# Multi-Agent Trajectory Optimization with Decentralized NMPC

Adheesh Kadiresan, Arata Katayama, Joseph Nicol, Brandon Pae, Taimur Shaikh

***Abstract*— This paper explores decentralized Nonlinear Model Predictive Control (NMPC) for multi-agent trajectory optimization with GPU acceleration. We present integration of GPU-accelerated obstacle prediction and collision cost computation with robot collision avoidance. We developed a Karush-Kuhn-Tucker (KKT) solver for trajectory optimization. Both models meet our success criteria. Future work will focus on improving GPU performance and testing in larger-scale scenarios.**

## I. Introduction / Motivation

**Multi-agent trajectory optimization is a critical challenge in robotics, with applications ranging from drone swarms to autonomous vehicle coordination.** As the complexity of robotic systems increases, so does the need for efficient and scalable solutions to manage the interactions between multiple agents operating in shared environments. This paper explores the development of a decentralized Nonlinear Model Predictive Control (NMPC) approach for multi-agent trajectory optimization, with a focus on parallel computing techniques to enhance performance.

The problem of multi-agent trajectory optimization presents several key challenges:

- **Decentralization**: Each agent must independently plan its trajectory while considering the dynamic behavior of other agents and obstacles.
- **Online vs. Offline performance**: Solutions must be computed quickly enough to be applicable in dynamic, real-world scenarios.
- **Collision avoidance**: Agents must navigate shared spaces without colliding with each other or static obstacles.
- **Scalability**: The approach should be able to handle an increasing number of agents without a prohibitive increase in computational complexity.

Existing approaches to this problem can be broadly categorized into centralized and decentralized solutions. Centralized methods, while potentially offering globally optimal solutions, often suffer from scalability issues as the number of agents increases. Decentralized approaches, on the other hand, distribute the computational load across agents, potentially offering better scalability at the cost of global optimality.

Our work builds upon recent advancements in decentralized NMPC for multi-agent systems, particularly in the context of aerial robotics. We extend these concepts by incorporating parallel computing techniques, specifically leveraging GPU acceleration for certain computationally intensive tasks within the NMPC framework.

The main explorations of this paper are:

- A decentralized NMPC formulation for multi-agent trajectory optimization with dynamic collision avoidance.
- Implementation of GPU-accelerated algorithms for obstacle prediction and collision cost computation.
- Development of a custom Karush-Kuhn-Tucker (KKT) solver for trajectory optimization.

**Our aim is to create a framework for multi-agent trajectory optimization that can scale to handle complex scenarios with numerous agents and obstacles.**

The remainder of this paper is organized as follows: Section II provides background on NMPC and optimization methods. Section IV details our problem formulation. Section V describes our proposed solutions to each facet of the problem, including the decentralized NMPC formulation and parallel computing implementations. Section VI presents experimental results, followed by a discussion in Section VII. Finally, we conclude with insights and directions for future work in Section VIII.

## II. Background

### A. Nonlinear MPC

Decentralized NMPC has been widely studied as an approach to efficiently manage multi-agent systems. In [1], the authors describe applying this technique to drone swarms, under nonlinear dynamics and strict operational constraints. The work referenced in this paper builds upon the principles of NMPC by addressing challenges in formulating and solving optimal control problems (OCPs) in decentralized settings where agents operate independently but interact dynamically with their environment and one another.

The core problem is modeled as an OCP, which aims to optimize the trajectory and control inputs of individual agents while accounting for environmental constraints, moving obstacles, and interactions with neighboring agents. The OCP formulation includes the following key components:

- **Cost Function Design**
  The cost function comprises three primary terms:
  *State Reference Tracking:* Penalizes deviations between predicted states and desired reference states to ensure agents follow prescribed trajectories.
  *Control Input Regularization:* Penalizes deviations of predicted control inputs from desired inputs to encourage smooth control effort.
  *Proximity Awareness:* A proximity-based logistic function penalizes closeness to other agents. This smooth and bounded approach ensures numerical stability and avoids steep penalties that could destabilize the optimization process.

- **Constraints**
  *Collision Avoidance:* Incorporates hard constraints to ensure minimum separation between agents. These constraints are non-convex and only activate when the distance between agents falls below a predefined threshold, enhancing computational efficiency in non-critical scenarios.
  *External Disturbance Modeling:* Utilizes a model-based filter to estimate external disturbances affecting the agents, improving accuracy in dynamic environments.
  *Dynamic Constraints:* The system dynamics and constraints are discretized using the fourth-order Runge-Kutta method, capturing nonlinear behaviors with high accuracy.
- **Assumptions on Agent Behavior:**
  A constant velocity model is used to predict the motion of other agents, simplifying the computation without sacrificing significant accuracy in typical drone scenarios.

### B. KKT vs. Penalty Methods

There are many algorithms to solve constrained optimization problems. Two popular ones are Karush-Kuhn-Tucker (KKT) and penalty methods. These methods are fundamentally different in how they handle constraints.

