

Closing the Loop between Motion Planning and Task Execution using Real-Time GPU-based Planners *

Jia Pan and Dinesh Manocha

Department of Computer Science, University of North Carolina at Chapel Hill
{panj, dm}@cs.unc.edu

Abstract

Many task execution techniques tend to repeatedly invoke motion planning algorithms in order to perform complex tasks. In order to accelerate the perform of such methods, we present a real-time global motion planner that utilizes the computational capabilities of current many-core GPUs (graphics processing units). Our approach is based on randomized sample-based planners and we describe highly parallel algorithms to generate samples, perform collision queries, nearest-neighbor computations, local planning and graph search to compute collision-free paths for rigid robots. Our approach can efficiently solve the single-query and multi-query versions of the planning problem and can obtain one to two orders of speedup over prior CPU-based global planning algorithms. The resulting GPU-based planning algorithm can also be used for real-time feedback for task execution in challenging scenarios.

Introduction

Motion planning algorithms have been widely used to perform different tasks for autonomous robots. In case of simple tasks, a single call to the motion planning algorithm may be sufficient to execute the tasks. For example, if the task is to compute a collision-free path for the robot in an environment with static obstacles, we can formalize it as a path finding problem in the configuration space and solve it by the randomized planning algorithms (Guitton and Farges 2009). In many cases, the motion planning steps and sub-task execution are usually performed in an interleaved manner, i.e. the planning result for one of the subtasks may be used to construct the formulation of the following subtasks (Talamadupula, Benton, and Schermerhorn 2009). In other words, we use the motion planner as a feedback step for the high-level task execution algorithm. Next, we highlight several cases where different types of tasks are performed in this manner, including task execution in dynamic environments, and tasks with multiple constraints or components.

*We would like to thank Jean-Paul Laumond for some of the models. This research was supported in part by ARO Contract W911NF-04-1-0088, NSF awards 0636208, 0917040 and 0904990, DARPA/RDECOM Contract WR91CRB-08- C-0137, and Intel.

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Task Execution in Dynamic Environments

A dynamic environment typically consists of moving obstacles (Jaillet and Siméon 2004) or sensors with uncertainty about the localization (Prentice and Roy 2009). In these cases, the robot has only partial observation or representation of the environment which makes it hard to have a complete prior specification of the task problem (Talamadupula, Benton, and Schermerhorn 2009). Therefore the robot needs to handle incomplete and evolving task representation. Due to the dynamic scene representation, the robot should update the information of the environment and modify the specification of the task. Each modified task is performed by a sequence of motion planning computations, and their results are used to execute the task. These steps are repeated whenever the obstacles move (Jaillet and Siméon 2004) or when additional sensor information is available (Prentice and Roy 2009).

Task Execution with Multiple Constraints

Many high-level tasks boil down to generating motion that can satisfy multiple constraints. Some of the constraints are difficult in terms of integrating into task's representation, such as generating natural-look motion for a human-like robot (Pan et al. 2010), nonholonomic constraints (Sánchez et al. 2006) or dynamics constraints (Yoshida et al. 2005). In practice, these tasks need to satisfy all the hard and soft constraints simultaneously, which can correspond to a complex multi-objective optimization and can be challenging for state-of-the-art motion planning algorithms. For example, the robot may need to manipulate one object to the target position while satisfying the dynamics constraints. A possible solution for tasks with multiple constraints is as follows. First, a "simpler" task with only partial constraints is solved efficiently with motion planning algorithms. Next, we modify the solution to compute a trajectory that can satisfy more constraints. If the modified trajectory violates some of the earlier constraints, we use motion planning (or replanning) algorithm to satisfy such constraints. This process is repeated until a trajectory that satisfies all the constraints is computed. Many previous works (Sánchez et al. 2006; Yoshida et al. 2005; Esteves, Arechavaleta, and Laumond 2005) use such scheme to compute a collision-free trajectory that can also satisfy all the task-specific constraints.

Task Execution with Multiple Components

In order to perform complicated tasks, some planning algorithms use a hybrid scheme that integrates a task planner with a motion planner (Guitton and Farges 2009). These methods first decompose a complicated task into many “primitive” or “simple” subtasks, then use a task planner to arrange them in suitable order and finally solve them by reducing to a series of motion planning computations. For example, a subtask graph is used to formalize the task planner (Hauser and Latombe 2009) and PRM-based motion planner is used to compute the trajectory for each subtask. A recursive task planner is used (Stilman et al. 2007) to arrange the position of the obstacles so as to find a valid manipulation in an environment with movable obstacles.

