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 makes no assumption 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 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.

1 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 wheel chairs, 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 [2, 12, 18]. However, the resulting algorithms may not generate smooth paths or handle dynamic constraints in real time.Other technique for dynamic environments are 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 real-time motion planning in dynamic scenes. In order to deal with unpredictable environments, we use replanning algorithms that interleave planning with execution [14, 12, 17, 26]. In these cases, the robot may only compute partial or sub-optimal plans in the given time interval. In order to produce smooth paths and handle dynamic constraints, we combine replanning techniques with optimization-based planning [15, 32, 25].

We present a novel parallel optimization-based motion planning algorithm for dynamic scenes. Our planning algorithm optimize multiple trajectories in parallel to explore a broader subset of the configuration space and compute 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 pro-

Chonhyon Park and Jia Pan and Dinesh Manocha
the Department of Computer Science, the University of North Carolina, Chapel Hill, e-mail: {chpark,panj,dm}@cs.unc.edu

cessing 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 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 performance and responsiveness in Section 5. We highlight its performance in simulated dynamic environments in Section 6.

2 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.

2.1 Motion Planning in Dynamic Environments

Many approaches for motion planning in dynamic environment 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 [21]. Other techniques compute an appropriate robot velocity that can avoid a collision with moving obstacles during a short time step [10, 34]. The state space for planning in a dynamic environment is given as $\mathscr{C} \times T$, i.e., the Cartesian product of configuration space and time. Some RRT variants can handle continuous state space directly [26], while other methods discretize the state space and use classic heuristic search [28, 27] or roadmap based algorithms [3].

Some planning algorithms for dynamic environments [3, 28] assume that the inertial constraints, such as acceleration and torque limit, are not significant for the robot. Such assumption 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 many scenarios, the planning computation can be expensive. As a result, path planning before execution strategy can lead to long delays during 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 [14, 12, 26] or search-based methods [17, 22]. Most replanning algorithms use fixed time steps when interleaving between planning and execution [26]. Some recent work [12] computes the interleaving timing step in an adaptive manner to balance between safety, responsiveness, and completeness of the overall system.

2.2 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 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 using optimization techniques. One simple technique is the so-called 'shortcut'

heuristic, which picks pairs of configurations along a collision-free path and invokes a local planner to replace the intervening sub-path with a shorter one [7]. Other examples include elastic bands or elastic strips planning, which models paths as mass-spring systems and uses gradient-based methods to compute a minimum-energy path [6, 31]. Other optimization-based planning algorithms can relax the assumptions about the initial path and may start with an in-collision path. Some recent approaches, such as [32, 15, 9], 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, and compute the final trajectory. 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 [19, 20] and can be more expensive.

2.3 Parallel Algorithms

Due to the rapid advances in multi-core and many-core 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 [1, 8, 30].

The rasterization capabilities of a GPU can be used for real-time motion planning of low DOF robots [13] or to improve the sample generation in narrow passages [29]. Recently, the GPUs have been exploited to accelerate sampling-based motion planners in high dimensional spaces, including sample-based planning [24], RRT algorithms [4], and search-based planning [16].

3 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 first describe the underlying framework for optimization-based planning and give an overview of our planning and execution framework.

3.1 Optimization-based Planning

Traditionally, the goal of motion planning is to find a collision free trajectory from the start configuration to 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 $\mathscr{C}$ of a robot. In this case, the dimension $\mathscr{D}$ of $\mathscr{C}$ 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 $\mathscr{T}$, 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 $\mathbf{Q} \in \mathscr{R}^{D \cdot N}$:

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

Similar to previous work [32, 15, 25], we define the objective function of our optimization problem as:

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

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 Kalakrishnan et al [15]. The solution to the optimization problem in Equation (2) corresponds to the optimal trajectory of the robot.

In order to compute static and dynamic obstacle costs, we use signed Euclidean Distance Transform (EDT) and geometric collision detection as in previous work [25]. 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 $\mathscr{B}$ by using a set of overlapping spheres $b \in \mathscr{B}$. 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:

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

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 only requires a simple table lookup after one-time 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 formalize the cost of dynamic obstacles. Object-space collision detection algorithms based on bounding volume hierarchies [11] are used to compute the dynamic obstacle cost efficiently.

In real-world applications, we cannot make assumptions about the future 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, we compute a conservative local bound on the trajectories of dynamic obstacles and use them for the collision detection. The allowed sensing error and the obstacle velocity are used 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 follows:

$$\overline{O}^d([t_0,t_1]) = \bigcup_{t\in[t_0,t_1]} (1 + \frac{\text{sensing error}}{\dot{O}^d(t)})\overline{O}^d(t) \tag{4}$$

3.2 Planning and Execution Framework

In order to improve the responsiveness of the robot in dynamic environments, we use a replanning approach previously used for sampling-based motion planning [12]. 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 latest sensor information. During each planning step, we compute an estimate of the position and velocity of dynamic obstacles based on senor data. Next, a conservative bound of 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 safeness of the trajectory during the planning interval, however