The first major difference is that KKT enforces hard constraints. We can examine this general optimization problem:

$$\min_x f(x) \quad \text{s.t.} \quad g(x) = 0,$$

where $f(x)$ is the main cost function and $g(x)$ is the equality constraint. The KKT conditions are obtained by first constructing the Lagrangian, as shown below:

$$\min_x \max_{\lambda \geq 0} \mathcal{L}(x, \lambda) = f(x) + \lambda g(x)$$

These components can then be assembled into a KKT system:

$$\begin{bmatrix} Q & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} -q \\ c \end{bmatrix}$$

The key components are $Q$, which is the second derivative of $f(x)$, and $C$, which is the first derivative of $g(x)$. As a result, we are always searching for a valid pair of $x$, $\lambda$ when solving this system, which is why KKT enforces hard constraints.

On the other hand, penalty methods transform the constrained optimization problem into an unconstrained one by adding a penalty term to the main cost function. This allows soft constraints that are not strictly enforced before convergence, as shown below with the augmented Lagrangian:

$$\mathcal{L}_A(x, \lambda) = f(x) + \mu g(x)^T g(x) + \lambda g(x)$$

In the beginning, $\mu$ is small and we're solving a mostly unconstrained problem. However, we then increase $\mu$ and update $\lambda$ to converge to a feasible solution. It's important to note that large values of $\mu$ can lead to numerical instability.

Overall, penalty methods are appropriate for problems where an approximate (and potentially faster) solution is preferred.

## III. Related Work

Multi-agent trajectory optimization and collision avoidance have been extensively studied in robotics and artificial intelligence literature. This review focuses on key approaches and recent advancements in the field.

### A. Centralized Approaches

Centralized methods compute trajectories for all agents simultaneously, often providing globally optimal solutions. Sharon et al. (2015) proposed Conflict-Based Search (CBS), an optimal solver for the Multi-Agent Pathfinding (MAPF) problem [2]. CBS uses a two-level search, where a high-level search builds a constraint tree and a low-level search finds paths for individual agents. While CBS guarantees optimality, its scalability is limited for large numbers of agents.

### B. Decentralized Approaches

Decentralized methods distribute the computational load across agents, offering better scalability at the cost of global optimality. Luis et al. (2017) presented a decentralized Nonlinear Model Predictive Control (NMPC) approach for collision avoidance among multiple aerial robots [3]. Their method unifies trajectory tracking and collision avoidance into a single optimization problem, allowing each agent to independently plan its trajectory while considering other agents' behaviors. Kamel et al. (2017) expanded on this concept by incorporating state estimator uncertainty and communication delay for robust collision avoidance [1]. Their approach uses a potential field-like term in the cost function to penalize collisions between agents, with additional hard constraints to guarantee collision-free trajectories.

### C. GPU Acceleration

To address computational challenges in multi-agent trajectory optimization, recent work has explored GPU acceleration. Chen and Liu (2022) proposed a GPU-accelerated method for joint multi-agent trajectory optimization [4]. By reformulating the problem and leveraging parallel computing capabilities of GPUs, they achieved significant speedups compared to CPU-based methods.

### D. Dynamic Environments

Phillips and Likhachev (2011) introduced Safe Interval Path Planning (SIPP), an algorithm for path planning in dynamic environments [5]. SIPP uses a novel state representation that captures safe time intervals, allowing efficient planning around moving obstacles.

### E. Game Theory Approaches

Chen and Liu (2022) presented an approach using Dynamic Potential Games for efficient constrained multi-agent trajectory optimization [6]. This method formulates the problem as a potential game, allowing for decentralized computation while still achieving near-optimal solutions.

### F. Partial Linearization

Scutari et al. (2014) made contributions to Jacobi best-response algorithms where agents simultaneously solve convexified subproblems derived from the original problem [7]. The approach allowed parallel computation across convex subproblems assigned to each agent, leading to faster convergence. Moreover, their algorithm retains to convergence properties even with inexact solutions.

### G. Applications in Air Traffic Management

Bertrand et al. (2022) applied multi-agent path planning techniques to air traffic management, focusing on reducing collision risks [8]. Their work demonstrates the potential of these methods in real-world scenarios with strict safety requirements.

In conclusion, the field of multi-agent trajectory optimization has seen significant advancements in recent years, with a trend towards decentralized, computationally efficient methods that can handle dynamic environments and scale to large numbers of agents. GPU acceleration and game-theoretic approaches represent promising directions for future research, while applications in domains like air traffic management highlight the practical importance of these techniques.

## IV. Our Problem

### Decentralized NMPC with Dynamic Collision Avoidance

**To formulate the problem, we build upon the framework provided by Bose [9] for a basic Non-Linear Model Predictive Control (NMPC) problem.** Initially designed for a single agent in a constrained 2D state space (10x10 grid) with slow-moving obstacles following predetermined paths, we extend this framework to accommodate a decentralized multi-agent approach. Our extension necessitates a significant restructuring of the codebase to model each agent as an independent entity capable of solving its own trajectory optimization problem.

In our decentralized NMPC formulation, each agent must account for both the predicted trajectories of static obstacles and the dynamic trajectories of neighboring robots. This approach aligns with the work of Luis et al. [3], who demonstrated the effectiveness of decentralized NMPC for collision avoidance in multi-robot aerial systems.

