IEEE Inter Conf on Robotics and Automation (ICRA), Barcelona, Spain, 2005, pp. 3879–3884

Distributed Sampling-Based Roadmap of Trees for Large-Scale Motion Planning

Erion Plaku and Lydia E. Kavraki
Rice University
Department of Computer Science
Houston, Texas 77005, USA
{plakue, kavraki}@cs.rice.edu

***Abstract*— High-dimensional problems arising from complex robotic systems test the limits of current motion planners and require the development of efficient distributed motion planners that take full advantage of all the available resources.**

This paper shows how to effectively distribute the computation of the Sampling-based Roadmap of Trees (SRT) algorithm using a decentralized master-client scheme. The distributed SRT algorithm allows us to solve very high-dimensional problems that cannot be efficiently addressed with existing planners. Our experiments show nearly linear speedups with eighty processors and indicate that similar speedups can be obtained with several hundred processors.

***Index Terms*— motion planning, roadmap, distributed algorithm, PRM, SRT.**

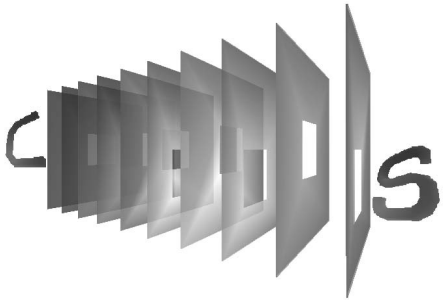


Fig. 1. A scene from our benchmarks. In problem "ConsR2," two robots must exchange places by going through ten small holes.

I. Introduction

Sampling-based planners have been used extensively in recent years for multiple query or single query motion planning [12]–[15], [17], [20]. In multiple query motion planning, a roadmap is built during a preprocessing phase in order to quickly respond to many queries. An example of such a planner is the Probabilistic Roadmap Method (PRM) [13]. Alternatively, in single query motion planning, there is no preprocessing phase and the configuration space is typically explored using a single or a bi-directional tree. Examples of such planners include Rapidly-exploring Random Trees (RRTs) [17] and Expansive Space Trees[1] (ESTs) [12].

High-dimensional problems such as those arising in planning with flexible objects [14], [16], [19], reconfigurable robots [21], coordination tasks [20], manipulation planning [19], and computational biology search problems [3], [4] test the limits of current motion planner implementations. Solving interesting problems for these complex robotic systems requires the development of better planners to reduce the time and the space used, which motivates our work [1], [6]. An important avenue is to effectively distribute computation in motion planning. This paper describes an efficient distributed motion planner that can be used to solve problems that are beyond the capabilities of current sequential planners.

Despite the need for fast solutions for high-dimensional problems, little effort has been devoted to the development of distributed motion planners, especially when contrasted with the work focused on sequential motion planners. The work in [18] gives a parallel algorithm for 6 degrees of freedom manipulators based on the property that the configuration space obstacle for a union of objects is the union of the configuration space obstacle of the individual objects. In [8], [9], a parallel version of the randomized path planner [5] is proposed that uses the OR paradigm, i.e., different processors compute the same algorithm and as soon as a solution is found, the computation stops. The work in [11] discretizes and then decomposes the configuration space into hypercubes and cyclically assigns the exploration of the hypercubes to the available processors. The method is impracticable for high-dimensional problems due to the discretization of the configuration space. The works in [2] and [7] focus on embarrassingly parallel algorithms for PRM and RRT, respectively. Embarrassingly parallel algorithms avoid any interprocess communication and in the context of PRM and RRT are limited to memory-shared systems.

In our earlier work [1], [6], the Sampling-based Roadmap of Trees[2] (SRT) planner was developed as a powerful motion planner that seamlessly integrates multiple query planners with single query planners and can be efficiently distributed. The distribution scheme, however, places a heavy computational burden on the master proces-

Work on this paper by E. Plaku and L. E. Kavraki has been supported in part by NSF 0205671, NSF 0308237, EIA-0216467 and a Sloan Fellowship to L. E. Kavraki.