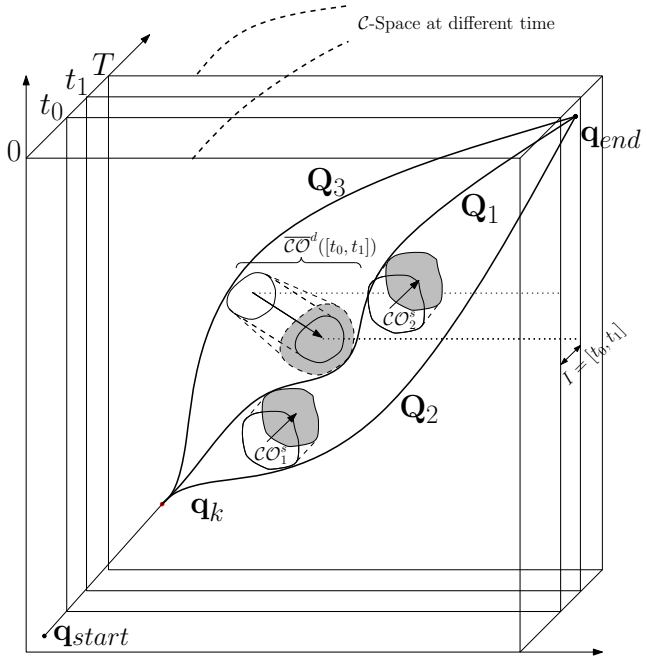


Fig. 1: Multiple trajectories that arise in the optimization-based motion planning. The coordinate system shows how the configuration space changes over time as the dynamic obstacles move over time: each plane slice represents the configuration space at time t. In the environment, there are three C-obstacles: 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. Though all the trajectories have no collision with obstacles during the time interval I, trajectory Q_2 represents the most optimal solution based on smoothness and collision-free cost functions. Our goal is to use commodity parallel hardware to compute these multiple trajectories in parallel and improve the probability of computing an optimal solution.

the size of the bound increases as the planning interval increases. It turns out that large conservative bounds make it hard for planner to compute a solution or result in a less optimal solution in the given time bound. 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 Figure 1.

4 Parallel Replanning

Nowadays, all commodity processors have multiple cores. Even some of the robot systems are equipped with multi-core CPU processors. Furthermore, these robot systems provide expansibility in terms of using many-core accelerators, such as graphics processing units (GPUs). These many-

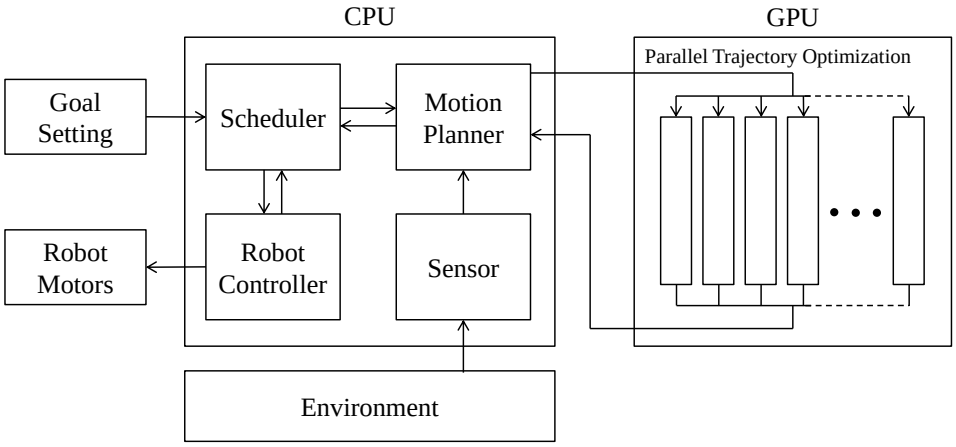


Fig. 2: The overall architecture of our parallel replanning algorithm. The planner consists of four individual modules(scheduler, motion planner, robot controller, sensor), and each of them runs as a separate thread. When the motion planning module receives a planning request from the scheduler, it launches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

core accelerators are massively parallel processors, which offer a very high peak performance (e.g. up to 1 TFlop single precision capaibilities). Our goal is to exploit the computational capabilities of these commodity parallel processors for optimization-based planners and real-time replaning. In this section, we present a new parallel algorithm to solve the optimization problem highlighted in in Equation (2).

Our parallel replanning algorithm is based on stochastic optimization solver introduced by [15] to solve Equation (2). The solver is a derivative-free method which allows us to optimize arbitrary cost functions for which derivatives are not available, or are non-differentiable or non-smooth. We parallelize our algorithm in two ways. First, we parallelize optimization of a single trajectory by parallelizing each step of optimization by using multiple threads on a GPU. Second, we parallelize 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 trajectory computed by our planner.

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 also present the GPU-based algorithm for single trajectory optimization.

