Inter Symp of Robotics Research (ISRR), Siena, Italy, 2003. Springer-Verlag, STAR, vol. 15, pp. 80–89

Probabilistic Roadmaps of Trees for Parallel Computation of Multiple Query Roadmaps

Mert Akinc, Kostas E. Bekris, Brian Y. Chen, Andrew M. Ladd, Erion Plaku, and Lydia E. Kavraki

Rice University
Department of Computer Science
Houston, TX, 77005
{makinc,bekris,brianyc,aladd,plakue,kavraki}@cs.rice.edu

Abstract. We propose the combination of techniques that solve multiple queries for motion planning problems with single query planners in a motion planning framework that can be efficiently parallelized. In multiple query motion planning, a data structure is built during a preprocessing phase in order to quickly respond to on-line queries. Alternatively, in single query planning, there is no preprocessing phase and all computations occur during query resolution. This paper shows how to effectively combine a powerful sample-based method primarily designed for multiple query planning (the Probabilistic Roadmap Method - PRM) with sample-based tree methods that were primarily designed for single query planning (such as Expansive Space Trees, Rapidly Exploring Random Trees, and others). Our planner, which we call the Probabilistic Roadmap of Trees (PRT), uses a tree algorithm as a subroutine for PRM. The nodes of the PRM roadmap are now trees. We take advantage of the very powerful sampling schemes of recent tree planners to populate our roadmaps. The combined sampling scheme is in the spirit of the non-uniform sampling and refinement techniques employed in earlier work on PRM. PRT not only achieves a smooth spectrum between multiple query and single query planning but it combines advantages of both. We present experiments which show that PRT is capable of solving problems that cannot be addressed efficiently with PRM or single-query planners. A key advantage of PRT is that it is significantly more decoupled than PRM and sample-based tree planners. Using this property, we designed and implemented a parallel version of PRT. Our experiments show that PRT distributes well and can easily solve high dimensional problems that exhaust resources available to single machines.

1 Introduction

Sample-based planners have been used extensively during the last decade for multiple query or single query motion planning [6,9,10,12,14,16]. In multiple query motion planning, a data structure, typically a graph, is built during a preprocessing phase in order to quickly respond to on-line queries [6,10,13]. Alternatively, in single query planning, there is no preprocessing phase and all computations occur during query resolution. Such planners typically explore the space using a single or a bi-directional tree [4,9,14,16]. Recent papers (e.g., [9,14]) contain extensive references to sample-based motion planners.

The Probabilistic Roadmap Method (PRM) is an efficient and easy to implement planner primarily designed for multiple query motion planning problems [10].

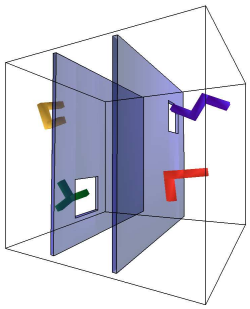


Fig. 1: A scene from our benchmarks. In problem "narrow4h2" each robot must go through two very narrow passages to the outer side of the opposite wall.

PRM operates by sampling "milestones" (configurations) in the free configuration space and connecting them using a local planner. Although a typical implementation uses a very simple local planner and uniform pseudo-random sampling, it has been shown that a variety of alternate approaches ranging in sophistication and cost can be applied without sacrificing correctness in hopes of obtaining a faster planner [8]. Indeed, two of the key and most studied issues in the context of PRM are the power of the local planner and the way sampling is performed. For recent work see [2,14].