[1]The acronym EST to describe Expansive Space Trees does not appear in the original papers, but is used in this paper for convenience.

[2]The name of the planner in [1], [6] is changed from Probabilistic Roadmap of Trees (PRT) to Sampling-based Roadmap of Trees (SRT) to emphasize the importance of sampling, which in turn, can be done in a variety of ways.

sor, which manages the distribution of computations among the client processors. As the number of client processors increases, it becomes difficult for the master to balance the computational load among the clients and thus it reduces the efficiency of the distribution.

This paper presents a novel distribution scheme for SRT that remains highly efficient for both memory-shared and message-passing systems even when the computation is distributed over hundreds of processors. The bottleneck of the distribution scheme in [1] is eliminated by introducing several master processors that cooperate with each-other to distribute the computation evenly among the client processors. The distributed SRT algorithm allows us to solve very high-dimensional problems that cannot be efficiently addressed with existing planners. This paper presents experiments with 126 degrees of freedom (DOF) where the computation of SRT is distributed over eighty processors. Figure 1 shows an example. We were able to obtain nearly linear speedups for the distributed computation of SRT. Our experiments with eighty processors indicate that similar speedups can be obtained with several hundred processors. Our results pave the way for the use of distributed SRT to planning problems of unprecedented complexity.

In section II, we briefly review the sequential SRT algorithm introduced in [1], [6]. Section III describes the distribution of the SRT algorithm. In section IV we describe the experimental setup, the set of benchmarks used to test the efficiency of our planner, and the results obtained. We conclude in section V with a discussion on the distributed SRT.

II. SRT Planner

In this section, we briefly review the sequential SRT algorithm [1], [6]. The pseudocode for SRT is given in Algorithm 1. SRT constructs a roadmap aimed at capturing the connectivity of the free configuration space. The nodes of the roadmap are not single configurations but trees, which are referred to as milestones. As illustrated in line 3 of Algorithm 1, each milestone is generated using a sampling-based tree planner, such as RRT [17] and EST [12]. Connections between milestones are computed in line 10 of Algorithm 1 by using bi-directional RRTs or ESTs. SRT can use the roadmap to answer multiple queries or stop the computation of the roadmap as soon as a solution to the single query at hand is obtained.

III. Distributed SRT

In this section, we describe the design and implementation of a new distributed version of SRT. A different and simplified version of a distributed SRT appears in [1]. Before relating the details, we discuss data and control flow dependency in each stage of the SRT algorithm. Milestone computations are independent of each-other; each milestone can be processed in parallel. Distributing the computation of a single milestone is considerably more involved due to the sampling scheme we use to generate milestones. Random edge selection can be done entirely 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 all the milestones. Finally, edge computations are not independent of each-other. Since milestones can change after an edge computation as a result of adding new configurations to the milestones and since computing an edge requires both milestones, the edge computations cannot be efficiently distributed without some effort. It requires milestones to be sent from one client to the other. Furthermore, computation pruning due to connected 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 computations.

Algorithm 1: Sampling-based Roadmap of Trees (SRT).

Input: K, number of milestones.
Output: A roadmap $G_T = (V_T, E_T)$.

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 rooted at a collision-free random config.
4: $V_T \leftarrow V_T \cup \{T\}$.
5: $Q \leftarrow Q \cup \{q_T\}$, where q_T is the representative of T.
6: **for all** $T \in V_T$ **do**
7: $S \leftarrow$ a set of n_c closest and n_r random $q_{T'} \in Q$ to q_T.
8: $E_C \leftarrow E_C \cup \{(T, T') : q_{T'} \in S\}$.
9: **for all** $(T_1, T_2) \in E_C$ **do**
10: **if not** *connected*(T_1, T_2) **and** *connect*(T_1, T_2) **then**
11: $E_T \leftarrow E_T \cup \{(T_1, T_2)\}$.

