Real-time Optimization-based Planning in Dynamic Environments using GPUs

Chonhyon Park and Jia Pan and Dinesh Manocha

***Abstract*— We present a novel algorithm to compute collision-free trajectories in dynamic environments. Our approach is general and does not require a priori knowledge about the obstacles or their motion. We use a replanning framework that interleaves optimization-based planning with execution. Furthermore, we describe a parallel formulation that exploits a high number of cores on commodity graphics processors (GPUs) to compute a high-quality path in a given time interval. We derive bounds on how parallelization can improve the responsiveness of the planner and the quality of the trajectory.**

I. Introduction

Robots are increasingly used in dynamic or time-varying environments. These scenarios are composed of moving obstacles, and it is important to compute collision-free trajectories for navigation or task planning. Some of the applications include automated wheelchairs, manufacturing tasks with robots retrieving parts from moving conveyors, air and freeway traffic control, etc. The motion of the obstacles can be unpredictable and new obstacles may be introduced in the environment. As a result, we need to develop appropriate algorithms for planning and executing appropriate trajectories in such dynamic scenes.

There is extensive work on motion planning. Some of the widely used techniques are based on sample-based planning, though they are mostly limited to static environments. There is recent work on extending sample-based planning techniques to dynamic scenes by incorporating the notion of time as an additional dimension in the configuration space [1], [2], [3]. However, the resulting algorithms may not generate smooth paths or handle dynamic constraints in real time. Other techniques for dynamic environments are either limited to local collision avoidance with the obstacles, or make some assumptions about the motion of dynamic obstacles.

In this paper, we address the problem of collision-free trajectory computation in dynamic scenes. In order to deal with unpredictable environments, we use replanning algorithms that interleave planning with execution [2], [4], [5], [6]. In these cases, the robot may only compute partial or sub-optimal plans in the given time interval. In order to generate smooth paths and handle dynamic constraints, we combine replanning techniques with optimization-based planning [7], [8], [9].

We present a novel parallel optimization-based motion planning algorithm for dynamic scenes. Our planning algorithm optimizes multiple trajectories in parallel to explore a broader subset of the configuration space and computes a high-quality path. The parallelization improves the optimality of the solution and makes it possible to compute a safe solution for the robot in a shorter time interval. We map our multiple trajectory optimization algorithm to many-core GPUs (graphics processing units) and utilize their massively parallel capabilities to achieve 20-30X speedup over serial optimization-based planner. Furthermore, we derive bounds on how parallelization improves the responsiveness and the quality of the trajectory computed by our planner. We highlight the performance of our parallel replanning algorithm in the ROS simulation environment with a 7-DOF robot and and human-like dynamic obstacles.

The rest of the paper is organized as follows. In Section 2, we give a brief overview of prior work on motion planning in dynamic environments and optimization-based planning. We present an overview of optimization-based planning and execution framework in Section 3. In Section 4, we describe the parallel replanning algorithm and analyze its responsiveness and quality in Section 5. We highlight its performance in Section 6.

II. Related Work

In this section, we give a brief overview of prior work on motion planning in dynamic environments, optimization-based planning and parallel algorithms for motion planning.

A. Motion Planning in Dynamic Environments

Many approaches for motion planning in dynamic environments assume that the trajectories of moving objects are known a priori. Some algorithms discretize the continuous trajectory and model dynamic obstacles as static obstacles within a short horizon [10]. Other techniques compute an appropriate robot velocity that can avoid a collision with moving obstacles during a short time step [11], [12]. The state space for planning in a dynamic environment is given as $\mathcal{C} \times T$, i.e., the Cartesian product of configuration space and time. Some RRT variants can handle continuous state space directly [6], while other methods discretize the state space and use classic heuristic search [13], [14] or roadmap based algorithms [15].

Some planning algorithms for dynamic environments [15], [13] assume that the inertial constraints, such as acceleration and torque limit, are not significant for the robot. Such assumptions imply that the robot can stop and accelerate instantaneously, which may not be feasible for physical robots. Moreover, these algorithms attempt to find a good solution for path planning before robot execution starts. In

Chonhyon Park, Jia Pan and Dinesh Manocha are with the Department of Computer Science, University of North Carolina at Chapel Hill. E-mail: {chpark, panj, dm}@cs.unc.edu. The accompanying video can be found at http://gamma.cs.unc.edu/ITOMP.

many scenarios, the planning computation can be expensive. As a result, path planning before execution strategy can lead to long delays during the robot's movement and may cause collisions for robots operating in environments with fast dynamic obstacles. One solution to overcome these problems is based on real-time replanning, which interleaves planning with execution so that the robot may only compute partial or sub-optimal plans for execution to avoid collisions. Different algorithms can be used as the underlying planners in the real-time replanning framework, including sample-based planners [2], [4], [6] or search-based methods [5], [16]. Most replanning algorithms use fixed time steps when interleaving between planning and execution [6]. Some recent work [2] computes the interleaving timing step in an adaptive manner to balance between safety, responsiveness, and completeness of the overall system.