In this paper we replace the local planner of PRM with a powerful single query sample-based motion planner. We call our planner the Probabilistic Roadmap of Trees (PRT) [5]. Among the single query planners that have been developed recently, Expansive Space Trees (ESTs) [9] and Rapidly Exploring Random Trees (RRTs) [14] have been very successful and are used in our work. However, other sample-based tree planners can be used (e.g., [12]). Our work is important in many respects. In particular, we obtain a planner which is faster than PRM and more robust than the tree planners that we used, namely ESTs and RRTs. Moreover, PRT provides a smooth spectrum between single query and multiple query planning that combines the advantages of both. Furthermore, we take advantage of recent very effective sampling methods employed by ESTs and RRTs and provide a new sampling scheme for PRM. It should be noted that the proposed overall sampling of PRT is in the spirit of non-uniform sampling and refinement techniques used in earlier work of PRM. Last but not least, we obtain a planner which is significantly more decoupled than PRM and tree planners such as ESTs and RRTs and can be parallelized effectively. We designed and implemented a parallel version of PRT. Although many subroutines of PRM can be run effectively in a highly distributed fashion, efficient coordination of various processing resources requires significant additional algorithmic design. By increasing the power of the local planner and by using more complex milestones, PRT distributes its computation almost evenly among processors, requires little communication, and allows us to solve very high dimensional problems and problems that exceed the resources available to the sequential implementation.

This paper presents experiments with up to 36 degrees-of-freedom (DOF) where PRT obtains a solution at a fraction of the running time needed by PRM, EST, or RRT. Figure 1 shows an example. We were able to obtain nearly linear speedup for parallel PRT.

Our long term goal is to study high-dimensional problems [12] such as those arising in planning with flexible objects [13], reconfigurable robots [17], complex planning instances [16], and computational biology search problems [1,3]. Such problems test the limits of current planner implementations. One important avenue of untapped potential is in making effective use of parallelism in motion planning. Our work describes a robust planner, which provides a smooth transition from single query to multiple query planners and can be used for problems that are beyond the capabilities of current planners.

2 The PRT Planner

In this section, we describe the basic operation of the PRT algorithm [5]. PRT constructs a roadmap aiming at capturing the connectivity of the free configuration space, $\mathcal{C}_{\text{free}}$. The nodes of the roadmap are not single configurations but trees, which are referred to as milestones. Connections between milestones are computed by using sample-based tree planners. The tree planners that we have used are RRTs [14] and ESTs [9]. The pseudocode for PRT is given in Algorithm 1.

A roadmap is an undirected graph $G = (V, E)$ over a finite set of configurations $V \subset \mathcal{C}_{\text{free}}$ and each edge $(u, v) \in E$ represents a local path from u to v. The undirected graph $G_T = (V_T, E_T)$ is an induced subgraph of the roadmap which is defined by partitioning G into a set of subgraphs $T_1, \ldots, T_K$ which are trees and contracting them into the vertices of G_T. In other words, $V_T = \{T_1, \ldots, T_K\}$ and $(T_i, T_j) \in E_T$ if there exists $v_i \in T_i$ and $v_j \in T_j$ such that v_i and v_j have been connected by a local path. As shown in Algorithm 1, the roadmap construction proceeds in three stages: milestone computation (lines 1–6), edge selection (lines 7–11), and edge computation (lines 12–16).

In PRT, the trees T_i or milestones of the roadmap G_T are computed by sampling their roots uniformly at random in $\mathcal{C}_{\text{free}}$ and then growing the trees using a sample-based tree planner which has as its goal expansion and exploration. We have found RRTs [14] and ESTs [9] to be suitable to PRT, but other sample-based tree planners (e.g., [12]) can be used as well.

The selection of candidate edges is governed by two parameters n_{close} and n_{random}. Each milestone T_i defines a representative configuration q_i which is computed as an aggregate of the configurations in T_i. Our implementation uses the centroid. If $Q = \{q_1, \ldots, q_K\}$ is the set of centroids, then for each i, we determine n_{close} closest and n_{random} configurations q_j to q_i such that $i \neq j$ and set each (T_i, T_j) as a candidate edge. The notion of closeness is determined by the metric d. The graph of candidate edges is denoted $G_C = (V_T, E_C)$.