4.1 Parallelized Replanning with Multiple Trajectories

As shown in Figure 2, our algorithm consists of several modules: scheduler, motion planner, robot controller and sensor. The scheduler sends a planning request to the motion planner when it gets new goal information. The motion planner starts optimizing multiple trajectory in parallel. When the motion planner computes a new trajectory which is safe for the given time interval Δ_t, the scheduler notifies 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 to the motion planner. The motion planner also gets updated environment description from the sensors and utilizes

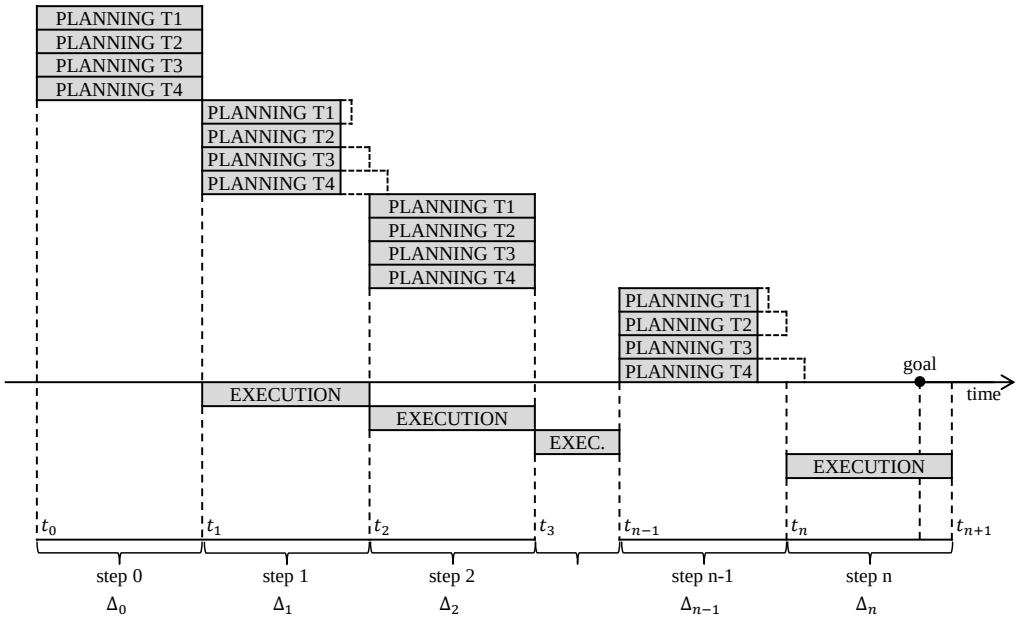


Fig. 3: The timeline of interleaving planning and execution in parallel replanning. In this figure, we assume the number of trajectories computed by parallel optimization algorithm as four. At time t_0, the planner starts planning for time interval $[t_1, t_2]$, during the time budget $[t_0, t_1]$. It 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 scheduler module executes the trajectory. The optimization of multiple trajectories makes it possible to explore a large subset of configuration space and increases the probability of computing the global optimum. While the robot controller executes the planned robot motion, the planner starts planning for the next time step $[t_1, t_2]$ and the result is used at time t_2. This process is repeated until the goal is reached. During the replanning, if the planner determines that one of the computed trajectories is the optimal solution, it terminates the rest of the trajectory computations and returns the optimal trajectory.

it 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.

Figure 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 tries to generates trajectories of 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. It could be due to the limited time budget, or the local optima corresponding to that particular solution. However, parallelization using multiple trajectories increases the possibility that there exists a collision-free trajectory and we can also expect to compute a higher quality solution, as we discussed in Section 3.2.