It is important for our formulation to have a clear model of inter-agent communication. While it may be tempting to assume perfect and asymmetric information transmission of positions between robots, this assumption often fails to capture the complexities of real-world environments. Instead, we propose a more nuanced communication model that considers factors such as limited communication range, potential packet loss, and communication delays, as outline by Bertrand et al. [8].

### Parallelizing obstacle prediction computation

In Model Predictive Control (MPC) algorithms for multi-agent systems, the future state of each agent is dependent to the predicted positions of all other agents and obstacles in the environment. This interdependence results in a complex planning scenario, where each agent's path must be calculated while considering the dynamic movements of relatively close entities.

**Algorithm 1 shows the serial implementation of obstacle prediction by Bose [9]. The algorithm processes the future position of each obstacle sequentially within a single loop. For every obstacle, its current position and velocity are used to calculate a sequence of controlled inputs based on the current velocity, and then predict its future positions over the specified horizon.**

**Algorithm 1** Predict Obstacle Positions (Serial)

```
function PREDICT_OBSTACLE_POSITIONS(obstacles)
    obstacle_predictions ← empty list
    for each obstacle in obstacles do
        position ← obstacle.current_position
        velocity ← obstacle.current_velocity
        u ← repeat(velocity, horizon_length)
        prediction ← update_state(position, u, timestep)
        Add prediction to obstacle_predictions
    return obstacle_predictions
```

Even with a time complexity of O(n), with large enough numbers of agents and obstacles this sequential approach can potentially become a bottleneck, especially in dynamic systems where real-time decision making is critical. Since each obstacle's future position is calculated independently of others, the process is inherently parallelizable. By distributing the computations across multiple processing units, such GPU threads, the system can handle a much larger number of agents and obstacles without sacrificing responsiveness or accuracy. This parallel approach is crucial for meeting the demands of dynamic, real-time environments where rapid updates and decision-making are essential.

### Parallelizing Total Collision Cost Computation

Total collision cost is an important parameter in the code to quantify how much to penalize routes or steps that result in collisions between the robot and obstacles. In this implementation, the cost is a soft constraint, as collisions are not forbidden, but they incur a cost. The cost for a single

obstacle varies inversely with the distance between it and the robot agent. The cost algorithm for one obstacle is given below:

$$cost = \frac{Q_c}{1 + e^{\kappa(d-2*r_{robot})}}$$

Here, $Q_c$ is a tuning parameter, $\kappa$ is a smoothness parameter, $r_{robot}$ represents the radius of the robot, and $d$ represents the norm between the robot and the obstacle, which is just the Euclidean distance in a 2-dimensional grid.

The original algorithm to compute total collision cost iterated over the horizon distance (how many steps into the future to take into account when deciding the next step) and over each obstacle, in series, in a double-for loop. The cost between the robot and each obstacle was summed over each step within the horizon distance. This is inefficient and has potential to be parallelized. The solution will be discussed in the following section.

*Custom KKT Solver*

The existing method to calculate the optimal trajectory for the agent was velocity obstacles. It first found "problem regions" in velocity space that could cause collisions. With this constraint in mind, it did a sampling and found feasible velocities, as shown in the code below.

```
v_satisfying_constraints = check_constraints(
    v_sample, Amat, bvec)
```

**However, the performance of this method was unsatisfactory because the agent nearly collided with the obstacles in its trajectory, as shown in the figure below.**

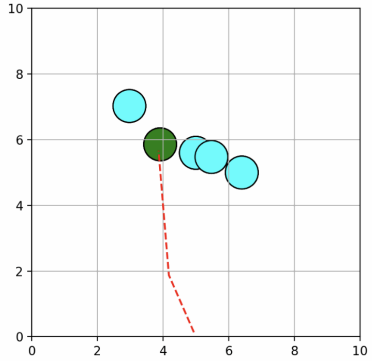


Fig. 1. Existing velocity obstacle method

As a result, we set out to develop a different method to solve for the optimal trajectory, as indicated in the subsequent section.

## V. Solution

*Decentralized NMPC Formulation*

**To address the challenge of decentralized NMPC with dynamic collision avoidance for multiple robots, we developed a comprehensive solution that integrates GPU-accelerated obstacle collision cost computation with a CPU-based robot-robot collision avoidance scheme.** Our approach builds upon the framework proposed by Luis et al., extending it to accommodate multiple agents and leveraging parallel computing capabilities[1].

The core of our solution revolves around a unified trajectory tracking and collision avoidance framework implemented as a constrained optimization problem solved in a receding horizon fashion. Each robot independently solves its own optimization problem, considering both moving obstacles and other robots as dynamic entities to avoid.

We formulate the NMPC problem for each robot as follows:

1) **State and Control Inputs**: The state vector includes the robot's (x,y) position, while control inputs are represented as velocity commands.
2) **Cost Function**: Our cost function comprises three main components:
   - Tracking cost to follow the reference trajectory
   - Collision cost which imparts a higher cost as a function of the distance between the current agent and another entity. This can further be decomposed into Obstacle Collision cost and Robot Collision cost.
3) **Constraints**: We implement bounds on control inputs to ensure physically feasible solutions.