B. Optimization-based planning

Optimization techniques can be used to compute a robot trajectory that is optimal under some specific metrics (e.g., smoothness or length) and that also satisfies various constraints (e.g., collision-free and dynamics constraints). Some algorithms assume that a collision-free trajectory is given and it can be refined or smoothened using optimization techniques. These include 'shortcut' heuristic [17], elastic bands or elastic strips planning [18], [19]. Other algorithms relax the assumptions about the initial path and may start with an in-collision path. Some recent approaches, such as [7], [8], [20], directly encode the collision-free constraints using a global potential field and compute a collision-free trajectory for robot execution. These methods typically represent various constraints (smoothness, torque, etc.) as soft constraints in terms of additional penalty terms to the objective function. In case the underlying robot has to satisfy hard constraints, e.g., dynamic constraints needed to maintain the balance for humanoid robots, the trajectory computation problem is solved using constrained optimization [21], [22] and this computation tends to be expensive for realtime applications.

C. Parallel Algorithms

Due to the rapid advances in multi-core and many-core commodity processors, designing efficient parallel planning algorithms that can benefit from their computational capabilities is an important topic in robotics. Many parallel algorithms have been proposed for motion planning by utilizing the properties of configuration space [23]. Moreover, techniques based on distributed representation [2] can be easily parallelized. In order to deal with very high-dimensional or challenging scenarios, distributed sample-based techniques have also been proposed [24], [25], [26].

The rasterization capabilities of a GPU can be used for real-time motion planning of low DOF robots [27] or for improving the sample generation in narrow passages [28]. Recently, the GPUs have been exploited to accelerate sampling-based motion planners in high-dimensional spaces, including sample-based planning [29], RRT algorithms [30], and search-based planning [31].

III. Overview

Our real-time replanning algorithm is based on optimization-based planning and uses parallel techniques to handle arbitrary dynamic environments. In this section, we describe the underlying framework for optimization-based planning and give an overview of our planning and execution framework.

A. Optimization-based Planning

Traditionally, the goal of motion planning is to find a collision-free trajectory between the start configuration and the goal configuration. Optimization-based planning reduces trajectory computation to an optimization problem that minimizes the costs corresponding to collision-free, smoothness, and dynamics constraints. Specifically, the start configuration vector $\mathbf{q}_{start}$ and the goal configuration vector $\mathbf{q}_{end}$ are defined in the configuration space $\mathcal{C}$ of a robot. In this case, the dimension $\mathcal{D}$ of $\mathcal{C}$ is equal to the number of free joints in the robot. There may be several static and dynamic obstacles in the environment corresponding to rigid bodies. We assume that a solution trajectory has a fixed time duration $\mathcal{T}$, and discretize it into N (excluding the two endpoints $\mathbf{q}_{start}$ and $\mathbf{q}_{end}$) waypoints equally spaced in time. The trajectory can be also represented as a vector $\mathbf{Q} \in \mathcal{R}^{D \cdot N}$:

$$\mathbf{Q} = [\mathbf{q}_1^T, \mathbf{q}_2^T, ..., \mathbf{q}_N^T]^T. \quad (1)$$

Similarly to the previous work [8], [7], [9], we define the objective function of our optimization problem as:

$$\min_{\mathbf{q}_1,...,\mathbf{q}_N} \sum_{i=1}^{N} (c_s(\mathbf{q}_i) + c_d(\mathbf{q}_i) + c_o(\mathbf{q}_i)) + \frac{1}{2}\|\mathbf{AQ}\|^2, \quad (2)$$

where the three cost terms $c_s(\cdot)$, $c_d(\cdot)$, and $c_o(\cdot)$ represent the static obstacle cost, dynamic obstacle cost, and the problem specific additional constraints, respectively. $\|\mathbf{AQ}\|^2$ represents the smoothness cost which is computed by the sum of squared accelerations along the trajectory, using the same matrix $\mathbf{A}$ proposed by Ratliff et al. [8]. The solution to the optimization problem in (2) corresponds to the optimal trajectory of the robot. Our algorithm ensures that the costs corresponding to $c_s(\cdot)$, $c_d(\cdot)$, and $c_o(\cdot)$ are larger than $\frac{1}{2}\|\mathbf{AQ}\|^2$, if they are nonzero, i.e., the costs corresponding to collision-free trajectories are smaller than the costs of any trajectories that have collisions with obstacles, regardless of the trajectory smoothness.

In order to compute static and dynamic obstacle costs, we use the signed Euclidean Distance Transform (EDT) and geometric collision detection. As in previous work by Ratliff et al [8], we divide the workspace into a 3D voxel grid and precompute the distance to the boundary of the nearest static obstacle with each voxel. Moreover, we approximate the robot's shape $\mathcal{B}$ by using a set of overlapping spheres $b \in \mathcal{B}$. In this case, the static obstacle cost for a configuration

$\mathbf{q}_i$ can be computed by table lookup in the voxel map as follows:

$$c_s(\mathbf{q}_i) = \sum_{b \in \mathcal{B}} \max(\epsilon + r_b - d(\mathbf{x}_b), 0)\|\dot{\mathbf{x}}_b\|, \qquad (3)$$

where r_b is the radius of one sphere b, $\mathbf{x}_b$ is the 3D point of sphere b computed from the kinematic model of the robot at configuration $\mathbf{q}_i$, $d(x)$ is the signed EDT for a 3D point x, and ϵ is a small safety margin between robot and the obstacles.

EDT can be efficiently used to compute the cost of static obstacles, since it requires only a simple table lookup after one-time initialization. However, using EDT for dynamic obstacles requires recomputation of EDT during each step, which can be expensive. Therefore, we use geometric collision detection between the robot and dynamic obstacles to formulate the cost function for dynamic obstacles. Object-space collision detection algorithms based on bounding-volume hierarchies [32] are used to compute the dynamic obstacle cost efficiently.