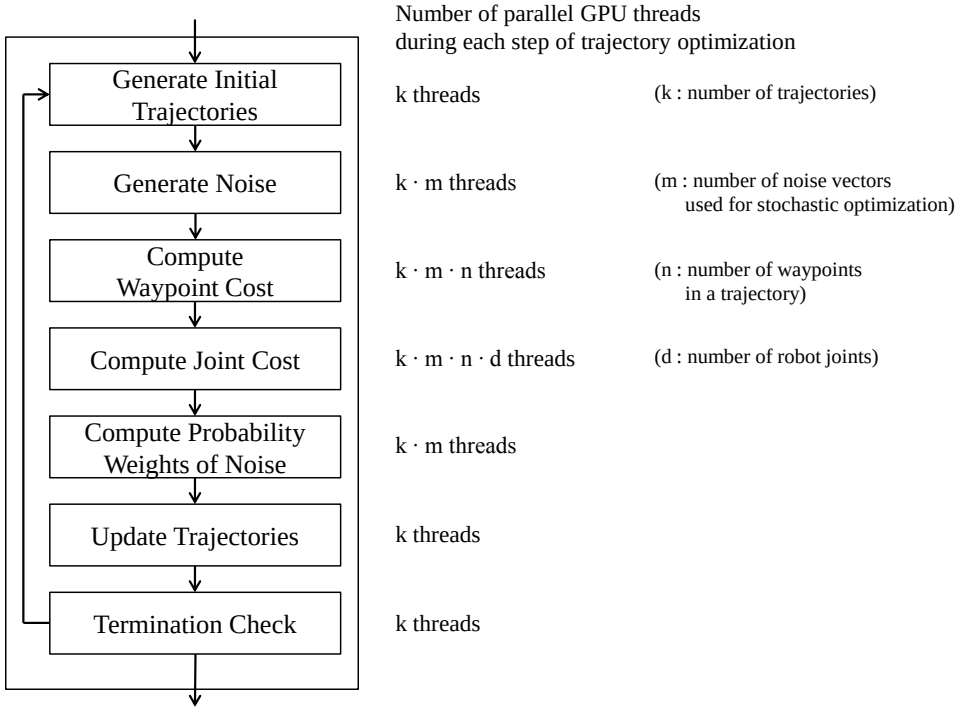


Fig. 4: The detailed breakdown of GPU trajectory optimization. It starts with the generation of k initial trajectories. From these initial trajectories, the algorithm iterates over stochastic optimization steps. First it generates random noise vectors which are used for stochastic optimization, then computes the cost for each waypoint on each noise trajectory. These waypoint costs include collision cost, end effector orientation cost, etc. We also compute joint cost, including smoothness cost or computing the torque constraints. The current trajectory cost is repeatedly improved using the cost of noise trajectories until the algorithm satisfies termination criteria(time budget runs out, or the current trajectory results in an optimal solution).

4.2 Highly Parallel Trajectory Optimization

With the parallelization of computation of multiple trajectories, our algorithm improves the responsiveness of the planner by parallelization of the optimization of a single trajectory. 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 Figure 4. We assume that the number of parallelized trajectories is k, the number of noise vectors used for stochastic optimization is m, a trajectory is discretized into n waypoints, and the number of the free joints of the robot, i.e., the dimension of the configuration space is d. Then the algorithm uses $(k, k \cdot m \cdot n \cdot d)$ 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 3, each trajectory is generated in the configuration space $\mathscr{C}$(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 m noise vectors to the current trajectory results in m noise trajectories, which are slightly modified from the original trajectory. The cost for an waypoint, such as costs for static

and dynamic obstacles are computed for each waypoint in the noise trajectories. As described in the Section 3.1, the static obstacle cost is computed by precomputed signed EDT. EDT is initialized once during the startup step of the planner and does not change over the course of the planner, the 3D space positions of the overlapping spheres $b \in \mathscr{B}$ 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 GPU collision detection algorithm [24]. Smoothness cost, computed by a matrix multiplication $\|\mathbf{AQ}\|^2$ for each joint, can be computed efficiently using parallel capabilities of a GPU. When costs of all noise trajectories are computed, the update of current trajectory is implemented by moving it towards the noise trajectory with low cost. It is computed by the weighted sum of noises which are inversely proportional to their costs. At the end of each iteration, the algorithm decides to stop the optimization or repeat the next iteration. If a trajectory is determined to the global optimal, optimization of all trajectories are interrupted and the optimal solution is returned. Or if the given time budget is expired, similarly, optimization of all trajectories are interrupted and the best solution is returned.

5 Analysis

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

5.1 Responsiveness

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

We assume that the different random seeds used by the algorithm are uniformly distributed, the time costs required by the solver to compute a solution are independent and identically distributed (i.i.d.) random variables, whose mean is μ and variation is σ^2. Note that parameters μ and σ^2 reflect the capability of the solver: 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 solver is sensitive to the initial values. Suppose there are n cores and we denote the time costs of different cores 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 1. The theoretical acceleration of optimization-based planner with n threads 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 $\tau = \frac{n+1}{2}$. A better way to describe the time cost of the solver is obtained based on a normal distribution. If X_i follows normal distribution, according to [5], then:

$$\mathbb{E}(X) \approx \mu + \Phi^{-1}(\frac{1-\alpha}{n-2\alpha+1})\sigma, \tag{5}$$

where $\alpha = 0.375$ and $\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt$. We show the acceleration ratio for different number of threads and different values of $\frac{\sigma}{\mu}$ in Figure 5. It is obvious that when the number of parallel