We utilize a GPU-accelerated potential field-like method to compute collision costs with obstacles. When multiple robots are dealing with multiple obstacles, we anticipate this to improve runtime. We keep collision cost calculation for inter-robot avoidance on the CPU, due to time and resource constraints, however this approach allowed for faster iteration on the algorithm and more nuanced formulations, as GPU programming may have restricted our ability to model the robots more realistically.

Some other elements of the system we experimented with that are inspired by [3] are:

1) **Dynamic Replanning**: The receding horizon approach allows for continuous replanning, adapting to changes in the environment and other robots' behaviors.
2) **Priority Schemes**: We experiment with our framework to support hierarchical avoidance schemes, where robots can be assigned different priorities in collision avoidance maneuvers.
3) **Communication Delay**: Incorporating a tuned parameter that scales the position input from other robots to the current robot being solved for, which models real-world noisy communication channels.
4) **Uncertainty Propagation**: In [3], the state estimator uncertainty is propagated through subsequent iterations of the NMPC algorithm such that the potential-like 'safe region' field around a given robot gets larger.

*Parallelized Obstacle Prediction*

**To parallelize the obstacle predictions, we use CUDA which allows parallel computation on the GPU.** With

the original implementation by Bose [9] being in Python, we utilized the Just-In-Time (JIT) compiler provided by Numba to directly define a CUDA kernel in Python. The implementation of the CUDA kernel for obstacle prediction using JIT compiler is as follows:

```
@cuda.jit
def predict_obstacle_kernel_nonlinear(obstacles
    , obstacle_predictions, timestep):
    i = cuda.grid(1)
    if i < obstacles.shape[1]:
        x, y, vx, vy = obstacles[:, i]

        # Initialize lower_triangular_ones
            array

        for j in range(HORIZON_LENGTH):
            t = j * timestep
            vx_t = vx * (1 + 0.1 * t)
            vy_t = vy * (1 + 0.1 * t)

            # Calculate dx and dy using
                lower_triangular_ones

            obstacle_predictions[i, j, 0] = x +
                dx
            obstacle_predictions[i, j, 1] = y +
                dy
```

Our CUDA implementation assigns each GPU thread to predict the position of a single obstacle. While Bose [9] had previously defined an update_state function for state updates, it was incompatible with CUDA functions. As a result, we implemented a similar non-linear model for state updates within our CUDA kernel, based on the original implementation. This approach allows us to perform state updates directly within the GPU-accelerated obstacle prediction process, ensuring compatibility and maintaining the non-linear characteristics of the original model.

Furthermore, similarly with Bose's original implementation we used a lower triangular matrix for efficient and accurate obstacle position prediction over time. This technique effectively captures the cumulative impact of velocity changes, and implements a numerical integration approach that is suited for discretized continuous systems.

The CUDA kernel is then called using the following configurations:

```
threads_per_block = 256
blocks_per_grid = (obstacles.shape[1] +
    threads_per_block - 1)

predict_obstacle_kernel_nonlinear[
    blocks_per_grid, threads_per_block](
        d_obstacles,
        d_obstacle_predictions,
        NMPC_TIMESTEP)
```

This arithmetic for blocks_per_grid ensures that there are enough blocks to process all obstacles, even if the number of obstacles is not evenly divisible by the number of threads_per_block.

*Parallelized Total Collision Cost Computation*

The computation for total collision cost was parallelized from the original algorithm provided in the problem section. The updated algorithm is provided below, and will be subsequently explained.

```
@cuda.jit
def total_collision_cost(robot, obstacles,
    total_cost_array):

    i, j = cuda.grid(2)  # Get 2D grid indices
    HORIZON_LENGTH, num_obstacles = robot.shape
        [0] // 2, obstacles.shape[0]

    if i < HORIZON_LENGTH and j < num_obstacles:
        rob = robot[2 * i: 2 * i + 2]
        obs = obstacles[j, 2 * i: 2 * i + 2]
        cost = collision_cost(rob, obs)
        cuda.atomic.add(total_cost_array, 0,
            cost)

@cuda.jit(device=True)
def collision_cost(x0, x1):
    d = math.sqrt((x0[0] - x1[0]) ** 2 + (x0[1]
        - x1[1]) ** 2)  # Avoid np.linalg.norm
    return Qc / (1 + math.exp(kappa * (d - 2 *
        ROBOT_RADIUS)))
```

The above is invoked with the following call that specifies the number of blocks per thread and threads per block.

```
total_collision_cost[blocks_per_grid,
    threads_per_block](x_robot,
    obstacle_predictions, total_cost_array)
```

In our implementation, the threads per block was (16, 16), meaning 256 threads were launched, which would be accessed with 2-dimensional indeces. Blocks per grid was also a 2-dimensional setup, with the number of x dimensions being the horizon length divided by 16, and the y dimensions being the number of obstacles divided by 16. This means that as long as the number of obstacles is under 16 (like in the case with 10), only one block would be launched.