In real-world applications, we cannot make assumptions about the future motion or trajectory of the obstacles. We can only locally estimate the trajectory based on sensor data. In order to guarantee the safety of the planned trajectory in a local time interval, we compute a conservative local bound on the trajectories of dynamic obstacles and use them for the collision detection. They are computed based on computing a conservative bound on the swept volume of the objects along the estimated trajectory. The allowed sensing error is considered to determine the size of bounds. The conservative bound for an obstacle O^d for the time interval $[t_0,\ t_1]$ is computed as:

$$\overline{O}^d([t_0, t_1]) = \bigcup_{t \in [t_0, t_1]} c(1 + e_s \cdot t) O^d(t), \qquad (4)$$

where e_s is the maximum allowed sensing error. As the sensing error increases, the bound becomes more conservative. When an obstacle has a constant velocity, it is guaranteed that the conservative bound includes the obstacle corresponding to $c = 1$. However, if an obstacle changes its velocity, we have to use a larger value of c in our conservative bound. The bound can be guaranteed to include the obstacle if we know the maximum acceleration of the obstacle.

B. Planning and Execution Framework

In order to improve the responsiveness of the robot in dynamic environments, we use a replanning approach that was previously used for sampling-based motion planning [4], [2]. Instead of planning and executing the entire trajectory at once, this formulation interleaves the planning and execution threads within a small time interval Δ_t. This approach allows us compute new estimates on the local trajectory of the obstacles based on the most current sensor information. The time interval Δ_t can be changed adaptively according to replanning performance [2]. During each planning step, we compute an estimate of the position and velocity of

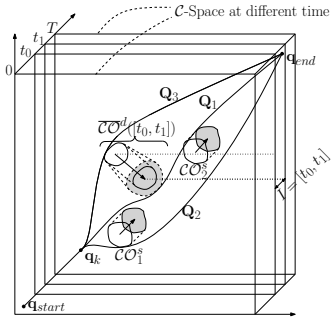


Fig. 1: *Multiple trajectories that arise in the optimization-based motion planning. The coordinate system shows how the configuration space changes over time as the dynamic obstacles move over time: each plane slice represents the configuration space at time t. In the environment, there are three C-obstacles: the two static obstacles CO_1^s, CO_2^s and the dynamic obstacle CO^d. The planned trajectories start at time 0, stop at time T, and are represented by a set of way points $\mathbf{q}_{start}$, $\mathbf{q}_1$, ..., $\mathbf{q}_k$, ..., $\mathbf{q}_N$, $\mathbf{q}_{end}$. The three trajectories for the time interval $I = [t_0, t_1]$ are generated with different random seeds and represent different solutions to the planner in these configurations corresponding to the dynamic obstacles.*

dynamic obstacles using the senor data. Next, a conservative bound on dynamic obstacles during the local time interval is computed using these values, and the planner uses this bound to compute the cost for dynamic obstacles. This cost is only used during the time interval Δ_t, as the predicted positions of dynamic obstacles may not be valid over a long time horizon. This bound guarantees the safety of the trajectory during the planning interval; however the size of the bound increases as the planning interval increases. Large conservative bounds make it hard for the planner to compute a solution in the given time or they result in a less optimal solution because of the time constraints. Hence, it is important to choose a short time interval to improve the responsiveness of the robot. Our goal is to exploit the parallelism in commodity processors to improve the efficiency of the optimization-based planner. This parallelism results in two benefits:

- The faster computation allows us to use shorter time intervals, which can improve the responsiveness and safety for robots working in fast changing environments.
- Based on parallel threads, we can try to compute multiple trajectories corresponding to different seed values, and thereby explore a broader configuration space to compute a more optimal solution, as illustrated in Fig. 1.

IV. Parallel Replanning

Nowadays, all commodity processors have multiple cores. Even some of the robot systems are equipped with multi-core CPU processors (e.g. Quad-Core i7 Xeon Processors in PR2 robot). Furthermore, these robot systems provide expansibility in terms of using many-core accelerators, such as graphics processing units (GPUs). These many-core accelerators are massively parallel processors, which offer a very high peak performance (e.g. up to 3 TFLOP/s on NVIDIA Kepler

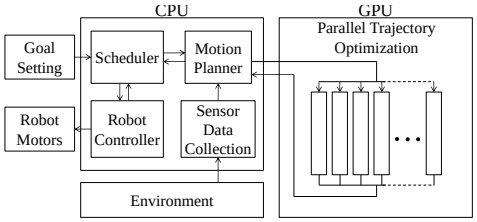


Fig. 2: *The overall architecture of our parallel replanning algorithm. The planner consists of four individual modules (scheduler, motion planner, robot controller, sensor data collection), each of which runs as a separate thread. When the motion planning module receives a planning request from the scheduler, it launches optimization of multiple trajectories in parallel.*

GPU). Our goal is to exploit the computational capabilities of these commodity parallel processors for optimization-based planners and real-time replanning in dynamic scenes. In this section, we present a new parallel algorithm to solve the optimization problem highlighted in in (2).

