model predictive control for autonomous navigation using embedded graphics processing unit," in *Proc. 20th IFAC World Cong., Toulouse, France*, 9–14 July, 2017, pp. 11 883–11 888.
- [11] L. Yu, A. Goldsmith, and S. Di Cairano, "Efficient convex optimization on gpus for embedded model predictive control," in *Proc. General Purpose GPUs, Austin, TX, USA*, 2017, pp. 12–21.
- [12] P. Hyatt, C. S. Williams, and M. D. Killpack, "Parameterized and GPU-parallelized real-time model predictive control for high degree of freedom robots," 2020. [Online]. Available: <https://arxiv.org/abs/2001.04931>
- [13] B. Plancher and S. Kuindersma, "Realtime model predictive control using parallel DDP on a GPU," in *Proc. IEEE ICRA*, Montreal, Canada, May, 2019.
- [14] K. Ling, B. Wu, and J. Maciejowski, "Embedded model predictive control (MPC) using a FPGA," in *17th IFAC World Cong., Seoul, Korea*, 6–11 July, 2008, pp. 15 250–15 255.
- [15] J. L. Jerez *et al.*, "Embedded online optimization for model predictive control at megahertz rates," *IEEE Trans. Autom. Control*, vol. 59, no. 12, pp. 3238–3251, 2014.
- [16] R. Scattolini, "Architectures for distributed and hierarchical model predictive control – a review," *Journal of Process Control*, vol. 19, no. 5, pp. 723–731, May 2009.
- [17] C. Amo Alonso and N. Matni, "Distributed and localized closed loop model predictive control via system level synthesis," in *Proc. IEEE CDC*, 2020, pp. 5598–5605.
- [18] J. C. Doyle *et al.*, "The 'robust yet fragile' nature of the Internet," *Proc. Nat. Acad. Sci.*, vol. 102, no. 41, pp. 14 497–14 502, 2005.
- [19] R. R. Negenborn, "Multi-agent model predictive control with applications to power networks," 2007. [Online]. Available: <http://www.negenborn.net/mampc/>
- [20] C. Lynn and D. Bassett, "The physics of brain network structure, function and control," *Nature Rev. Phys.*, vol. 1, no. 5, pp. 318–332, 2019.
- [21] R. Mahadevan, J. S. Edwards, and F. J. Doyle, "Dynamic flux balance analysis of diauxic growth in *Escherichia coli*," *Biophys. J.*, vol. 83, no. 3, pp. 1331–1340, Sep. 2002.
- [22] J. Anderson, J. C. Doyle, S. H. Low, and N. Matni, "System level synthesis," *Annual Reviews in Control*, vol. 59, no. 12, pp. 3238–3251, 2019.
- [23] Y.-S. Wang, N. Matni, and J. C. Doyle, "A system level approach to controller synthesis," *IEEE Trans. Autom. Control*, vol. 34, no. 8, pp. 982–987, 2019.
- [24] —, "Separable and localized system-level synthesis for large-scale systems," *IEEE Trans. Autom. Control*, vol. 63, no. 12, pp. 4234–4249, 2018.
- [25] S. Boyd, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.*, vol. 3, no. 1, pp. 1–122, 2010.
- [26] J. S. Li, "SLS-MATLAB: Matlab toolbox for system level synthesis," 2019. [Online]. Available: <https://github.com/sls-caltech/sls-code>
- [27] S.-H. Tseng and J. S. Li, "SLSPy: Python-based system-level controller synthesis framework," 2020. [Online]. Available: <http://arxiv.org/abs/2004.12565>
- [28] S. Diamond and S. Boyd, "CVXPY: A Python-embedded modeling language for convex optimization," *Journal of Machine Learning Research*, vol. 17, no. 83, pp. 1–5, 2016.