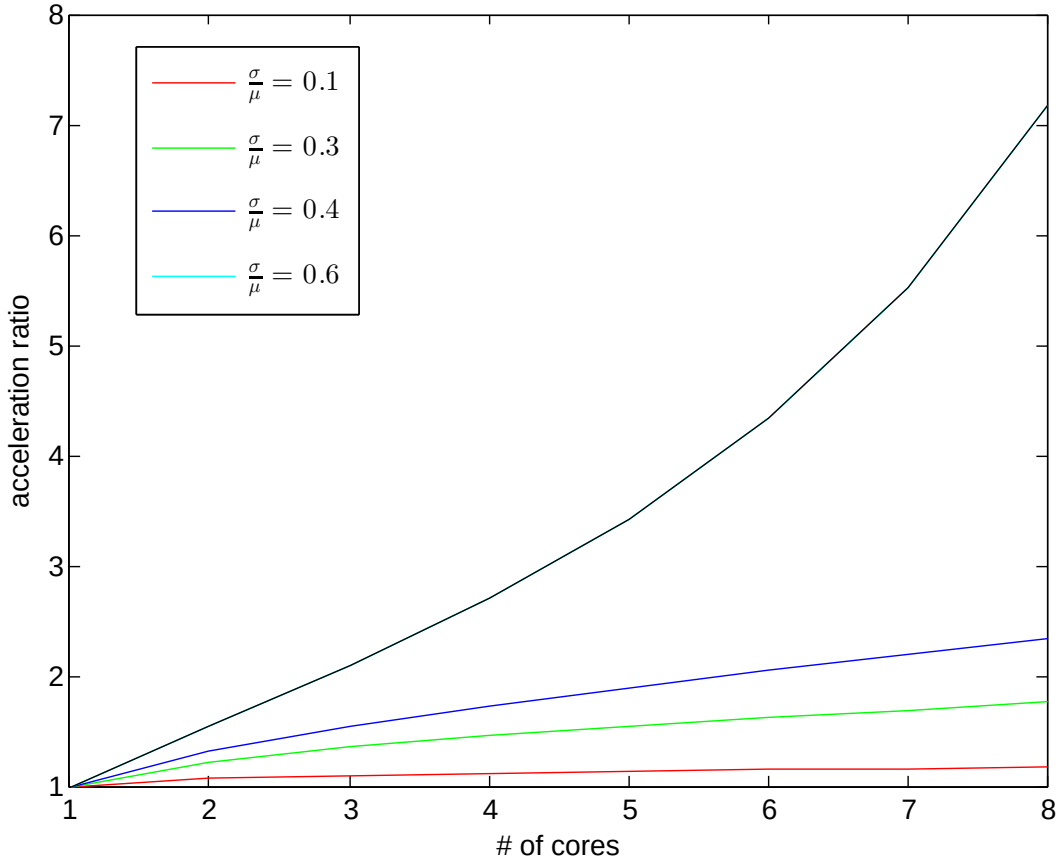


Fig. 5: Benefits of a parallel, multi-threaded algorithm in terms of the responsiveness improvement. We assume that the time costs of different threads using random initializations are i.i.d. Gaussian variables with mean μ and variation σ^2, which provide a measure of the complexity of the optimization problem. We show the theoretical acceleration under such assumption by varying the number of parallel threads and the ratio between σ and μ.

threads increases, the acceleration ratio increases rapidly. Moreover, the acceleration is large when $\frac{\sigma}{\mu}$ is large, i.e., when the solver is sensitive to the initial seed value.

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 and it improves the performance of optimization algorithm. Figure 6 shows the performance of GPU-based parallel optimization algorithm. When we perform multiple trajectory optimization using CPUs, the algorithm assigns one CPU core to compute each trajectory. It implies that the increased number of trajectories only uses more cores and does not affect the performance of a single trajectory optimization computation. However, the GPU-based algorithm also utilizes various cores to improve the performance of a single trajectory computation, as shown in Figure 4. If we increase the number of trajectories, then it 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. We observe that multiple trajectory optimization not only improves the overall performance of the optimization algorithm, but also improves the quality of the trajectory.

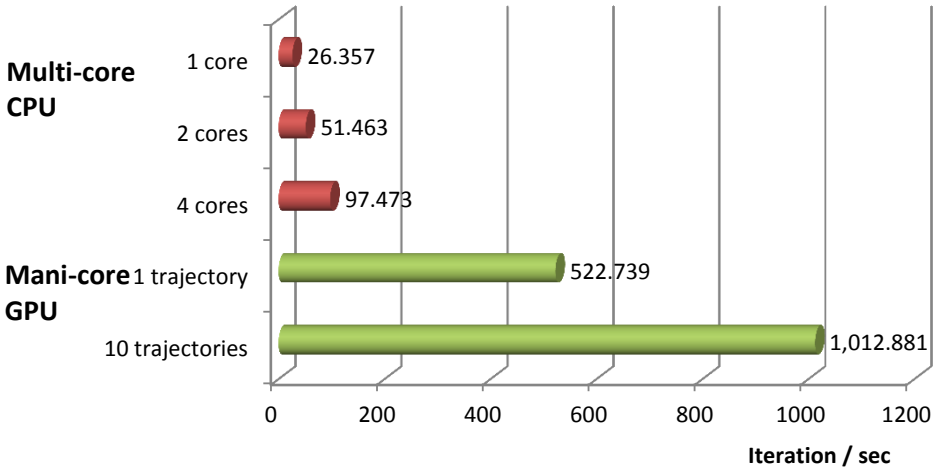


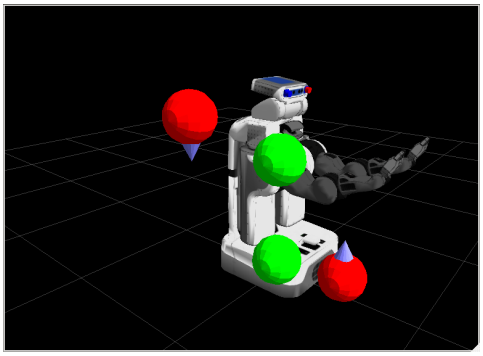
Fig. 6: Benefits of the parallel algorithm in terms of the performance of optimization algorithm. The graph shows the number of optimization iterations that can be performed per second. When multiple trajectories are used on a multicore CPU (by varying the number of cores) optimization, each core is used to compute one single trajectory. Each trajectory computation is performed independently. The number of iterations performed per second increases as a linear function of the number of cores. In case of many-core GPU optimization, increasing the number of trajectories results in sharing of GPU resources among different trajectory computations and we don't observe a linear relationship. Overall, we see a better utilization of GPU resources, if we optimize a higher number of trajectories in parallel. In this case, we observe two times improvement in terms of GPU resource utilization when we try to optimize 10 trajectories, as opposed to a single trajectory.