Our parallel replanning algorithm is based on the stochastic optimization solver introduced by [7] to solve (2). The solver is a derivative-free method which allows us to plan trajectories in dynamic environments where derivatives for the cost of dynamic obstacles are not available. We parallelize our algorithm in two ways. First, we parallelize the optimization of a single trajectory by parallelizing each step of optimization using multiple threads on a GPU (Fig. 4). Second, we parallelize the optimization of multiple trajectories by using different initial seed values. Since it is a randomized algorithm, the solver may converge to different local minima, and the running time of the solver also varies based on the initial seed values. In practice, such parallelization can improve the responsiveness and the quality of the resulting trajectory.

In this section, we describe our parallel replanning algorithm, which exploits multiple cores. First we present the framework of the parallel replanning pipeline with multiple trajectories. We also present the GPU-based algorithm for single trajectory optimization.

A. Parallelized Replanning with Multiple Trajectories

As shown in Fig. 2, our algorithm consists of several modules: scheduler, motion planner, robot controller and sensor data collection. The scheduler sends a planning request to the motion planner when it gets new goal information. The motion planner starts optimizing multiple trajectories in parallel. When the motion planner computes a new trajectory which is safe for the given time interval Δ_t, the scheduler sends the trajectory to the robot controller to execute the trajectory. While the robot controller executes the trajectory, the scheduler requests planning of the next execution interval from the motion planner. The motion planner also gets updated environment descriptions from the sensors and utilizes them to derive bounds on the trajectories of dynamic obstacles during the next time interval. Since all modules run in separate threads, each module does not need to wait on other modules and can work concurrently.

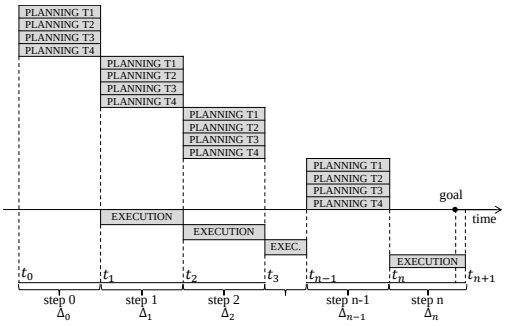


Fig. 3: *The timeline of interleaving planning and execution in parallel replanning. In this figure, we assume the number of trajectories computed by parallel optimization algorithm as four. At time t_0, the planner starts planning for time interval [t_1, t_2], during the time budget [t_0, t_1]. It finds a solution by trying to optimize four trajectories in parallel. At time t_1, the planner is interrupted and returns the result corresponding to the best trajectory to scheduler module. Then the scheduler module executes the trajectory.*

Fig. 3 illustrates interleaved planning and execution with multiple trajectory planning. During step i, the planner has a time budget $\Delta_i = t_{i+1} - t_i$, and it is also the time budget available for execution during step i. During the planning computation in step i, the planner generates trajectories corresponding to the next execution step, i.e, the time interval $[t_{i+1}, t_{i+2}]$. The sensor information at t_i is used to estimate conservative bounds for the dynamic obstacles during the interval $[t_{i+1}, t_{i+2}]$.

Within the time budget, multiple initial trajectories are refined by the optimization algorithm to generate multiple solutions which are sub-optimal and have different costs. Some of the solutions may not be collision-free for the execution interval, which could be due to the limited time budget, or the local optima corresponding to that particular solution. However, the parallelization using multiple trajectories increases the probability that a collision-free trajectory will be found. It also usually yields a higher-quality solution, as we discussed in Section III-B.

B. Highly Parallel Trajectory Optimization

Because we parallelize the computation of multiple trajectories, our approach improves the responsiveness of the planner. We parallelize various aspects of the stochastic solver on the GPUs by using random noise vectors.

The trajectory optimization process and the number of threads used during each step are illustrated in Fig. 4. The algorithm uses $(k \cdot m \cdot n \cdot d)$ threads in parallel according to these steps and exploits the computational power of GPUs.

The algorithm starts with the generation of k initial trajectories. As defined in Section III, each trajectory is generated in the configuration space $\mathcal{C}$(which has dimension d), which has n waypoints from q_{start} to q_{end}. Then the algorithm generates m random noise vectors (with dimension d) for all the n waypoints on the trajectory. These noise vectors are used to perform stochastic update of the trajectory. Adding these

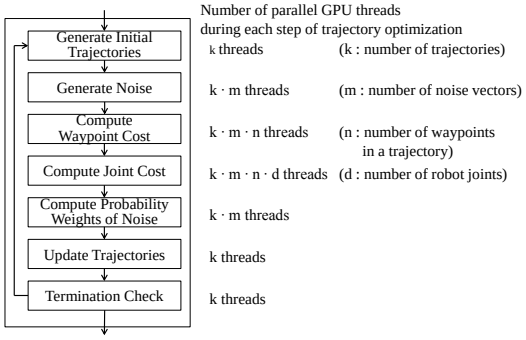


Fig. 4: *The detailed breakdown of GPU trajectory optimization. It starts with the generation of k initial trajectories. From these initial trajectories, the algorithm iterates over stochastic optimization steps. The waypoint costs include collision cost, end effector orientation cost, etc. We also compute joint cost, which might include smoothness costs or the cost of computing the torque constraints. The current trajectory cost is repeatedly improved until the time budget runs out.*

m noise vectors to the current trajectory results in m noise trajectories. The cost for a waypoint, such as costs for static and dynamic obstacles, are computed for each waypoint in the noise trajectories. As described in the Section III-A, the static obstacle cost is computed by precomputed signed EDT. The 3D space positions of the overlapping spheres $b \in \mathcal{B}$ of the robot are computed by the kinematic model of the robot in the configuration of each waypoint. Collision detection for the cost of dynamic obstacles is computed by the GPU collision detection algorithm [29]. Smoothness cost, computed by a matrix multiplication $\|\mathbf{AQ}\|^2$ for each joint, can be computed efficiently using the parallel capabilities of a GPU. When the costs of all noise trajectories are computed, the current trajectory is updated by moving it towards a direction which reduces the cost. The update vector is computed by the weighted sum of noise vectors, which are inversely proportional to their costs. If the given time budget is expired, the optimization of all trajectories are interrupted and the best solution is returned.