We have designed a master-client architecture for our distributed implementation of SRT. The clients are responsible for milestone and edge computations while the masters ensure that the load is distributed as evenly as possible among the clients. The masters arbitrate milestone ownership, edge selection, maintain the connected component data structure, and coordinate the activities of all the processors. A given set of processors $P = \{P_1, P_2, \ldots, P_p\}$ is partitioned into a set of master processors $M = \{M_1, M_2, \ldots, M_m\}$ and a set of client processors $C = \{C_1, C_2, \ldots, C_c\}$. Each client $C_i \in C$ is owned by some master $M_{C_i} \in M$. Consequently, each master $M_i \in M$ owns a set of clients $C_{M_i} \subseteq C$. Each master is responsible for only a fraction of the clients in order to ensure a timely response to their needs, since all the useful computation is done by the clients. This design was chosen as we expect to significantly increase the number of clients. In our implementation, each master owns c/m clients. The distributed SRT algorithm proceeds through several stages: milestone computations, candidate edges, and edge computations. The pseudocode of the distribution of the SRT planner is given in Algorithm 2.

A. Milestone Computations

The milestone computation stage is described in Algorithm 2 under Compute Milestones. During this stage, all the clients and masters M_2 through M_m compute

Algorithm 2: Distributed SRT.

Master		Client
1: $\diamondsuit$*Executed by* $M_1 \in M$	COMPUTE MILESTONES	1: $\diamondsuit$*Executed by all* $C_j \in C$ *and* $M_2, \ldots, M_m \in M$
2: $Q \leftarrow \emptyset$.		2: $T_{C_j} \leftarrow \emptyset$.
3: **for** $i = 1$ to K **do**		3: **while** `finish` has not been received **do**
4: Wait for some q_T to arrive; $Q \leftarrow Q \cup \{q_T\}$.		4: $T \leftarrow$ generate a milestone; $T_{C_j} \leftarrow T_{C_j} \cup \{T\}$.
5: Broadcast `finish`.		5: Send representative q_T to master M_1.
6: $G_C = (V_T, E_C) \leftarrow$ graph of candidate edges.		6: $\diamondsuit$*Executed by* $M_2, \ldots, M_m \in M$
7: Send G_C to all other masters.		7: Send milestones to clients to balance load.

1: $\diamondsuit$*Executed by all* $M_i \in M$	COMPUTE EDGES	1: $\diamondsuit$*Executed by all* $C_j \in C$
2: $W = C_{M_i} \leftarrow$ working clients.		2: **while** `finish` has not been received **do**
3: **while** unprocessed edges remain in G_C **do**		3: **if** send(T', C_i) is received **then**
4: **for** $C_j \in W$ **do**		4: Send copy of milestone T' to client C_i.
5: **if** $\exists e = (T', T'') \in E_C \cap T_{C_j} \times T_{C_j}$ **then**		5: **if** recv(T', C_i) is received **then**
6: Send e to C_j.		6: **if** milestones received > 5 **then**
7: $W \leftarrow W - \{C_j\}$.		7: Mark first received milestone T for deletion.
8: **else**		8: Send additions made to T to owner of T.
9: Find milestones $S = \{T_1, \ldots, T_5\}$ s.t. $\exists (T', T'') \in E_C \cap (T_{C_j} \cup S) \times (T_{C_j} \cup S)$.		9: Delete T.
10: Notify owner of each milestone $T \in S$ to send a copy of T to C_j.		10: Add T' to the list of milestones received.
11: **if** computed edges arrived from $C_h \in C$ **then**		11: **if** add$(T', C_i, c_1, \ldots, c_\ell)$ is received **then**
12: $W \leftarrow W \cup \{C_h\}$, if $C_h \in C_{M_i}$.		12: Add $c_1, \ldots, c_\ell$ to T'.
13: Update connected comps and G_C.		13: Update indices accordingly.
14: Broadcast `finish`.		14: **if** $e = (T', T'')$ is received **then**
		15: Connect T' and T''.
		16: Send result to all the masters.

milestones and send their representatives to master M_1 until a predefined total number of K milestones have been computed. Each set of milestones T_{C_i} computed by client C_i is stored locally in C_i while the set of the representatives is stored in M_1. The milestones computed by masters M_2 through M_m are sent to those clients which computed the smallest number of milestones in order to balance the load as much as possible. During the milestone computation stage, the communication is limited and non-blocking resulting in an efficient distribution of the computation load and little overhead.