In the algorithm, the first function calls the second function. In the first function, instead of a double for-loop, the algorithm parallelizes each step within the horizon distance and between all robots. Controlling which (robot, obstacle, time step) corresponds to a thread is accomplished with simple if statements. These ensure each index is not out-of-bounds with respect to the horizon length or number of obstacles. Then, the algorithm directly uses the thread indexes to access the relevant robot and obstacle positions. Then, the second function is called to compute the cost.

The second function is mostly similar to before, except in order to run on the GPU, we replaced the NumPy function calls with ones from Python's math library. This involved computing the norm (or distance) manually.

Finally, the first function sums up all the costs with an atomic add. This is necessary to ensure that parallel adds do not conflict and remain accurate. However, this does slow down the computation since atomic adds have a serial component.

*Custom KKT solver*

**We implemented a method inspired by KKT to calculate the optimal trajectory for the agent (i.e. find the next state and velocity).**

Our implementation uses a modified version of KKT. In particular, we implemented the following function:

```
def kkt_cost(u):
    # base cost
    cost = total_cost(u, robot, obstacles,
        v_desired)

    # MU weights the constraint violation
    constraint_violation = np.maximum(0, np.abs
        (u) - VMAX)
    return cost + MU * np.sum(
        constraint_violation)
```

**While the typical KKT system uses hard constraints, our solution uses penalty methods to transform the constrained optimization problem into an unconstrained one.** In particular, we include a $\mu$ term which adjusts the importance of the constraint, as shown above.

This kkt cost function is then used with scipy's SLSQP (Sequential Least Squares) solver. We used the SLSQP method because it is a high level version of KKT conditions internally and abstracts the optimization process.

```
# SLSQP (Sequential Least SQuares Programming)
    - uses KKT under the hood
res = minimize(
    kkt_cost,           # Objective function
    u0,                 # Initial guess for u
    method='SLSQP',     # Optimization method
    bounds=bounds       # Variable bounds
)
```

We also introduce additional constraints, including upper and lower bounds on the velocity, as shown below:

```
bounds = Bounds([-VMAX, -VMAX], [VMAX, VMAX])
```

Overall, our modified version of KKT was inspired by the existing NMPC code, which provided structure for the optimization problem. We found that this approach was more practical and fast than directly solving the KKT conditions by hand.

**After constructing this method, we initially observed that robots briefly contacted obstacles before 'avoiding' collisions.** In reality, the center of our nodes did not collide, but the surrounding shape appeared to. We addressed this by increasing `SAFETY_MARGIN`, allowing robots to register and start avoiding obstacles earlier. At this same point, we increased `MU`, `Qc`, and `kappa` to strengthen this collision avoidance behavior.

- `MU` (Penalty Parameter): We increased MU to 200. This parameter is related to the Lagrange multipliers in the KKT conditions and strengthens the enforcement of constraints in the optimization problem.
- `Qc` (Collision Cost Weight): Qc was set to 200, which affects the quadratic programming formulation of the problem. A higher Qc value places more emphasis on collision avoidance in the process.
- `kappa` (Collision Cost Shape Parameter): We increased kappa to 15. This influences the overall responsiveness of the system to potential collisions.
- `SAFETY_MARGIN`: We increased the safety margin to 2.0. This will ensure that the optimization problem considers a larger buffer zone around the robots, prompting earlier avoidance actions.

These adjustments collectively enhance the collision avoidance capabilities. This essentially amplifies the result of getting too close to obstacles. The algorithm now prioritizes safer paths. The larger `SAFETY_MARGIN` is essentially an expanded but invisible bubble around the node, triggering avoidance earlier. Sometimes this will result in slightly longer paths, but will significantly reduce collisions, especially in complex environments with many obstacles. In the real world, usually the cost of a collision will far outweigh the cost of a slightly longer route.

## VI. Results

*Parallelized Obstacle Collision Computation*

When integrating both the parallelized collision cost and parallelized obstacle prediction kernels into the decentralized multi-agent NMPC formulation, we ran benchmarks. To obtain these, we ran our full NMPC simulation for 3 robots and 4 obstacles 10-20 times, and obtained average timing results, presented in the three figures below.

Figure 2 shows the absolute timing contributions of a CPU implementation of the obstacle collision cost, the associated GPU kernel we wrote (excluding I/O back to the host), and the total GPU time (including I/O)

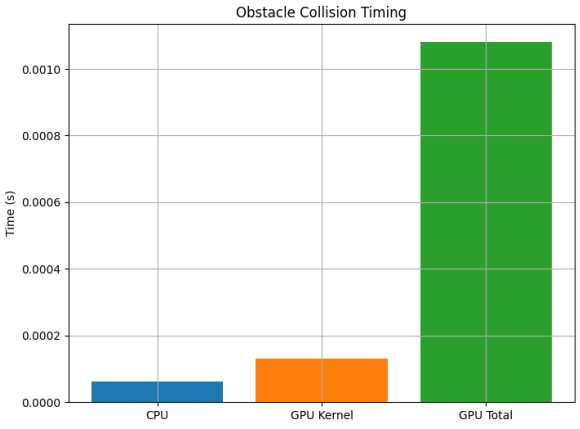


Fig. 2. Timing comparison of Obstacle Collision Cost Computation

**We see that our GPU implementation actually significantly slowed down average execution time for this particular part of the algorithm.** One thing to note is that the time on the GPU (Orange Bar) is only a fraction of total GPU time, leading us to believe that I/O-bound activity is the main culprit for slowdown. We discuss this further in the next section

Figure 6 shows the absolute timing contributions of a CPU implementation of the obstacle prediction function, the associated GPU kernel we wrote (excluding I/O back to the host), and the total GPU time (including I/O)

Here, we see an even bigger relative slowdown when switching to the GPU, with an approximately 30x slowdown.