V. Analysis

In this section, we analyze the benefits of parallelization on the improvement in responsiveness and the quality of the trajectory computed by the planner.

A. Responsiveness

The use of multiple trajectories improves the responsiveness of our planner. The optimization function corresponding to (2) typically has multiple local minima. In general, any trajectory that is collision-free, satisfies all constraints, and is smooth can be regarded as an acceptable solution. In this section, we show that the optimization of multiple trajectories by our GPU-based algorithm improves the performance of our planner.

The trajectory optimization uses the random number-based algorithm in two stages. First, it generates initial trajectories using randomly generated seeds. Then the algorithm uses stochastic optimization to improve the trajectories. Both of these steps have similar statistical characteristics and their performance is improved by parallelization. In this section, we mainly focus on analyzing initial trajectory generation.

In terms of generating initial trajectories, we assume that the different random seeds used by the algorithm are uniformly distributed. Each trajectory has a different distance to collision-free solutions, and the expected time cost of the trajectory is proportional to the distance. We define the distance from a trajectory $\mathbf{Q}$ to collision-free solutions as:

$$d(\mathbf{Q}) = \max_i \left(\inf\{\|\mathbf{q}_i - \mathbf{p}\| | \mathbf{p} \in \mathcal{C}_{free}\}\right), \tag{5}$$

where $\mathcal{C}_{free}$ represents the collision-free space in the configuration space. Let the mean of the trajectory distances be μ and their variation be σ^2. Note that parameters μ and σ^2 reflect the problem space: large μ implies that the environment is challenging and the solver needs more time to compute an acceptable result; large σ^2 means that the result is sensitive to the choice of initial values.

Suppose the planner optimizes n trajectories and we denote the time costs of different trajectories by X_1, ..., X_n, respectively. Then the time cost for the parallelized solver is $X = \min(X_1, ..., X_n)$, which is called the first order statistic of $\{X_i\}$. We measure the theoretical acceleration due to parallelization by computing the expected time costs without and with parallelization:

Definition *The theoretical acceleration of an optimization-based planner with n trajectories is $\tau = \frac{\mathbb{E}(X_i)}{\mathbb{E}(X)} = \frac{\mu}{\mathbb{E}(X)}$, where $X = \min(X_1, ..., X_n)$.*

If X_i follows the uniform distribution, then the acceleration ratio can be simply represented as $\tau = \frac{n+1}{2}$. For general distributions, we can get the expected time costs for n trajectories from the probability density function of the distribution of X_i. Since all the trajectories are generated for the same configuration space, they share the same probability density function. The probability of the first order statistics falling in the interval $[u + du]$ is

$$\begin{aligned}&\left(1 - \left(\int_{u+du}^{\infty} p_{X_i}(u)du\right)^n\right) - \left(1 - \left(\int_{u}^{\infty} p_{X_i}(u)du\right)^n\right) \\ &= \left(\int_{u}^{\infty} p_{X_i}(u)du\right)^n - \left(\int_{u+du}^{\infty} p_{X_i}(u)du\right)^n\end{aligned} \tag{6}$$

where $p_{X_i}(u)$ is the probability density function of X_i.

With this probability density function for the first order statistics $p_X(u)$, the expected time cost can be evaluated as:

$$\mathbb{E}(X) = \int_0^{\infty} u \cdot p_X(u)du \tag{7}$$

We evaluate the trajectory distance distribution of the configuration space from some experiments (Fig. 5). We measure the Euclidean distances to the nearest collision-free points from the waypoints of the all possible initial

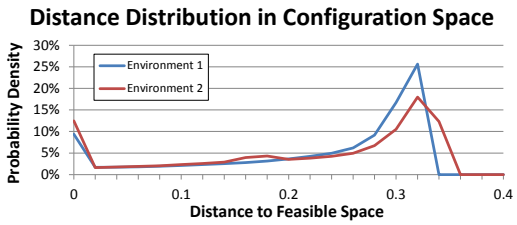


Fig. 5: *The distribution of the distance to the solution in configuration space. The robot has four revolute joints. We discretize the 4-DOF space and measure the distances to the collision-free space from the trajectories generated from all the discretized points. Environment 1 has 12 small obstacles, and the environment 2 has 3 obstacles in the scene.*

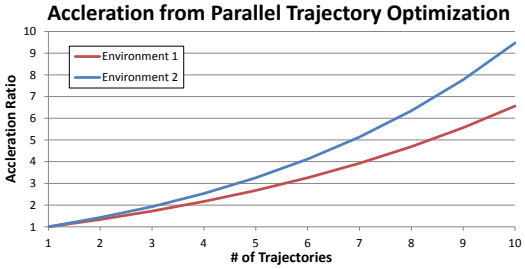


Fig. 6: *Benefits of a parallel, multi-threaded algorithm in terms of the responsiveness improvement. We assume that the time costs of different trajectories for optimization are proportional to the distance to the feasible solution. We show the acceleration by varying the number of trajectories on the two distributions from Fig. 5.*

trajectories in the configuration space, then evaluate the distribution. With this distribution, we evaluate the expected time cost with varying number of trajectories using (7). Fig. 6 shows the acceleration ratio. This graph shows that the higher the number of trajectories, we obtain a higher speedup based on parallelization.. Additionally, the acceleration is larger in the second environment, which has a bigger mean; this indicates that the benefit is greater when the environment is more challenging.