The objective of our planner is to determine the existence of a path. To this end, we avoid computing a candidate edge unless placing that edge in E_T would

Algorithm 1: PRT

1: $V_T \leftarrow \emptyset$, $E_T \leftarrow \emptyset$, $Q \leftarrow \emptyset$, $E_C \leftarrow \emptyset$.
2: **while** $|V_T| < K$ **do**
3: $T \leftarrow$ build tree with root a randomly chosen free configuration.
4: $V_T \leftarrow V_T \cup \{T\}$.
5: $Q \leftarrow Q \cup \{q_T\}$, where q_T is the representative of T.
6: **end while**
7: **for all** $T \in V_T$ **do**
8: $S_{\text{close}} \leftarrow$ a set of n_{close} closest $q \in Q$ to q_T.
9: $S_{\text{random}} \leftarrow$ a set of n_{random} random $q \in Q$ to q_T.
10: $E_C \leftarrow E_C \cup \{(T, T') : q_{T'} \in S_{\text{close}} \cup S_{\text{random}}\}$.
11: **end for**
12: **for all** $(T_1, T_2) \in E_C$ **do**
13: **if not** *component*(G, T_1, T_2) **and** *tree-planner*(T_1, T_2) **then**
14: $E_T \leftarrow E_T \cup \{(T_1, T_2)\}$.
15: **end if**
16: **end for**

---

decrease the number of connected components in G_T. Then, for each candidate edge (T_i, T_j), a number of close pairs of configurations of T_i and T_j are quickly checked with a fast deterministic local planner, *i.e.*, a straight-line planner. If any local path is found, the edge (T_i, T_j) is added to E_T and no further computation takes place. Otherwise, a more complex tree-connection algorithm is executed, e.g., bi-directional RRT, EST, or other similar algorithms. During the tree connection additional configurations are typically added to the trees T_i and T_j.

3 Parallel Planning

In this section, we describe the design and implementation of a parallel version of PRT. Before relating the details, we discuss data and control flow dependency in each stage of the PRT algorithm. During milestone computation, there are no dependencies. Each single milestone can be processed in parallel. Additional parallelization is stymied by the sampling scheme we use to generate milestones and would be considerably more involved. Random edge selection can be done in parallel; however, the distribution of the closest edge selection is more difficult since it requires the construction of a search structure that depends on the representatives of the milestones. Finally, edge computations are not entirely independent of eachother. Since milestones can change after an edge computation and since computing an edge requires direct knowledge of both milestones, the edge computations cannot be efficiently parallelized without some effort. Furthermore, computation pruning due to component analysis entails control flow dependencies throughout the computation of the edges. Our experiments with the sequential implementation revealed that the bulk of the run time occurs in milestone and edge computation.

PRT for Parallel Computation of Multiple Query Roadmaps

Algorithm 2: Hierarchical Operation of Parallel PRT

Scheduler		**Processor** P_i
1: Synchronize with processors. 2: COMPUTE MILESTONES. 3: COMPUTE EDGES.	PARALLEL PRT	1: Wait for synchronization. 2: COMPUTE MILESTONES. 3: COMPUTE EDGES.
1: $Q \leftarrow \emptyset$. 2: $i \leftarrow 0$. 3: **while** $i < K$ **do** 4: Wait for some T_{rep} to arrive. 5: $Q \leftarrow Q \cup \{T_{\text{rep}}\}$. 6: $i \leftarrow i + 1$. 7: **end while** 8: Broadcast `finish` to processors.	COMPUTE MILESTONES	1: $T_{P_i} \leftarrow \emptyset$. 2: Post request for message from scheduler. 3: **while** `finish` has not been received **do** 4: $T \leftarrow$ generate a milestone. 5: $T_{P_i} \leftarrow T_{P_i} \cup \{T\}$. 6: Send T_{rep} to the scheduler. 7: **end while**
1: $G_C = (V_T, E_C) \leftarrow$ graph of candidate edges. 2: $L_{P_i} = (V_i, E_i) \leftarrow$ empty graph, for all P_i. 3: $W = \{P_1, \ldots, P_n\}$. 4: **while** unprocessed edges remain in G_C **do** 5: COMPUTE PARTITIONS. 6: **for** i: $P_i \in W$ **and** $\lvert E_i \rvert > 0$ **do** 7: $e \leftarrow$ randomly selected from E_i. 8: Send e to P_i. 9: $E_i \leftarrow E_i - \{e\}$. 10: $W \leftarrow W - \{P_i\}$. 11: **end for** 12: **if** computed edges have arrived **then** 13: update connected components. 14: remove from G_C and L_{P_i}'s all the unnecessary edges. 15: **end if** 16: **end while** 17: Broadcast `finish` to processors.	COMPUTE EDGES	1: Post request for message from scheduler. 2: **while** `finish` has not been received **do** 3: **while** no message has been received **do** 4: Complete a pending send operation. 5: Complete a pending receive operation. 6: **end while** 7: **if** partition message has been received **then** 8: COMPUTE PARTITIONS. 9: **end if** 10: **if** $e = (v_1, v_2)$ has been received **then** 11: Complete pending receive operations (if any) on T_{v_1} and T_{v_2}. 12: Try to connect T_{v_1} and T_{v_2}. 13: Send result to scheduler. 14: **end if** 15: Post request for message from scheduler. 16: **end while**
1: $S = \{P_i : E_i = \emptyset\}$. 2: Compute $G'_C = (V_S, E_S)$, where 3: $V_S = \bigcup_{P \in S} V_P$, and 4: $E_S = \{(v_1, v_2) \in E : v_1, v_2 \in V_S\}$. 5: Partition G'_C into L_{P_i}'s for $P_i \in S$. 6: **for** i: $P_i \in S$ **do** 7: $\texttt{map}_v \leftarrow P_i$ for all $v \in V_{P_i}$. 8: **end for** 9: Send `map` to P_i for all $P_i \in S$.	COMPUTE PARTITIONS	1: Complete all pending send operations. 2: Complete all pending receive operations. 3: Receive `map` from server. 4: **for** $i = 1$ to `nr_ms` **do** 5: **if** $T_i \in T_{P_i}$ **and** $P_i \neq \texttt{map}_i$ **then** 6: Post request to send T_i to $\texttt{map}_i$. 7: **end if** 8: **if** $T_i \notin T_{P_i}$ **and** $P_i = \texttt{map}_i$ **then** 9: Post request to receive T_i from $\texttt{map}_i$. 10: **end if** 11: **end for**