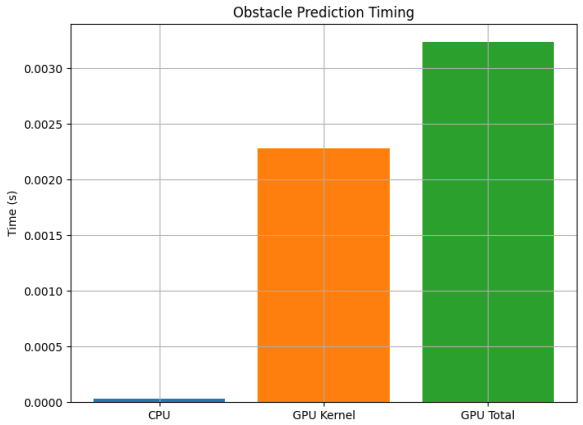


Fig. 3. Timing comparison of Obstacle Prediction Computation

Moreover, it appears that additional time consumed by I/O between the GPU and CPU makes up a smaller portion of overall execution.

Finally, Figure 4 depicts the breakdown for both kernels between actual kernel time and additional time (i.e. copying data to the host).

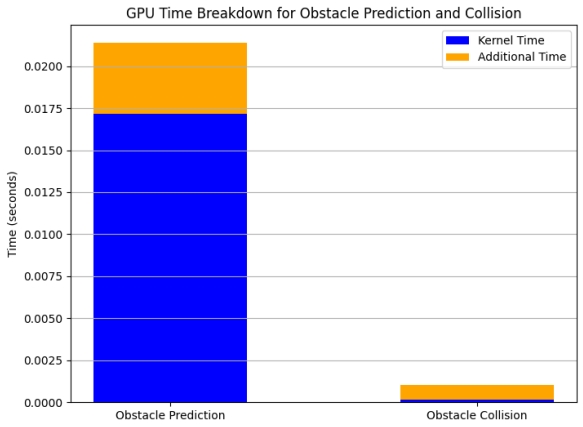


Fig. 4. Kernel Timing Comparisons between time on the GPU and additional (I/O) time

This Figure puts into perspective how much slower the prediction is than the collision cost calculation, as well as the relative contributions of non-kernel activity to each.

*Parallel Total Collision Cost Computation*

The runtimes for parallel vs sequential total cost computation are provided below. This first table represents a comparison for four obstacles.

| **Method** | **Avg Total Runtime (4 s.f.)** |
|---|---|
| Sequential | 16.46 seconds |
| Parallel | 35.29 seconds |

*1) 4 obstacles:* Here, we see that the GPU version actually takes longer to run. This is likely due to the overhead that is the time it takes to copy data from the CPU memory to the GPU memory. For small input sizes, this cost will dominate and cause the GPU to be slower, as we see above.

The following table presents the comparison for 100 obstacles.

*2) 100 obstacles:*

| **Method** | **Avg Total Runtime (5 s.f.)** |
|---|---|
| Sequential | 1:24.40 seconds |
| Parallel | 2:24.57 seconds |

Here, we also see that the GPU is slower, but by a smaller factor than in the 4 obstacles case. (This is most likely from the GPU parallelization.

The robot path for the 4-obstacle case is shown below in several frames:

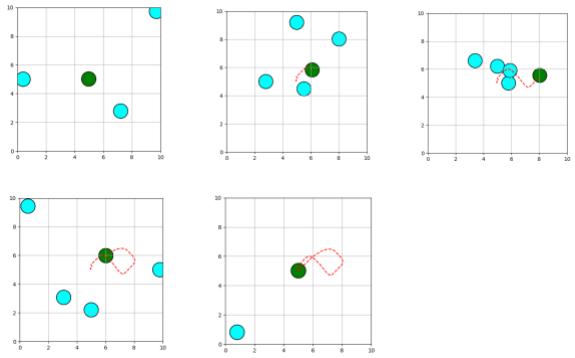

Fig. 5. Timing comparison of Total Cost Computation with 4 obstacles

The robot path for the 100-obstacle case is shown below in several frames:

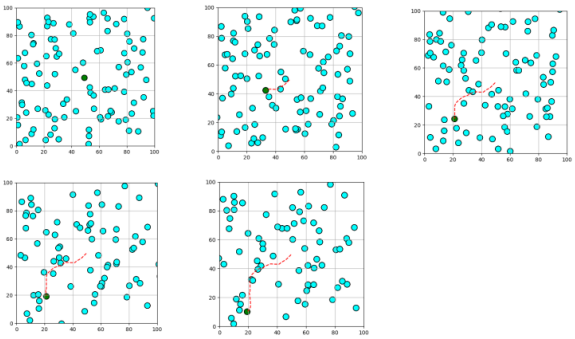

Fig. 6. Timing comparison of Total Cost Computation with 100 obstacles

*Custom KKT solver*

**In our performance testing over 100 iterations, the KKT method actually showed an increase in runtime compared to the original method.** However, we expect this difference to be offset in more complex scenarios with more collision opportunities.

| Method | Avg Total Runtime (3 s.f.) |
|---|---|
| Original Method | 5.68 seconds |
| KKT Method | 5.89 seconds |

We expect our KKT method's accuracy to improve with scale, particularly in environments with more obstacles. This idea comes from the method's enhanced collision avoidance capabilities and how it features a more comprehensive consideration of the environment. This will be tested and proved within our next steps. When we integrate the KKT method with both parallelized and distributed code implementations, we can utilize the 100-obstacle model with the integrated KKT method to retrieve more precise performance metrics.

Our KKT implementation successfully navigates robots around multiple, randomly projected obstacles. The method's effectiveness comes from its dynamic adjustment based on cost function, balancing path efficiency with collision avoidance. The ability to handle random obstacle configurations shows adaptability for real-world applications where unpredictable events are common.

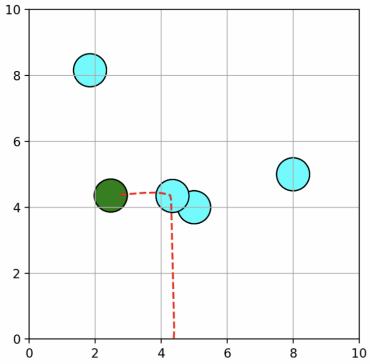


Fig. 7. Our KKT method

## VII. Discussion

*GPU Performance Analysis*

**The implementation of GPU-accelerated algorithms for obstacle prediction and collision cost computation in our decentralized multi-agent NMPC framework yielded unexpected results, providing valuable insights into the challenges of parallelizing complex robotics algorithms.**

Our timing benchmarks reveal that the GPU implementations of both the obstacle collision cost computation and obstacle prediction functions unexpectedly increased execution time compared to their CPU counterparts. This counter-intuitive outcome warrants a deeper analysis of the factors contributing to the performance degradation.

*1) Obstacle Collision Cost Computation:* The GPU kernel for collision cost calculation showed a modest slowdown compared to the CPU version. However, the most significant contributor to the increased execution time was the data transfer overhead between the CPU and GPU. As evident in Figure 4, the actual GPU computation time (orange bar) is only a fraction of the total GPU time (gray bar), indicating that the process is primarily I/O-bound.

This observation suggests that for the current problem size and complexity, the benefits of parallel computation on the GPU are outweighed by the costs of data transfer. The relatively small number of obstacles (4) and robots (3) in our test scenario may not provide sufficient computational workload to amortize the overhead of GPU memory allocation and data transfer.

*2) Obstacle Prediction Computation:* The obstacle prediction function exhibited an even more pronounced performance degradation, with approximately a 30-fold increase in execution time when ported to the GPU (Figure 6). Unlike the collision cost computation, the prediction algorithm shows a smaller proportion of time spent on I/O operations, suggesting that the kernel itself is not efficiently utilizing the GPU's parallel architecture.

This substantial slowdown could be attributed to several factors:

1) **Thread Divergence**: The nonlinear nature of our obstacle prediction algorithm may lead to divergent execution paths among GPU threads, reducing parallel efficiency.
2) **Memory Access Patterns**: The algorithm's memory access patterns may not be optimized for coalesced memory operations on the GPU, resulting in suboptimal memory bandwidth utilization.
3) **Kernel Complexity**: The prediction algorithm is more computationally involved than the collision calculation, and includes operations that must be carefully thought out when implementing on a kernel. Given some more time, we believe that we would be able to achieve significant improvements in our results in this specific area.

### A. KKT Method

**Our approach to KKT solver shows a balance between theoretical optimization principles and practical implementation considerations.** When we modified our first KKT method to use penalty functions instead of hard constraints, we transformed the problem to be unconstrained.

The use of SLSQP method as the underlying solver helps to leverage existing implementations of KKT conditions. Initial observations of brief obstacle contacts highlight the balance required in tuning optimization parameters for real life scenarios. The adjustments we made to the key parameters demonstrated how optimizations of such systems can be calculated.

The increased `SAFETY_MARGIN`, introduces a trade-off between path optimality and collision avoidance. While this can result in longer trajectories, it enhances safety; which is more of an important consideration in real-world robotic applications.

However, we raise questions about performance in more complex scenarios. Integrating parallelized methods with the KKT solver, will help to leverage GPU acceleration for larger-scale problems.

## VIII. Conclusion

Our research on multi-agent trajectory optimization has provided some valuable insights and results. We successfully implemented a parallelized, decentralized NMPC framework and a KKT solver that navigates robots around multiple obstacles, showing adaptability in complex environments.