We also analyze the responsiveness of the planner based on GPU parallelization. The computation of each waypoint and each joint are processed in parallel using multiple threads on a GPU, which improves the performance of the optimization algorithm. Fig. 7 shows the performance of the GPU-based parallel optimization algorithm. The environment of the first benchmark in Section VI is used for this measurement. The GPU-based algorithm utilizes various cores to improve the performance of a single-trajectory computation, as shown in Fig. 4. Increasing the number of trajectories causes the system to share the resources for multiple trajectories. Overall, we observe that by simultaneously optimizing multiple trajectories, we obtain a higher throughput using GPUs.

B. Quality

The parallel algorithm also improves the quality of the solution that the planner computes. The optimization problem in Equation 2 has $D \cdot N$ degrees of freedom; N tends to be a large number (often several hundreds). The space has a

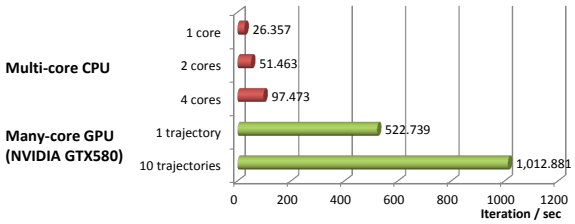


Fig. 7: *Benefits of the parallel algorithm in terms of the performance of the optimization algorithm. The graph shows the number of optimization iterations that can be performed per second. When multiple trajectories are used on a multicore CPU (by varying the number of cores), each core is used to compute one single trajectory. The number of iterations performed per second increases as a linear function of the number of cores. In the case of many-core GPU optimization, increasing the number of trajectories results in sharing of GPU resources among different trajectory computations, and the relationship is non-linear. Overall, we see a better utilization of GPU resources if we optimize a higher number of trajectories in parallel.*

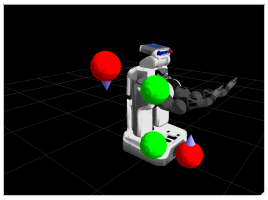
number of global optima, acceptable local optima, and many other local optima which are not acceptable (not collision-free or not smooth). It is difficult to find the global optimal solution when searching in such a high-dimensional space. However, we can show that the use of multiple initializations can increase the probability of computing the the global optima or a solution that is close to the global optima. According to [33], the probability for a pure random search to find the global optima using n uniform samples is defined as Lemma 5.1.

Lemma 5.1: An optimization-based planner with n threads will compute the global optima with the probability $1-(1-\frac{|A|}{|S|})^n$, where S is the entire search space. A is the neighborhood around the local optimal solutions where the local optimization converges to one of the global optima. $|\cdot|$ is the measurement of the search space.

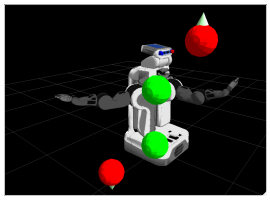
Here $\frac{|A|}{|S|}$ measures the probability that one random sample lies in the neighborhood of the global optima. Although it is hard to measure the exact value of $|A|$ in a high-dimensional space, it can be expected that $|A|$ will be smaller as the envionment becomes more complex and has more local optima. Each initial random value converges at one of the local optima. If it is a global optimum, the planner finds a global optimal solution. Using more trajectories increases the probability that one of the initial values is placed in A. As a result, Lemma 5.1 provides a lower bound on the probability that an optimization-based planner with n threads will compute the global optima. When the number of threads increases, we have a higher chance of computing the global optimal trajectory. In the same manner, the increasing number of threads improves the probability that the planner computes an acceptable solution.

VI. Results

In this section, we highlight the performance of our parallel planning algorithm in dynamic environments. All



(a) *Start configuration used in the performance measurement*



(b) *Goal configuration used in the performance measurement*

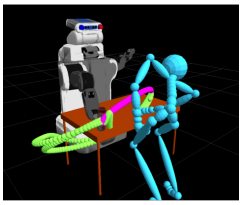
Fig. 8: *Planning environment used to evaluate the performance of our planner. The planner computes a trajectory of robot arm which avoids dynamic obstacles and moves horizontally from right to left. Green spheres are static, and red spheres are dynamic obstacles. Figure (a), (b) Show the start and goal configurations of the right arm of the robot.*

Scenario	Average planning time (ms)	Std. dev. planning time(ms)
CPU 1 core	810	0.339
CPU 2 core	663	0.284
CPU 4 core	622	0.180
GPU 1 trajectory	337	0.204
GPU 4 trajectory	203	0.326
GPU 10 trajectory	60	0.071

TABLE I: *Results obtained from our trajectory computation algorithm based on different levels of parallelization and number of trajectories (for the benchmarks shown in Fig. 8). The planning time decreases when the planner uses more trajectories.*

experiments are performed on a PC equipped with an Intel i7-2600 8-core CPU 3.4GHz with 8GB of memory. Our experiments are based on the accuracy of the PR2 robot's LIDAR sensor (i.e. 30mm), and the planning routines obtain information about dynamic obstacles (positions and velocities) every 200 ms. Our GPU algorithm is implemented on an NVIDIA Geforce GTX580 graphics card, which supports 512 CUDA cores.