We have chosen a scheduler–processor architecture for our parallel implementation. The processors are responsible for milestone and edge computations. The scheduler arbitrates milestone ownership, handles edge selection, assigns edge candidates to processors, and manages the connected component data structure. Parallel PRT is described in Algorithm 2.

During the milestone computation stage, each processor P_i computes a set T_{P_i} of milestones and sends to the scheduler their representatives until a predefined total number K of milestones have been computed. We call the subgraph of G_C induced by T_{P_i}, the local graph, $L_{P_i} = (T_{P_i}, E_{P_i})$. The edges of L_{P_i} are those which processor P_i can compute without communicating with other processors. During the edge computation, for each i, the scheduler selects an edge e_i uniformly at random from L_{P_i}, deletes e_i from G_C and L_i, and assigns the computation of e_i to processor P_i. If the edge connection is successful, then e_i is added to G_T. Then all edges $(T_i, T_j) \in G_C$ such that T_i and T_j lie in the same connected component of G_T are deleted from G_C as they will not change the connected component structure of G_T. The above steps are repeated until there are no more edges in G_C. At each step, certain L_{P_i}'s may be empty due to edge deletions and cause some of the processors, say $P_1, \ldots, P_N$, to become idle. Our implementation avoids this problem by repartitioning the milestones owned by these processors. Given the graph G_C, we formulate the problem of finding "good" graph partitions as an optimization problem: determine a partition $T_{P_1}, \ldots, T_{P_N}$ of the milestones that maximizes $\sum_{i=1}^{N} |E_C \cap E_{P_i}|$. This is an instance of the graph partition into N parts problem which is known to be NP-hard for $N \geq 2$. We partition the graph using the classical Kernighan-Lin algorithm [11] which is a greedy local optimization approach. Once the partitions are computed, they must be assigned to the processors in such a way that the number of milestones that need to be exchanged is minimized. This is an instance of the maximum bipartite matching problem and can be solved efficiently with the Hungarian algorithm [15].

4 Experiments and Results

The experiments in this paper were chosen for two purposes: to test PRT on problems that cannot be efficiently solved by PRM and single-query planners and to evaluate parallel PRT performance compared to the sequential implementation.