- The KKT solver successfully navigated robots around multiple obstacles, balancing path efficiency and collision avoidance.
- GPU implementations for obstacle prediction and collision cost computation produced accurate results and avoided collisions, but increased execution time, primarily due to data transfer overhead and inefficient utilization of GPU architecture.
- Adjusting parameters like `SAFETY_MARGIN`, `MU`, `Qc`, and `kappa` with the KKT enhanced collision avoidance, but created discrepancy between path optimality and safety.
- The decentralized NMPC integrated GPU-accelerated obstacle collision cost computation with CPU-based robot-robot collision avoidance, allowing for independent optimization by each robot.

### A. Next Steps

- Integrate the KKT method with both parallelized and distributed code implementations.
- Optimize memory access patterns to improve coalesced memory operations on the GPU
- Reduce thread divergence in the NMPC obstacle prediction calculation through warp-level optimization
- Conduct performance testing with an increased number of agents to obtain more accurate scalability results.
- Utilize the 100-obstacle model with the integrated KKT method to retrieve more precise performance metrics.

These goals will provide a more comprehensive evaluation of both parallelized decentralized NMPC and KKT method's effectiveness and efficiency in complex, large-scale scenarios.

### B. Contributions

Brandon worked on implementing the KKT solver to replace the existing velocity obstacles method. This process involved understanding the drawbacks of the existing method, how to set up the optimization problem, and implement a custom cost function and KKT solver, which included a penalty for soft constraints. He also worked on fine-tuning the visualizations.

Joe focused on researching and fine-tuning the KKT solver. Upon developing the initial KKT method, he observed issues with the results and the robot's actions, and optimized the system to enhance its collision avoidance behavior. He worked with the parameters mentioned to determine the ideal and optimal solution. Additionally, Joe investigated the original methodology behind the KKT solver, while also managing and recording timing and performance metrics of this system. Both Joe and Brandon worked together on the KKT solver, managed the repository, and formatted results.

Adheesh worked on parallelizing the total cost computation. Looking over the nonlinear MPC code, he noticed a double for-loop in this section that indicated some room for optimization. He optimized this algorithm by parallelizing the algorithm over the horizon distance and the number of obstacles. Adheesh also created and ran the associated benchmarks and created the figures. He also contributed to the background research into nonlinear MPC.

Arata developed a GPU-accelerated approach for obstacle prediction, adapting Bose's original serial NMPC implementation [9]). He utilized Numba's (JIT) compiler to create a CUDA kernel, enabling parallel computation on the GPU. Due to CUDA's incompatibility with the original numpy-based state update function, Arata reconstructed the nonlinear model for state updates within the CUDA kernel. This adaptation allowed for efficient parallel processing of obstacle predictions while maintaining the non-linear characteristics of the original model.

Taimur's work focused on modelling the decentralized NMPC problem, from formulating the multi-agent state space to deciding the method of handling inter-robot avoidance (which was not present in the original repository [9]). Taimur integrated Arata's and Adheesh's parallelized kernels for obstacle prediction and obstacle collision cost respectively into the decentralized environment, and ran the associated benchmarks and created the figures.

### C. Running Code

To run the KKT code, please visit our repository here and run the following commands:

```
cd ./decentralized
python3 decentralized.py -f kkt/new_kkt.mp4 -m kkt
```

To run the parallelized decentralized code, we use a specific branch in our repo:

```
git checkout taimur/integrate-decentralized-parallel-cost cd ./decentralized
```

In this branch, open the notebook 'RoboticsTest.ipynb' to generate a visualization for the NMPC simulation.

## REFERENCES


[1] M. Kamel, J. Alonso-Mora, R. Siegwart, and J. Nieto, "Nonlinear model predictive control for multi-micro aerial vehicle robust collision avoidance," Mar 2017. [Online]. Available: https://arxiv.org/abs/1703.01164

[2] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015.

[3] C. E. Luis, M. Vukosavljev, and A. P. Schoellig, "Decentralized nonlinear model predictive control for collision avoidance of multiple aerial robots," *arXiv preprint arXiv:1703.01164*, 2017.

[4] Y. Chen and C. Liu, "Gpu acceleration of joint multi-agent trajectory optimization," *arXiv preprint arXiv:2206.08963*, 2022.

[5] M. Phillips and M. Likhachev, "Sipp: Safe interval path planning for dynamic environments," in *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011, pp. 5628–5635.

[6] Y. Chen and C. Liu, "Efficient constrained multi-agent trajectory optimization using dynamic potential games," *arXiv preprint arXiv:2206.08963*, 2022.

[7] G. Scutari, F. Facchinei, P. Song, D. P. Palomar, and J.-S. Pang, "Decomposition by partial linearization: Parallel optimization of multi-agent systems," *IEEE Transactions on Signal Processing*, vol. 62, no. 3, pp. 641–656, 2013.

[8] S. Bertrand, J. Morio, S. Hadjres, and D. Delahaye, "Reducing collision risk in multi-agent path planning: Application to air traffic management," *arXiv preprint arXiv:2212.04122*, 2022.

[9] A. Bose, "Multi agent path planning," https://github.com/atb033/multi_agent_path_planning, 2024.