Real-Time Planning

The main factor that influences the efficiency of task execution is the performance of the motion planning algorithm that is invoked repeatedly in the interleaved task execution scheme. Moreover, in a dynamic environment, a real-time planner can reduce the delay of feedback loop and therefore improve the stability and robustness of task accomplishment algorithm.

In this paper we present a novel parallel algorithm for real-time motion planning that exploits the computational capability of a \$400 commodity graphics processing unit (GPU). Current GPUs are programmable many-core processors that can support thousands of concurrent threads and we use them for real-time computation of a global planning using sample-based probabilistic roadmaps (PRM). We describe efficient parallel strategies for the construction phase that include sample generation, collision detection, connecting nearby samples and local planning. The query phase is also performed in parallel based on graph search. In order to design an efficient single query planner, we use a lazy strategy that defers collision checking and local planning till the search step is performed.

In practice, our algorithm can generate thousands of samples for rigid robots with 3 or 6 DOFs and compute the roadmap for these samples at close to interactive rates including construction of all hierarchies. It performs no pre-computation and is applicable to dynamic scenes, articulated models or non-rigid robots. We highlight its performance on multiple benchmarks on a commodity PC with a NVIDIA GTX 285 GPU and observe a 10-80 times performance improvement over prior CPU-based implementations of the same algorithm. We believe that such a planner can be used to accelerate many task-execution schemes.

The rest of the paper is organized as follows. We give a brief overview of the GPU motion planning framework in Section 2 and present some of the details in Section 3. Finally we highlight the results in Section 4.

Overview

In this section, we give an overview of the GPU-based motion planning framework, which can be used for single-query and multiple-query problems.

We choose PRM as the underlying motion planning algorithm, because it is more suitable to exploit the multiple cores and data parallelism on GPUs. The PRM algorithm is composed of several steps and each step performs similar operations on the input samples or on the links joining those samples.

The PRM algorithm has two phases: roadmap construction and query phase. The roadmap construction phase includes four main steps: 1) generate samples in the configuration space; 2) compute milestones that correspond to samples in the free space by performing discrete collision queries; 3) for each milestone, find other milestones that are closest to it; 4) connect nearby milestones using local planning and form a roadmap. The query phase includes two parts: 1) connect initial and goal configurations of query to the roadmap; 2) execute a graph search algorithm on the roadmap and find collision free paths. The basic flowchart of PRM is shown in the left part of Fig 1. We use a many-core GPU to improve the performance of each component significantly and the framework for the overall GPU-based planner is shown in the right side of Fig 1.

We use MD5 cryptographic hash function (Tzeng and Wei 2008) to generate random samples for each thread independently. Unlike most existing ‘parallel’ random number generators (Howes and Thomas 2007), which tends to batch multiple sequential generators together, we use a real parallel random generator (Tzeng and Wei 2008).

For each sample generated, we need to check whether it is a milestone, i.e. does not collide with the obstacles. We first compute the bounding volume hierarchy (BVH) for both the obstacles and the robot on GPU using a parallel tree construction algorithm (Lauterbach et al. 2009). Next, each thread performs a hierarchical collision detection between BVHs to test for the overlap between the obstacles and the robot in the configuration corresponding to a given sample.

For each milestone, we perform k -nearest neighbor (KNN) query to find its nearest neighbors. We have tried two different KNN algorithms. In (Pan, Lauterbach, and Manocha 2010b), we use parallel KNNs based on BVHs, which is effective but has three drawbacks: 1) it is difficult to extend to high-dimensional cases; 2) it is hard to compute tight bounds on the runtime performance; 3) it is difficult to control the approximation level of the nearest neighbors that are computed. As a result, in our recent work (Pan, Lauterbach, and Manocha 2010), we instead use the local sensitive hashing (LSH) based KNN search algorithm to overcome these issues.

Once the roadmap is constructed by performing the local planning step among the milestones, we connect the initial and the goal configurations to the roadmap, using the KNN query. Finally we perform parallel graph search on the roadmap to obtain a collision-free path.

Parallelized PRM Motion Planner on GPU

In this section, we give details about how some important components of our algorithm are parallelized. For further details, please refer to (Pan, Lauterbach, and Manocha 2010b) and (Pan, Lauterbach, and Manocha 2010).