Benchmarks We ran our experiments on a set of benchmarks chosen to vary in type and in difficulty. Problems "fence2" and "fence4" consisted of two and four non-convex parts, respectively, in a box split by a regular fence-like wall (Figure 2). Problem "narrow4h2" consisted of four non-convex parts and two walls with two disjoint small square holes (Figure 1). Problem "narrow6" consisted of six non-convex parts and a single wall with a small square hole in it. Problems "random4" and "random-chain" consisted of four non-convex parts and a 12-DOF articulated arm represented as an open kinematic chain, respectively, in a box filled with random objects. Problem "puma-maze" consisted of a 6R articulated limb similar to a Puma560 surrounded by several vertical bars (Fig 1 and 2).

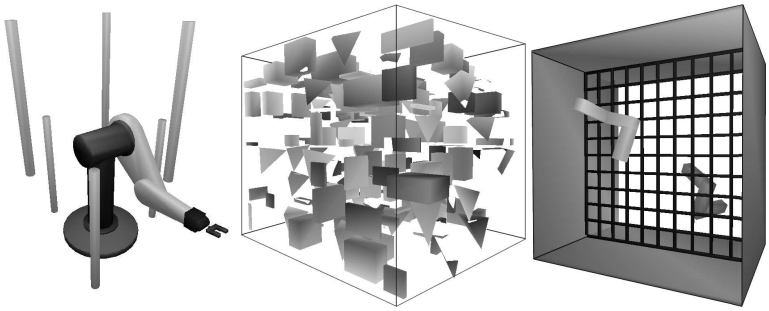


Fig. 2: Scenes from our benchmarks. From left to right, the depicted scenes are "puma box", "random4", and "fence2".

Hardware and Software Setup The implementation was carried out in Ansi C/C++ using the GNU compilers and libraries. Additionally, we made use of the SWIFT++ collision detection library [7], the Atlas2 implementation of LAPACK for numerical routines, the MPICH implementation of MPI standard for communication and OpenGL for visualization. The processing nodes consisted of eleven dual AMD Athlon 1900MPs with 1GB of memory each. The scheduler node was an AMD Athlon 1800XP with 500MB of memory. The network topology was switched 100Mbps for the processing nodes with a 1Gbps backbone to the scheduler node. All of the nodes ran Debian Linux with kernel 2.4.21.

Comparison of PRT *with Other Planners* By setting parameters in different ways our implementation of PRT can be made into PRM, bi-directional RRT, or EST. We tested their performance on various difficult benchmarks. Our experiments showed that PRM, RRT, or EST could not solve the "fence2" or "fence4" problems even after 8 hours of computation, while PRT was able to solve these problems in 868.18 and 3307.14 seconds on average, respectively. We also tested these algorithms on "narrow6" and "narrow4h2" benchmarks. PRM was not able to solve any of these problems after several hours of computation, and for the cost of two or three bi-directional RRT or EST queries, we can preprocess the space with PRT to obtain a structure that answers queries more robustly and more quickly than these sample-based tree planners.

Measuring Parallel Efficiency To measure the parallel efficiency of PRT, we ran on various benchmarks the parallel code with 1, 2, 4, 8, 16 and 22 processors - the maximum number of processors we had available. Run times are averaged over several runs. In Table 1, we report results for PRT with RRT and EST as its local planners. In each case, we report time with one and twenty-two processors (time[1] and time[22]). Also, for the parallel runs, we report fraction of time spent in milestone computation (mc), edge computation, (ec), communication (comm), waiting (idle), and parallel efficiency (eff), which is calculated by $t_s/(t_f \cdot N)$, where t_s is sequential time, t_p is parallel time, and N is the number of processors.

Table 1: Parallel PRT versus Sequential PRT.