B. Candidate Edges

During this stage, master M_1 computes the graph of candidate edges $G_C = (V_T, E_C)$ using the milestone representatives. Each milestone T is connected to its k-closest neighbors, where the distance between two milestones is defined as the distance between their representative configurations. In addition, milestone T is connected to a number of random neighbors to offset any problems with the metric used. The graph of candidate edges is sent to all the other masters M_2, …, M_m, which update their local copies of G_C. There is no distribution of this stage since it constitutes only a small amount of the total computation time and requires complex search data structures.

C. Edge Computations

The edge computation stage is described in Algorithm 2 under COMPUTE EDGES. The connections between milestones are computed by the clients while the masters decide which candidate edges should be connected.

Let $e = (T', T'')$ be the edge that master M_{C_i} sent to client C_i. If both milestones are currently owned by client C_i, then C_i runs the tree-connection algorithm on T' and T'' and if the connection is successful, it sends to all the masters the indices of two configurations $q' \in T'$ and $q'' \in T''$ that are connected by a local path. Otherwise, client C_i has to wait until it receives copies of the milestones that it does not own from their respective owners. During this time C_i could complete send operations that would help other clients to compute their assigned edges.

Upon receiving a computed edge, each master adds the edge to G_T and updates the graph of candidate edges. 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.

Initially, masters attempt to send to their assigned clients candidate edges whose milestones are locally stored, since the computation of such edges requires no communication with other clients. Once all the local candidate edges of a client C_i are computed, master M_{C_i} attempts to find 3–5 milestones, that when added to C_i create many local candidate edges. Copies of these milestones are sent to client C_i by their respective owners. Client C_i may have to delete copies of other milestones that it has received in previous steps to make room for the new milestones. Since edge computations usually add new configurations to the milestones involved, all the new configurations added to the milestones marked for deletion are sent back to their original owners, which in turn merge the additions with the existing configurations. Indices of configurations are updated accordingly.

Recall that clients receive candidate edges only from their respective masters. If client C_i receives an edge $e = (T', T'')$ it means that T' and T'' are owned by C_i or

that the respective owners of T' and T'' will send to C_i copies of these milestones. In both cases, T' and T'' will eventually be stored locally in C_i resulting in a deadlock-free design. Before deleting a local copy of milestone T, client C_i ensures that all the candidate edges already send to it by master M_{C_i} that involve T have been computed. Also note that milestones grow large as the result of edge connections and different merges that may occur throughout the computation process making it inefficient for clients to send these milestones to other clients. In our implementation, a client sends to other clients only a subset of the configurations of a large milestone.

IV. Experiments and Results

The experiments were chosen to evaluate the performance of the distributed SRT compared to the sequential implementation. In this paper, we use RRT as the sampling-based tree planner for SRT. The benchmarks presented in this paper are generally more difficult than those presented in [1], [6] in order to show the importance of an efficient distributed planner when solutions by traditional sequential planners, such as PRM, EST, RRT, or even newer and more powerful sequential planners, such as SRT, cannot be obtained in a reasonable amount of time.

A. Benchmarks

Figure 2 illustrates the environments and the robots that we used to create the benchmarks.

Benchmark "ConsR1" consists of ten consecutive walls each with a small hole, as shown in Figure 2(b). The robot is an object in the shape of the letter "C," as shown in Figure 2(a), which must move through all the ten holes. The dimensions and relative positioning of the holes are such that the robot is forced to wiggle its way through. Benchmark "ConsR2" is a similar problem with two robots in the shape of the letters "C" and "S," respectively, as shown in Figure 1. Benchmarks "ConsR1" and "ConsR2" have 6 and 12 DOFs, repsectively.