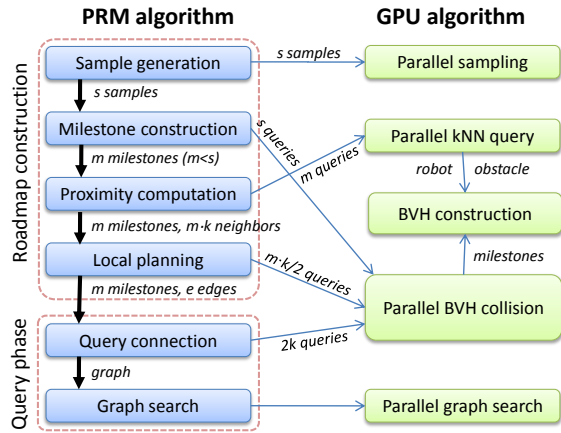


Figure 1: Overview of the GPU-based real-time planner (Pan, Lauterbach, and Manocha 2010b).

Hierarchy Computation

We construct bounding volume hierarchies (BVH) for the robot and for all the obstacles in the environment to accelerate the collision queries. We use the GPU-based construction algorithm introduced in (Lauterbach et al. 2009) to construct the *oriented bounding boxes* (OBB) hierarchy for a given triangled model in parallel on a GPU, which can provide higher culling efficiency for collision detection as compared to other BVHs.

Roadmap Construction

Roadmap construction phase tries to capture the connectivity of free configuration space, which is the main computational intensive part of PRM algorithm.

Milestone Computation We first generate random samples for each thread independently. Then for each configuration, we need to check whether it is a milestone, i.e. lies in the free space and does not collide with the obstacles. We use a hierarchical collision detection approach using BVHs to test for overlap between the obstacles and the robot. The collision detection is performed in each thread by using a traversal algorithm in the two BVHs.

Proximity Computation For each milestone computed, we need to find its k -nearest neighbors (KNN). Here we introduce our new KNN search algorithm based on locality-sensitive hashing, which outperforms previous GPU-based KNN search algorithms like the brute-force method (Garcia, Debreuve, and Barlaud 2008) or BVH-based method (Pan, Lauterbach, and Manocha 2010b).

Locality-sensitive hashing (LSH) is one popular algorithm for approximate clustering in high-dimensional spaces (Indyk and Motwani 1998). The key idea is to hash the points using several hashing functions to ensure that for each function the probability of collision is much higher for points that are close to each other than for those that are far apart. One formal description is:

$$\mathbb{P}_{h \in \mathcal{F}}[h(\mathbf{x}) = h(\mathbf{y})] \propto \text{sim}(\mathbf{x}, \mathbf{y}),$$

where $h(\cdot)$ is the hash function belongs to the so called *locality-sensitive hash* (LSH) function family \mathcal{F} ; \mathbf{x}, \mathbf{y} are two points in high-dimensional space; $\text{sim}(\cdot)$ is a function to measure the similarity between \mathbf{x} and \mathbf{y} : for KNN search, large $\text{sim}(\mathbf{x}, \mathbf{y})$ means that the distance between \mathbf{x} and \mathbf{y} is small. Different function families \mathcal{F} can be used for different metrics. Here we use the LSH families for l_2 metric introduced in (Datar et al. 2004).

The LSH-based KNN search structure is a hash table: points with the same hash values are pushed into the same bucket of hash table. The query process is to scan locally within the buckets which the query point is hashed to. LSH-based KNN search can perform one query in nearly constant time, which means the nearest neighbor in motion planning algorithm can be completed within linear time complexity to the number of samples.

To implement an efficient LSH-based KNN search on GPU, we need a hash-table structure specifically designed for the parallel architecture. We use the parallel cuckoo hashing (Alcantara et al. 2009) as the KNN search data structure. Cuckoo hashing places at most one item at each location in the hash table by allowing items to be moved after their initial placement. It stores the key-index pairs in f hash sub-tables ($f \geq 3$). Cuckoo hashing has three advantages. First its space overload is small. Secondly, cuckoo hashing can provide collision-free storage with very high probability. Finally, the lookups take only constant time.

Local Planning Local planning checks whether there is a local path between two milestones, which corresponds to an edge on the roadmap. Many methods are available for local planning. The most common way is to discretize the path between the two milestones into n_i steps and we claim the local path exists when all the intermediate samples are collision-free by performing discrete collision queries (DCD) at those steps. Local planning is the most expensive part of the PRM algorithm. For multi-query problems, its cost can be amortized over multiple queries as the roadmap is constructed only once. For a single-query problem, computing the whole roadmap is too expensive. Therefore, in the single-query case, we use a lazy strategy to defer local planning until absolutely needed, which can greatly improve performance for single queries.