Our first experiment is designed to estimate the responsiveness of the planner. We plan a trajectory of the 7 degree-of-freedom right arm of PR2 in a simulation environment. We measure the time needed to compute a collision-free solution by varying the number of trajectories using both CPU- and GPU-based planners. We perform this experiment to compute the appropriate time interval for a single planning time step during replanning; a shorter planning time means the planner is more responsive. We repeat the test 10 times for each scenario, and compute the average and standard deviation of the overall planning time. This result is shown in Table I. We observe that the GPU-based planner demonstrates better performance than a CPU-based planner. In both cases, it is shown that the performance of the planner increases as more trajectories are optimized in parallel. We restrict the maximum number of iterations to 500. The planner failed to compute the collision-free solution only once in our benchmarks, for a single-trajectory case on a GPU. This happens because the single-trajectory instance gets stuck in a local minimum and is unable to compute an acceptable solution.



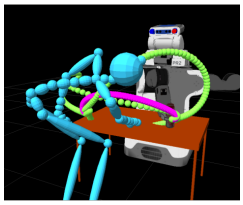


Fig. 9: *Parallel replanning in dynamic environments with a human obstacle. The planner optimizes multiple paths which are smooth and avoid collision with the obstacle. Each colored path corresponds to a different search in the configuration space. The optimal path for each case is shown in purple.*

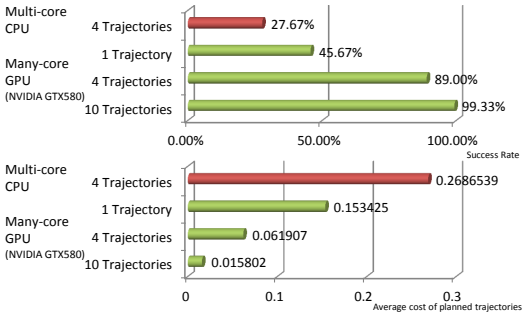


Fig. 10: *Success rate and trajectory cost results obtained from the replanning in dynamic environments on a multi-core CPU and a many-core GPU. The success rate and trajectory cost is measured for each planner. The use of multiple trajectories in our replanning algorithm results in higher success rates and trajectories with lower costs and thereby, improved quality.*

In the next experiment, we test our parallel replanning algorithm in dynamic environments with human-like obstacles (Fig. 9); these human-like obstacles follow the paths computed by motion-captured data, which is not known to the robot or the planner. The planner uses the replanning technique to reach the goal while avoiding collisions with the obstacles. During each step, the planner uses conservative local bounds that are based on positions and velocities of the obstacles. For this experiment, the CPU-based planner is too slow to handle the dynamic human motion used in this environment; As a result, we reduced the moving speed of the human obstacle by 3X, so that the CPU-based planner could handle it. We measure the success rate of the planner and the trajectory cost corresponding to the collision-free trajectory to the goal position. The total cost function used in the optimization algorithm is the sum of the obstacle cost and the smoothness cost. However the solution trajectories have only smoothness cost since they have no collisions. We measure the cost by varying the number of optimized trajectories in order to measure the effect of parallelization. We run 300 trials on the planning problem shown in Fig. 9; Fig. 10 highlights the performance. As the number of optimized trajectories increases, the success rate increases and the cost of the solution trajectory decreases. This result validates that the multiple trajectory optimization improves the quality of

the solution, as shown in Section V-B.

VII. Limitations, Conclusions, and Future Work

We present a novel parallel algorithm for real-time replanning in dynamic environments. The underlying planner uses an optimization-based formulation, and we parallelize the computation on many-core GPUs. Moreover, we derive bounds on how parallelization improves the responsiveness and the quality of the trajectory computed by our planner.

At the moment, our planner doesn't take into account uncertainty in sensor data; The conservative bound (4) is only good for local intervals. If there is a very strong or abrupt motion in any obstacle motion, this bound may not hold. We need to evaluate the performance in more complex environments with multiple obstacles.

There are many avenues for future work. Our current formulation does not take into account any uncertainty in sensor data. We would like to integrate our approach with a physical robot, model different constraints on the motion, and evaluate its performance in real-world scenarios. Furthermore, we would like to investigate other parallel optimization techniques to further improve the performance. Recently, we have extended our algorithm to high DOF robots [34].

VIII. Acknowledgments

This research is supported in part by ARO Contract W911NF-10-1-0506, NSF awards 0917040, 0904990, 1000579 and 1117127, and Willow Garage.

References

[1] F. Belkhouche, "Reactive path planning in a dynamic environment," *Robotics, IEEE Transactions on*, vol. 25, no. 4, pp. 902 –911, aug. 2009.

[2] K. Hauser, "On responsiveness, safety, and completeness in real-time motion planning," *Autonomous Robots*, vol. 32, no. 1, pp. 35–48, 2012.

[3] T. Kunz, U. Reiser, M. Stilman, and A. Verl, "Real-time path planning for a robot arm in changing environments," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, oct. 2010, pp. 5906 –5911.