Benchmark "BunnyR1" consists of a fence, a wall with a cross-like hole, and another fence placed consecutively near each-other, as shown in Figure 2(c). The robot is an object in the shape of a bunny, as shown in Figure 2(a), consisting of 8171 vertices and 16301 triangles. The motion planner finds collision-free paths that move the robot through the openings in the walls. The openings in the fence are as large as necessary to allow the robot to go through. Benchmark "BunnyR2" is similar to benchmark "BunnyR1," but with two bunny-like robots instead of one. Similarly, benchmark "BunnyR8" consists of eight bunny-like robots placed inside box that has a wall with a small cross-like hole in the middle, as shown in Figure 2(d). Benchmarks "BunnyR1," "BunnyR2," and "BunnyR8" have 6, 12, and 48 DOFs, repsectively.

Benchmark "LettersR21" consists of a box that has a wall with a circular hole in the middle inside which a circular ring is placed, as shown in Figure 2(e). There are twenty-one robots in the shape of the letters "A, B, C, ..., U," respectively, some of which are illustrated in Figure 2(a). The robots are placed in a grid-like format on one side of the wall and the objective of the motion planner is to move the robots through the openings in the wall to the other side and position the robots in the same grid-like format. Benchmark "Letters21" has 126 DOFs.

The environment of benchmark "MazeR1" consists of a 3D-maze object, as shown in Figure 2(f). The robot is a single object in the shape of a cylinder bent several times, as shown in Figure 2(a), which should move from the lower left corner to the upper right corner of the maze. The dimensions of the cells and of the robot are such that the robot must wiggle its way through. Benchmark "MazeR1" hase 6 DOFs.

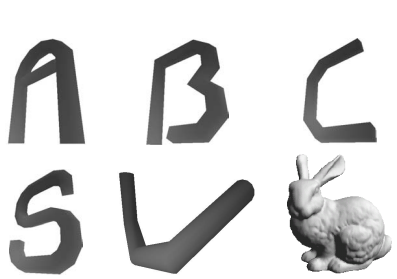
B. Hardware and Software Setup

The implementation of the distributed SRT algorithm was carried out in ANSI C/C++ using the Intel® 8.0 compilers and libraries. Additionally, we made use of the SWIFT++ collision detection library [10], the MPICH implementation of MPI standard for communication and OpenGL for visualization. The experiments were run on the Rice Terascale Cluster, a 1 TeraFLOP Linux cluster based on Intel® Itanium®2 processors. Each node has two 64-bit processors running at 900 MHz, with 1.5 MB of L2 data cache and 2 GB memory per processor. The nodes are connected by a Gigabit Ethernet network. For the experiments, we used only one processor per node.

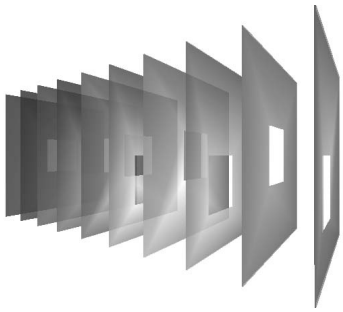
C. Efficiency of Distributed SRT

To measure the efficiency of the distributed SRT algorithm, we ran the distributed code on various benchmarks using 1, 4, 8, 16, 24, 32, 48, 64, and 80 client processors and 1, 2, and 3 master processors. Table I contains a summary of the results. For each benchmark, we report the computation time required by the sequential version of SRT (time[1]) and the distributed efficiency of SRT with 80 clients (efficiency[80 clients]) and n masters, where $n = 1,2,3$. The distributed efficiency is calculated as $t_s/(t_d \cdot N)$, where t_s is sequential time, t_d is distributed time, and N is the number of processors. As an example, referring to Table I, benchmark "ConsR2" requires approximately 20hrs of computation time by the sequential SRT. Ideally, when 81 processors are used the running time would be 15mins. When the distributed SRT is used with 80 clients and one master the running time is only 17mins which results in an efficiency of 0.89. Using 82 processors reduces the ideal running time to 14.9mins. When the number of masters is increased to two, the computation time of the distributed SRT is reduced to 16mins resulting in an efficiency of 0.92. Finally, using 83 processors results in an ideal time of 14.7mins. Increasing the number of masters to three keeps the same computation time of 16mins resulting in a slightly reduced efficiency of 0.91.