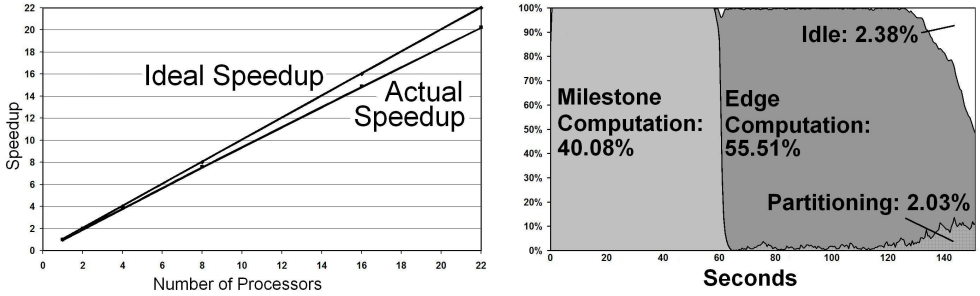
PRT with bi-directional RRT as the local planner.

benchmark	time[1](s)	time[22](s)	mc	ec	comm	idle	eff.
fence2	868.18	42.82	0.4108	0.4539	0.0965	0.0388	0.92
fence4	3307.40	151.84	0.4008	0.5551	0.0203	0.0238	0.99
narrow4h2	1666.95	93.21	0.3902	0.5028	0.0618	0.0452	0.81
narrow6	3131.71	173.41	0.4500	0.4509	0.0692	0.0299	0.82
random4	2242.39	125.56	0.3036	0.6412	0.0391	0.0161	0.81
random-chain	10050.48	512.28	0.2330	0.7219	0.0221	0.0230	0.89
puma-maze	8097.04	327.32	0.0248	0.8760	0.0844	0.0148	1.12

PRT with bi-directional EST as the local planner.

benchmark	time[1](s)	time[22](s)	mc	ec	comm	idle	eff.
fence2	872.78	41.54	0.2776	0.6311	0.0657	0.0256	0.95
fence4	3158.51	149.23	0.2365	0.7040	0.0449	0.0341	0.96
narrow4h2	1290.25	79.51	0.2715	0.5813	0.0936	0.0536	0.74
narrow6	2935.10	176.15	0.2605	0.6533	0.0583	0.0279	0.76
random4	1577.97	107.83	0.1317	0.7897	0.0488	0.0298	0.67
random-chain	10691.09	551.93	0.2186	0.7429	0.0155	0.0230	0.88
puma-maze	10207.89	414.10	0.0206	0.8939	0.0764	0.0091	1.12

Fig. 3: Parallel PRT Timings



In Figure 3, we present two plots of parallel PRT behavior. The plot on the left is for "fence2" and indicates the speedup obtained for different numbers of processors. The plot on the right is for "fence4" and presents logged data showing how processing nodes spend their time. These plots are characteristic of the behavior of the algorithm on the other benchmarks as well.

The overall efficiency of the parallel PRT is reasonably high on average 88.8% and in all our experiments in the range 67–99%. We also had a benchmark were superlinear, 1.12, speedup was obtained. Also, the speedup graph in Figure 3 is almost linear which suggests that the efficiency constant is not decaying with the number of processors. However, Algorithm 2 as presented places a load on the scheduler which is proportional to the number of processing nodes. As the number of processors in-

creases, this will eventually become a problem. A possible solution to this problem might be to increase the number of schedulers or to have a hierarchy of schedulers.

Nevertheless, there are several advantages of the parallel PRT algorithm. It is fairly simple and makes insignificant use of any blocking communication calls. Milestone and edge computations are also nearly fully distributed and storage is also distributed evenly.

Virtually all of the communication overhead occurs during the edge computations. This phase of the computation would be the most reasonable place to attempt to make further improvements. The graph partition scheme we used in our implementation optimized the sum of the number edges in the L_{P_i}'s. A better quantity to optimize would be to maximize the minimum number of edges over all L_{P_i}'s. This would favor better load balancing.

5 Discussion

We observed in our experiments that PRT is a powerful multiple query planner which combines advantages of traditional sampling-based single query and multiple query planners. By varying parameters, a smooth spectrum between single-query planners and PRM can be obtained from our PRT implementation. The sampling done in PRT has common attributes with earlier refinement and non-uniform sampling techniques used in PRM planning [10]. We believe that the efficiency of the PRT derives in part from offering the sample-based tree planner easier queries as they come from the closest neighbor clustering and the fact that the global sampling property of PRM is retained so that sample-based expansion heuristics, such as RRT and EST do not get trapped.