[4] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," *International Journal of Robotics Research*, vol. 21, no. 3, pp. 233–255, March 2002.

[5] S. Koenig, C. Tovey, and Y. Smirnov, "Performance bounds for planning in unknown terrain," *Artificial Intelligence*, vol. 147, no. 1-2, pp. 253–279, July 2003.

[6] S. Petti and T. Fraichard, "Safe motion planning in dynamic environments," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005, pp. 2210–2215.

[7] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, "STOMP: Stochastic trajectory optimization for motion planning," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2011, pp. 4569–4574.

[8] N. Ratliff, M. Zucker, J. A. D. Bagnell, and S. Srinivasa, "CHOMP: Gradient optimization techniques for efficient motion planning," in *Proceedings of International Conference on Robotics and Automation*, 2009, pp. 489–494.

[9] C. Park, J. Pan, and D. Manocha, "ITOMP: Incremental trajectory optimization for real-time replanning in dynamic environments," in *Proceedings of the International Conference on Automated Planning and Scheduling, to appear*, 2012.

[10] M. Likhachev and D. Ferguson, "Planning long dynamically feasible maneuvers for autonomous vehicles," *International Journal of Robotics Research*, vol. 28, no. 8, pp. 933–945, August 2009.

[11] P. Fiorini and Z. Shiller, "Motion planning in dynamic environments using velocity obstacles," *International Journal of Robotics Research*, vol. 17, no. 7, pp. 760–772, 1998.

[12] D. Wilkie, J. P. van den Berg, and D. Manocha, "Generalized velocity obstacles," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009, pp. 5573–5578.

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

[14] ——, "Planning in domains with cost function dependent actions," in *Proceedings of AAAI Conference on Artificial Intelligence*, 2011.

[15] J. van den Berg and M. Overmars, "Roadmap-based motion planning in dynamic environments," *IEEE Transactions on Robotics*, vol. 21, no. 5, pp. 885–897, oct. 2005.

[16] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime dynamic A*: An anytime, replanning algorithm," in *Proceedings of the International Conference on Automated Planning and Scheduling*, 2005.

[17] P. Chen and Y. Hwang, "Sandros: a dynamic graph search algorithm for motion planning," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 3, pp. 390–403, jun 1998.

[18] O. Brock and O. Khatib, "Elastic strips: A framework for motion generation in human environments," *International Journal of Robotics Research*, vol. 21, no. 12, pp. 1031–1052, 2002.

[19] S. Quinlan and O. Khatib, "Elastic bands: connecting path planning and control," in *Proceedings of IEEE International Conference on Robotics and Automation*, 1993, pp. 802–807 vol.2.

[20] A. Dragan, N. Ratliff, and S. Srinivasa, "Manipulation planning with goal sets using constrained trajectory optimization," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2011, pp. 4582–4588.

[21] S.-H. Lee, J. Kim, F. Park, M. Kim, and J. Bobrow, "Newton-type algorithms for dynamics-based robot movement optimization," *Robotics, IEEE Transactions on*, vol. 21, no. 4, pp. 657–667, Aug. 2005.

[22] S. Lengagne, P. Mathieu, A. Kheddar, and E. Yoshida, "Generation of dynamic motions under continuous constraints: Efficient computation using b-splines and taylor polynomials," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010, pp. 698–703.

[23] T. Lozano-Perez and P. O'Donnell, "Parallel robot motion planning," in *International Conference on Robotics and Automation*, 1991, pp. 1000–1007.

[24] N. Amato and L. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *Proceedings of IEEE International Conference on Robotics and Automation*, 1999, pp. 688–694 vol.1.

[25] D. Devaurs, T. Simeon, and J. Cortes, "Parallelizing RRT on distributed-memory architectures," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2011, pp. 2261 –2266.

[26] E. Plaku and L. Kavraki, "Distributed sampling-based roadmap of trees for large-scale motion planning," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2005, pp. 3868–3873.

[27] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha, "Interactive motion planning using hardware accelerated computation of generalized voronoi diagrams," in *International Conference on Robotics and Automation*, 2000, pp. 2931–2937.

[28] C. Pisula, K. Hoff, M. C. Lin, and D. Manocha, "Randomized path planning for a rigid body based on hardware accelerated voronoi sampling," in *International Workshop on Algorithmic Foundation of Robotics*, 2000, pp. 279–292.

[29] J. Pan, C. Lauterbach, and D. Manocha, "g-Planner: Real-time motion planning and global navigation using gpus," in *Proceedings of AAAI Conference on Artificial Intelligence*, 2010.

[30] J. Bialkowski, S. Karaman, and E. Frazzoli, "Massively parallelizing the RRT and the RRT*," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011, pp. 3513–3518.

[31] J. Kider, M. Henderson, M. Likhachev, and A. Safonova, "High-dimensional planning on the gpu," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2010, pp. 2515–2522.

[32] S. Gottschalk, M. C. Lin, and D. Manocha, "Obbtree: a hierarchical structure for rapid interference detection," in *SIGGRAPH*, 1996, pp. 171–180.

[33] A. Rinnooy Kan and G. Timmer, "Stochastic global optimization methods part i: Clustering methods," *Mathematical Programming*, vol. 39, pp. 27–56, 1987.

[34] C. Park, J. Pan, and D. Manocha, "Hierarchical optimization-based planning for high-DOF robots," University of North Carolina at Chapel Hill, Tech. Rep., 2012.