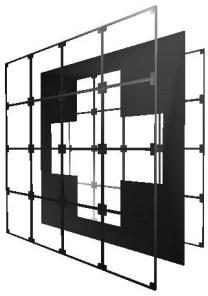
The overall efficiency of the distributed SRT algorithm is reasonably high on all our benchmarks. When only one master is used, the efficiency of the distributed SRT ranges from 0.58 to 0.89 with an average of 0.73 and



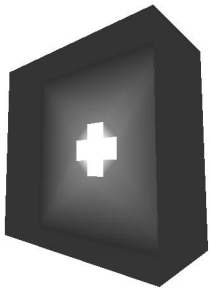
(a) Letters “A, B, C, S,” bent cylinder, and bunny.



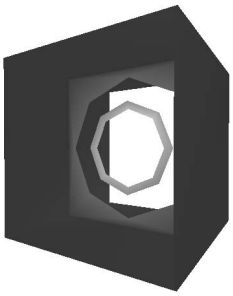
(b) “ConsR1” and “ConsR2.”



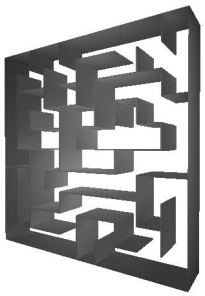
(c) “BunnyR1” and “BunnyR2.”



(d) “BunnyR8.”



(e) “LettersR21.”



(f) “MazeR1.”

Fig. 2. Path planning benchmarks: (a) robots, (b) – (f) scenes.

TABLE I
EFFICIENCY OF DISTRIBUTED SRT.

benchmark	time[1](s)	efficiency[80 clients]		
		1 master	2 masters	3 masters
ConsR1	19635.18	0.89	**0.93**	0.88
ConsR2	73388.19	0.89	**0.92**	0.91
BunnyR1	4702.47	0.69	0.85	**0.94**
BunnyR2	14355.57	0.74	**0.97**	0.88
BunnyR8	4437.82	0.71	0.84	**0.85**
LettersR21	5275.28	0.58	0.62	**0.65**
MazeR1	6572.53	0.63	0.72	**0.94**

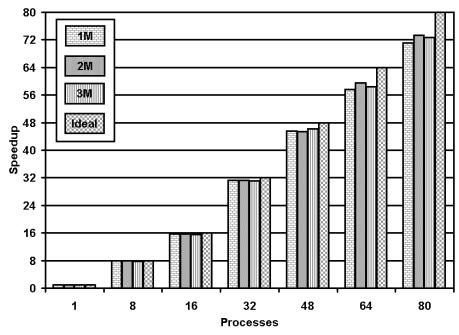


Fig. 3. Speedup of distributed SRT.

median of 0.71. When the number of masters is increased to two, the efficiency ranges from 0.62 to 0.97 with an average of 0.84 and median of 0.85. Increasing the number of masters to three does not change the efficiency significantly; it ranges from 0.65 to 0.94 with an average of 0.86 and median of 0.88. As illustrated in Table I, increasing the number of masters from one to two resulted in increased efficiency of our distributed algorithm for all the benchmarks. The largest increase, 31.08%, is obtained for benchmark “BunnyR2” and the lowest increase, 3.37%, is obtained for benchmark “ConsR2.” When only a single master is used, the load of the master increases proportionally to the number of the clients. As the number of clients becomes large, a single master is not able to handle their requests. Increasing the number of masters to two or three allows for a better distribution of the workload, and consequently, higher efficiency. This phenomenon is clearly seen in the “MazeR1” benchmark where the efficiency of the distributed SRT with 80 clients increased from 0.63 to 0.94, an increase of 49.20%, when the number of masters is changed from one to three.