We observed that PRT exhibits similar behavior no matter whether bi-directional RRT or EST is being used as its local planner. Both RRTs and ESTs are well-known to be extremely sensitive to the interplay between the metric and the success of the planner[14]. We also made this observation in our implementation. In environments with thin obstacles, in particular the fence environment, RRT and EST tended to produce many configurations that were stuck near obstacles. In these environments RRT and EST are forced to do a similar amount of work to the PRM or PRT preprocessing phases to answer a single query. The efficiency of PRT is not limited to the specific single query planners that we used. In fact, other sample-based tree planners with good coverage properties can be substituted. Furthermore, we suggested a parallel implementation of PRT and obtained an efficient division of labor allowing PRT to tackle problems of unprecedented complexity.

We plan to scale our PRT implementation to a cluster with several hundred nodes. To do this, it is likely that some decentralization of the scheduling computations will become necessary. Our goal is to apply our work to increasingly hard planning problems dealing with flexible robots [13], reconfigurable robots [17], complex planning instances [16], and computational biology applications [3,1].

Acknowledgement Work on this paper by M. Akinc, K. Bekris, B. Chen, A. Ladd, E. Plaku, and L. Kavraki has been partially supported by NSF 9702288, NSF 0308237, NSF 0205671,

an REU supplement, a Whitaker Grant, and a Sloan Fellowship to L. Kavraki. A. Ladd is also partially supported by an FCAR grant. The authors thank AMD for supplying the processors where the experiments were carried on.

References

1. N. Amato, K. Dill, and G. Song. Using motion planning to map protein folding landscapes and analyze folding kinetics of known native structures. In *RECOMB*, pages 2–11, April 2002.
2. N. M. Amato, O. B. Bayazit, L. K. Dale, C. Jones, and D. Vallejo. Choosing good distance metrics and local planners for probabilistic roadmap methods. *TRA*, pages 442–447, 2000.
3. M. Apaydin, D. Brutlag, C. Guestrin, D. Hsu, and J. Latombe. Stochastic roadmap simulation: An efficient representation and algorithm for analyzing molecular motion. In *RECOMB*, April 2002.
4. J. Barraquand and J. Latombe. Robot motion planning: A distributed representation approach. *IJRR*, 10:628–649, 1991.
5. K. Bekris, B. Chen, A. Ladd, E. Plaku, and L. Kavraki. Multiple query motion planning using single query primitives. To appear at IROS 2003 TR03-422, Rice University, Houston, TX, July 2003.
6. P. Bessiere, E. Mazer, and J.-M. Ahuactzin. Planning in continuous space with forbidden regions: The ariadne's clew algorithm. In K. G. et al, editor, *Algorithmic Foundations of Robotics*, pages 39–47. A.K. Peters, Wellsley MA, 1995.
7. S. Ehmann and M. Lin. Accurate and fast proximity queries between polyhedra using surface decomposition. *Computer Graphics Forum (Proc. of Eurographics)*, 2001.
8. R. Geraerts and M. Overmars. A comparitive study of probabilistic roadmap planners. In *Proc. WAFR*, 2002.
9. D. Hsu, R. Kindel, J. Latombe, and S. Rock. Randomized kinodynamic motion planning with moving obstacles. *IJRR*, 2001.
10. L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *TRA*, 12(4):566–580, 1996.
11. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technology Journal*, 49:291–307, 1970.
12. A. Ladd and L. Kavraki. Motion planning for knot untangling. In *WAFR*, 2002.
13. F. Lamiraux and L. Kavraki. Planning paths for elastic objects under manipulation constraints. *IJRR*, 20(3):188–208, 2001.
14. S. LaValle and J. Kuffner. Rapidly exploring random trees: Progress and prospects. In B. Donald, K. Lynch, and D.Rus, editors, *WAFR*, pages 293–308. A.K. Peters, 2001.
15. C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.
16. G. Sanchéz and J.-C. Latombe. On delaying collision checking in prm planning - application to multi-robot coordination. *IJRR*, 21(1):5–16, 2002.
17. M. Yim. *Locomotion with a Unit-Modular Reconfigurable Robot*. PhD thesis, Stanford Univ., December 1994. Stanford Technical Report STAN-CS-94-1536.