Query Phase

The query phase includes two parts: connecting queries to the roadmap and executing graph search to find paths. We connect queries to the roadmap with algorithms similar in proximity computation. And we use parallelized DFS or BFS algorithm to efficiently search paths on the roadmap.

Results

In this section, we highlight the performance of our algorithm on a set of benchmarks. We implement the PRM on the GPU (G-PRM) for multi-query planning problems and its lazy version (GL-PRM) for single-query problems. We compare them with the PRM and RRT algorithms implemented in the OOPSMP library. The benchmarks used are

	C-PRM	C-RRT	G-PRM	GL-PRM
piano	6.53s	19.44s	1.71s	111.23ms
helicopter	8.20s	20.94s	2.22s	129.33ms
maze3d1	138s	21.18s	14.78s	71.24ms
maze3d2	69.76s	17.4s	14.47s	408.6ms
maze3d3	8.45s	4.3s	1.40s	96.37ms
alpha1.5	65.73s	2.8s	12.86s	1.446s

Table 1: The left two columns highlight the implementations of PRM and RRT algorithm in the OOPSMP. The right two columns highlight the performance of our GPU-based algorithms.

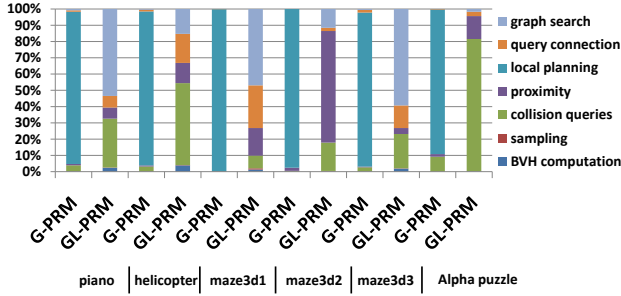


Figure 3: Split-up of timings: the fraction of time spent in parts of algorithm differ between G-PRM and GL-PRM.

shown in Fig 2. Table 1 shows the comparison of timings between algorithms. In general G-PRM is about 10 times faster than C-PRM and GL-PRM can provide another 10 times of acceleration for single query problems. C-RRT is usually faster than C-PRM. G-PRM is fast and can even handle dynamic scenes.

Fig 3 shows the timing breakdown between various steps for G-PRM and GL-PRM and the difference between the performance of two algorithms is clear: In G-PRM, local planning is the bottleneck which dominates the timing while in GL-PRM graph search takes longer time because local planning is performed in a lazy or output sensitive manner.

We test the scalability of G-PRM and GL-PRM on the *maze3d3* benchmark and the result is shown in Fig 4. It is obvious that GL-PRM is generally faster than G-PRM and both algorithms achieve near-linear scaling on the benchmark. However, notice that the performance of GL-PRM reduces faster than G-PRM. The reason is that when the number of samples increases, proximity computation becomes more and more expensive and dominates the timing when the number of samples is near 1 million.

Conclusions and Future Work

In this paper, we describe the GPU-based real-time motion planning algorithm, which can be used in an interleaved task execution architecture in order to improve the robot’s performance in order to accomplish complex tasks. Our algorithm exploits the parallelism within the PRM framework. The method provides 1-2 orders of magnitude higher performance over previous CPU-based planners. This makes

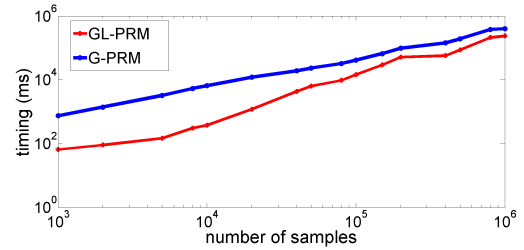


Figure 4: The scalability of G-PRM and GL-PRM algorithms.

our approach as the first method that can perform real-time motion planning and global navigation in general environment. The real-time planner is invoked repeatedly by a task execution algorithm and therefore can greatly improve the overall efficiency.

There are many avenues for future work. We are interested in extending the GPU planning algorithms to high-DOF articulated models. We are also interested in using exact continuous collision detection (CCD) algorithms for local planning. Finally, we would like to apply our real-time planner on robots to solve complicated task execution problems, especially the different scenarios corresponding to dynamic scenes and handling multiple constraints.

References

Alcantara, D. A.; Sharf, A.; Abbasinejad, F.; Sengupta, S.; Mitzenmacher, M.; Owens, J. D.; and Amenta, N. 2009. Real-time parallel hashing on the gpu. In *SIGGRAPH Asia*, 1–9.