5.2 Quality

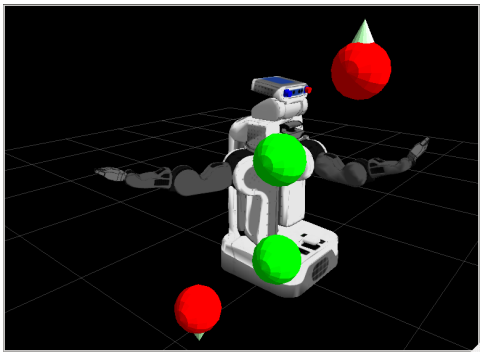
The parallel algorithm can also improve the probability of the planner in terms of computing the global optimal solution. The optimization problem in Equation 2 has $D \cdot N$ degrees of freedom, where N tends to be a large number corresponding to several hundreds. It is difficult to find a global optimal 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 global optima. According to [33], the probability for a pure random search to find a global optima using n uniform samples is defined as Theorem 1.

Lemma 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 global optimal solutions and* $|\cdot|$ *is the measurement on the search space.*

Here $\frac{|A|}{|S|}$ measures the probability that one random sample lies in the neighborhood of global optimal solutions. The stochastic solver in our implementation uses stochastic gradient method instead of random search. However, the optimization solver with a specific initial value converges to the global optimum solution if and only if any of the intermediate trajectories generated during the optimization algorithm reaches the global minima. As a result, Lemma 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.



(a) Start configuration used in the performance measurement



(b) Goal configuration used in the performance measurement

Fig. 7: 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	1 Fail
GPU 4 trajectory	203	0.326	
GPU 10 trajectory	60	0.071	

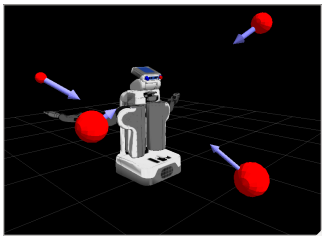
Table 1: Results obtained from our trajectory computation algorithm based on different levels of parallelization and number of trajectories (for the benchmarks shown in Figure 7). The planning time decreases when the planner uses more trajectories.

6 Results

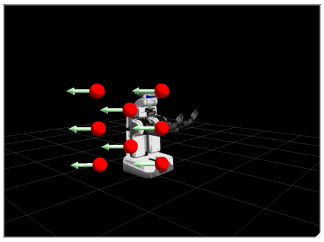
In this section, we highlight the performance of our parallel planning algorithm in dynamic environments. All experiments are performed on a PC equipped with an Intel i7-2600 8-core CPU 3.4GHz with 8GB of memory. Our GPU algorithm is implemented on an NVIDIA Geforce GTX580 graphics card, which supports 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. In the environment shown in Figure 7, there are two static (green) and two moving (red) obstacles. We measure the elapsed time to compute a collision-free solution with varying number of trajectories for both CPU and GPU-based planners. This experiment is performed to compute the appropriate time interval for a single planning time step during replanning. A shorter planning time makes the planner 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 1. We observe that the GPU-based planner demonstrates better performance than a CPU-based planner. In both cases, it is shown that when more trajectories are optimized in parallel, the performance of the planner increases. We restrict the maximum number of iterations to 500, and the planner failed to compute the collision-free solution once for a single trajectory case on GPUs. This happens because the single trajectory instance gets stuck in a local minima and is unable to compute an acceptable solution.

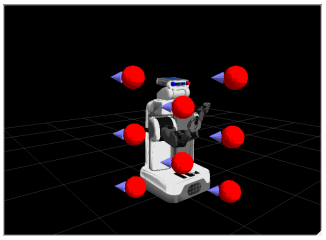
In the next experiment, we test our parallel replanning algorithm in dynamic environments with a high number of moving obstacles (Figure 8). The obstacles in the environments change their velocities periodically. However, this information is not known to the planner. The planner uses replanning



(a) Planning environment with moving obstacles (corresponding to red spheres), which have non-constant velocities (shown with blue arrows).



(b) Planning environment with fast moving obstacles



(c) Planning environment with slow moving obstacles

Fig. 8: Planning environments used in replanning experiments. The planner uses the latest obstacle position and velocity to estimate the local trajectory. (a) There are several obstacles that change their velocities during the planning phase. (b)(c) The obstacles in the environment have varying (high or low) speeds, shown with arrows.