Datar, M.; Immorlica, N.; Indyk, P.; and Mirrokni, V. S. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry (SCG)*, 253–262.

Esteves, C.; Arechavaleta, G.; and Laumond, J.-P. 2005. motion planning for human-robot interaction in manipulation tasks. In *Proceedings of IEEE International Conference on Mechatronics and Automation*, 1766–1771.

Garcia, V.; Debreuve, E.; and Barlaud, M. 2008. Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, CVPRW’08*, 1–6.

Guitton, J., and Farges, J.-L. 2009. Taking into account geometric constraints for task-oriented motion planning. In *ICAPS Workshop on Bridging the Gap Between Task and Motion Planning*, 26–33.

Hauser, K., and Latombe, J.-C. 2009. Integrating task and prm motion planning: Dealing with many infeasible motion planning queries. In *ICAPS Workshop on Bridging the Gap Between Task and Motion Planning*, 19–23.

Howes, L., and Thomas, D. 2007. Efficient random number generation and application using cuda. In Nguyen, H., ed., *GPU Gem 3*.

Indyk, P., and Motwani, R. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *ACM symposium on Theory of computing (STOC)*, 604–613.

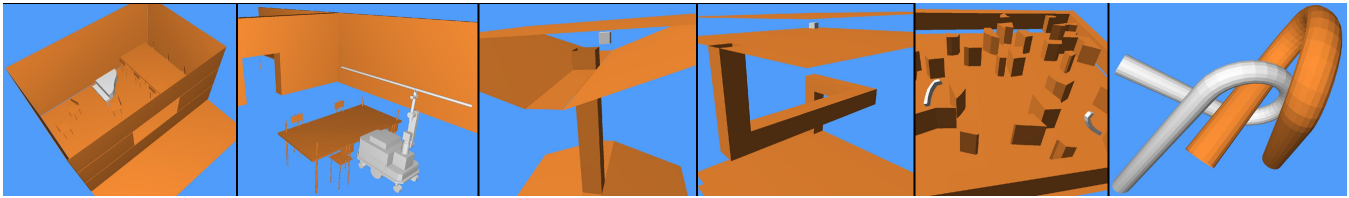


Figure 2: The benchmark scenes used for our algorithms in this order: piano (2484 triangles), helicopter (2484 triangles), maze3d1 (40 triangles), maze3d2 (40 triangles), maze3d3 (970 triangles), alpha puzzle (2016 triangles).

Jaillet, L., and Siméon, T. 2004. A prm-based motion planner for dynamically changing environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1606–1611.

Lauterbach, C.; Garland, M.; Sengupta, S.; Luebke, D.; and Manocha, D. 2009. Fast bvh construction on gpus. *Computer Graphics Forum* 28(2):375–384.

Pan, J.; Zhang, L.; Lin, M. C.; and Manocha, D. 2010. A hybrid approach for synthesizing human motion in constrained environments. In *International Conference on Computer Animation and Social Agents (CASA)*, to appear.

Pan, J.; Lauterbach, C.; and Manocha, D. 2010. Efficient nearest-neighbor computation for gpu-based motion planning. Technical report, Department of Computer Science, University of North Carolina.

Pan, J.; Lauterbach, C.; and Manocha, D. 2010b. G-planner: Real-time motion planning and global navigation using gpus. In *AAAI Conference on Artificial Intelligence (AAAI)*, to appear.

Prentice, S., and Roy, N. 2009. The belief roadmap: Efficient planning in belief space by factoring the covariance. *International Journal of Robotics Research* 28(11-12):1448–1465.

Sánchez, A.; Cuautele, R.; Zapata, R.; and Osorio, M. 2006. A reactive lazy prm approach for nonholonomic motion planning. In *Advances in Artificial Intelligence (IBERAMIA-SBIA)*, 542–551.

Stilman, M.; Ullrich Schamburek, J.; Kuffner, J.; and Asfour, T. 2007. Manipulation planning among movable obstacles. In *IEEE International Conference on Robotics and Automation (ICRA)*, 3327–3332.

Talamadupula, K.; Benton, J.; and Schermerhorn, P. 2009. Integrating a closed world planner with an open world. In *ICAPS Workshop on Bridging the Gap Between Task and Motion Planning*.

Tzeng, S., and Wei, L.-Y. 2008. Parallel white noise generation on a gpu via cryptographic hash. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 79–87.

Yoshida, E.; Belousov, I.; Esteves, C.; and Laumond, J.-P. 2005. Humanoid motion planning for dynamic tasks. In *Proceedings of IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, 1–6.