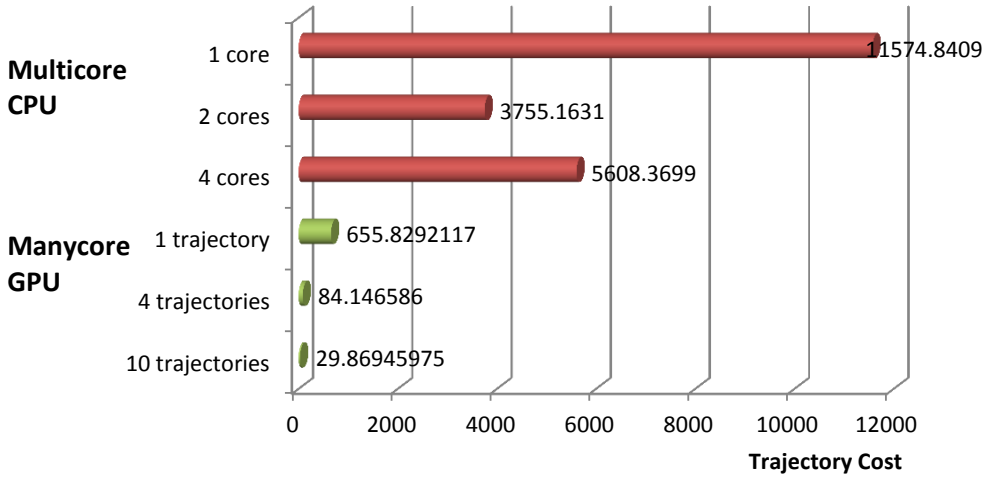
technique to reach the goal while avoiding collisions with the obstacles. During each step, the planner uses conservative local bounds that are based on the positions and velocities of the obstacles. We observe a different level of responsiveness between CPU and GPU-based planners. When the obstacles move at a high speed, the CPU-based planner may not be responsive. Moreover, we measure the cost of computing the entire solution trajectory, including robot execution. The cost used in this experiment consists of two costs, obstacle cost and smoothness cost. We measure the cost with varying number of optimized trajectories in order to measure the effect of parallelization. We run 10 trials on the planning problem shown in Figure 8. Figure 9(a) highlights the performance. As the number of optimized trajectories increases, the overall cost of entire trajectory computation decreases. This result validates that the multiple trajectory optimization improves the quality of solution, as shown in Section 5.2.

We also measure the trajectory cost with moving obstacles with two sets of varying speeds. We use the same environment and only change the speed of obstacles. The result is shown in Figure 9(b). We observe that higher speed obstacles result in trajectories with lower costs associated with them. Moreover, in both cases, multiple trajectory optimization improves the quality of the solution.

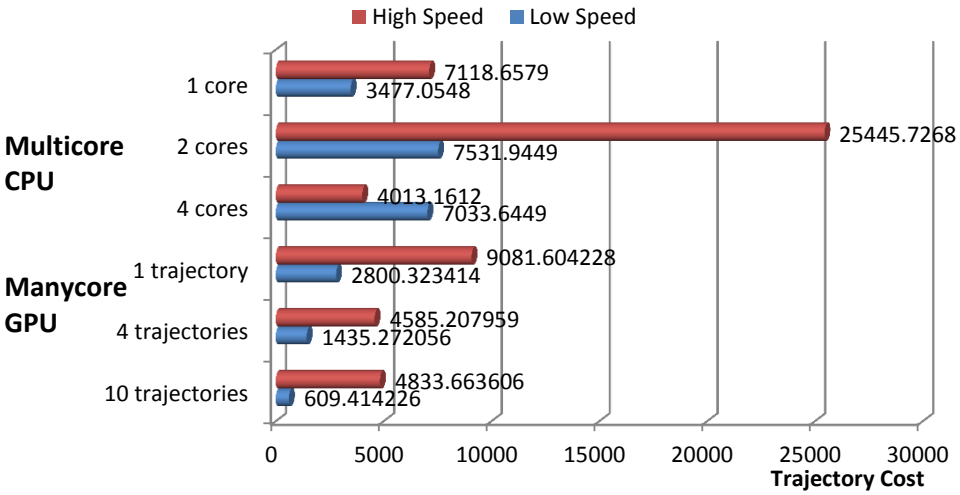
7 Conclusions and Future Work

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.

There are many avenues for future work. Our current formulation doesn't take into account any uncertainty or inaccuracies 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 to investigate other parallel optimization techniques to further improve the performance.



(a) Trajectory costs from replanning in dynamic environments



(b) Trajectory costs during the replanning algorithm based on obstacles moving high and low speeds

Fig. 9: Results obtained from the replanning in dynamic environments on a multi-core CPU and a many-core GPU. (a) Trajectory cost is measured for each planner. (b) Trajectory cost is measured for each planner with different obstacle speeds. The use of multiple trajectories in our replanning algorithm results in trajectories with lower costs and thereby, improved quality.

References

1. Amato, N., Dale, L.: Probabilistic roadmap methods are embarrassingly parallel. In: Proceedings of IEEE International Conference on Robotics and Automation, pp. 688–694 vol.1 (1999)
2. Belkhouche, F.: Reactive path planning in a dynamic environment. Robotics, IEEE Transactions on **25**(4), 902 –911 (2009)
3. van den Berg, J., Overmars, M.: Roadmap-based motion planning in dynamic environments. IEEE Transactions on Robotics **21**(5), 885–897 (2005)
4. Bialkowski, J., Karaman, S., Frazzoli, E.: Massively parallelizing the RRT and the RRT*. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3513–3518 (2011)
5. Blom, G.: Statistical Estimates and Transformed Beta Variables. John Wiley and Sons, Inc (1958)

6. Brock, O., Khatib, O.: Elastic strips: A framework for motion generation in human environments. International Journal of Robotics Research **21**(12), 1031–1052 (2002)
7. Chen, P., Hwang, Y.: Sandros: a dynamic graph search algorithm for motion planning. IEEE Transactions on Robotics and Automation **14**(3), 390–403 (1998)
8. Devaurs, D., Simeon, T., Cortes, J.: Parallelizing RRT on distributed-memory architectures. In: Proceedings of IEEE International Conference on Robotics and Automation, pp. 2261 –2266 (2011)
9. Dragan, A., Ratliff, N., Srinivasa, S.: Manipulation planning with goal sets using constrained trajectory optimization. In: Proceedings of IEEE International Conference on Robotics and Automation, pp. 4582–4588 (2011)
10. Fiorini, P., Shiller, Z.: Motion planning in dynamic environments using velocity obstacles. International Journal of Robotics Research **17**(7), 760–772 (1998)
11. Gottschalk, S., Lin, M.C., Manocha, D.: Obbtree: a hierarchical structure for rapid interference detection. In: SIGGRAPH, pp. 171–180 (1996)
12. Hauser, K.: On responsiveness, safety, and completeness in real-time motion planning. Autonomous Robots **32**(1), 35–48 (2012)
13. Hoff, K., Culver, T., Keyser, J., Lin, M., Manocha, D.: Interactive motion planning using hardware accelerated computation of generalized voronoi diagrams. In: International Conference on Robotics and Automation, pp. 2931–2937 (2000)
14. Hsu, D., Kindel, R., Latombe, J.C., Rock, S.: Randomized kinodynamic motion planning with moving obstacles. International Journal of Robotics Research **21**(3), 233–255 (2002)
15. Kalakrishnan, M., Chitta, S., Theodorou, E., Pastor, P., Schaal, S.: STOMP: Stochastic trajectory optimization for motion planning. In: Proceedings of IEEE International Conference on Robotics and Automation, pp. 4569–4574 (2011)
16. Kider, J., Henderson, M., Likhachev, M., Safonova, A.: High-dimensional planning on the gpu. In: Proceedings of IEEE International Conference on Robotics and Automation, pp. 2515–2522 (2010)
17. Koenig, S., Tovey, C., Smirnov, Y.: Performance bounds for planning in unknown terrain. Artificial Intelligence **147**(1-2), 253–279 (2003)
18. Kunz, T., Reiser, U., Stilman, M., Verl, A.: Real-time path planning for a robot arm in changing environments. In: Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on, pp. 5906 –5911 (2010)
19. Lee, S.H., Kim, J., Park, F., Kim, M., Bobrow, J.: Newton-type algorithms for dynamics-based robot movement optimization. Robotics, IEEE Transactions on **21**(4), 657–667 (2005)
20. Lengagne, S., Mathieu, P., Kheddar, A., Yoshida, E.: 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, pp. 698–703 (2010)
21. Likhachev, M., Ferguson, D.: Planning long dynamically feasible maneuvers for autonomous vehicles. International Journal of Robotics Research **28**(8), 933–945 (2009)
22. Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., Thrun, S.: Anytime dynamic A*: An anytime, replanning algorithm. In: Proceedings of the International Conference on Automated Planning and Scheduling (2005)
23. Lozano-Perez, T., O'Donnell, P.: Parallel robot motion planning. In: International Conference on Robotics and Automation, pp. 1000–1007 (1991)
24. Pan, J., Lauterbach, C., Manocha, D.: g-Planner: Real-time motion planning and global navigation using gpus. In: Proceedings of AAAI Conference on Artificial Intelligence (2010)
25. Park, C., Pan, J., Manocha, D.: 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)
26. Petti, S., Fraichard, T.: Safe motion planning in dynamic environments. In: Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 2210–2215 (2005)
27. Phillips, M., Likhachev, M.: Planning in domains with cost function dependent actions. In: Proceedings of AAAI Conference on Artificial Intelligence (2011)
28. Phillips, M., Likhachev, M.: SIPP: Safe interval path planning for dynamic environments. In: Proceedings of IEEE International Conference on Robotics and Automation, pp. 5628–5635 (2011)
29. Pisula, C., Hoff, K., Lin, M.C., Manocha, D.: Randomized path planning for a rigid body based on hardware accelerated voronoi sampling. In: International Workshop on Algorithmic Foundation of Robotics, pp. 279–292 (2000)
30. Plaku, E., Kavraki, L.: Distributed sampling-based roadmap of trees for large-scale motion planning. In: Proceedings of IEEE International Conference on Robotics and Automation, pp. 3868–3873 (2005)
31. Quinlan, S., Khatib, O.: Elastic bands: connecting path planning and control. In: Proceedings of IEEE International Conference on Robotics and Automation, pp. 802–807 vol.2 (1993)
32. Ratliff, N., Zucker, M., Bagnell, J.A.D., Srinivasa, S.: CHOMP: Gradient optimization techniques for efficient motion planning. In: Proceedings of International Conference on Robotics and Automation, pp. 489–494 (2009)
33. Rinnooy Kan, A., Timmer, G.: Stochastic global optimization methods part i: Clustering methods. Mathematical Programming **39**, 27–56 (1987)
34. Wilkie, D., van den Berg, J.P., Manocha, D.: Generalized velocity obstacles. In: Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 5573–5578 (2009)