Figure 3 compares the ideal speedup to the speedup obtained for the benchmark “ConsR2” when the distributed code is run with one, two, and three masters and up to 80 processors. Figure 4 presents logged data for the benchmark “ConsR1,” showing where clients spend their time, i.e., milestone computation, edge computation, or communication. The plots in Figures 3 and 4 are characteristic of the behavior of the distributed SRT on the other benchmarks as well. Figure 3 indicates a nearly linear speedup for the distributed SRT when two or three masters are used. As expected, the speedup is worse, but only slightly, when one master is used. Figure 4 indicates that virtually all of the overhead occurs during the last stages of edge computations. At this point, only a few edges have not been computed (fewer than the number of clients

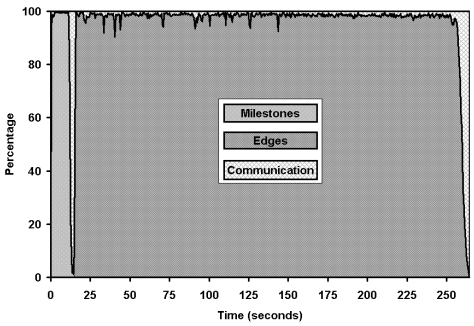


Fig. 4. Time distribution of SRT.

available) and consequently many clients starve wasting useful computation time. Nevertheless, milestone and edge computations are nearly fully distributed and storage is also distributed evenly.

Extrapolating from the results of our experiments, we suspect that similar speedups and computation distributions can be obtained even when the number of clients is doubled or tripled and the number of masters remains the same. We still have not reached the point were masters become the bottleneck, but 80 was the largest number of processors we had available.

V. Discussion

High-dimensional problems arising from complex robotic systems require the development of powerful sequential motion planners and the development of efficient distributed motion planners that take full advantage of all the available resources. This paper presents an efficient algorithm for evenly distributing the computation of SRT, allowing us to solve difficult problems with up to 126 DOF in few minutes. The distributed SRT planner provides a platform for solving problems of high complexity that cannot be solved in a reasonable amount of time even by the most efficient sequential planners.

We believe that the efficiency of the distributed algorithm derives in part from the hierarchical nature of SRT and the sharing of the scheduler's load among the different masters. Prior work on the distribution of SRT [1], [6] and the experiments with one master indicate that the master becomes the bottleneck as the number of clients increases, since it is unable to handle the large number of requests efficiently. Increasing the number of masters allows for an even distribution of the workload and, consequently, a higher efficiency. Using two or three masters, we were able to obtain nearly linear speedups when running on 80 processors – the largest number we had available. We believe that our master-client architecture can easily support several hundred processors and still yield nearly linear speedups.

Acknowledgment

The authors would like to thank all the members of the Physical and Biological Computing group at Rice University for their helpful comments and discussions.

References

[1] M. Akinc, K. E. Bekris, B. Y. Chen, A. M. Ladd, E. Plaku, and L. E. Kavraki, "Probabilistic roadmaps of trees for parallel computation of multiple query roadmaps," in *Interantional Symposium on Robotics Research*, ser. Springer Tracts in Advanced Robotics (STAR), D. Paolo and R. Chatila, Eds. Springer Verlag, 2003.

[2] N. M. Amato and L. Dale, "Proabilistic roadmap methods are embarrasingly parallel," in *IEEE International Conference on Robotics and Automation*, 1999, pp. 688–694.

[3] N. M. Amato, K. Dill, and G. Song, "Using motion planning to map protein folding landscapes and analyze folding kinetics of known native structures," in *International Conference on Research in Computational Molecular Biology*, April 2002, pp. 2–11.

[4] M. S. Apaydin, D. L. Brutlag, C. Guestrin, D. Hsu, and J.-C. Latombe, "Stochastic roadmap simulation: An efficient representation and algorithm for analyzing molecular motion," in *International Conference on Research in Computational Molecular Biology*, April 2002, pp. 12–21.

[5] J. Barraquand and J.-C. Latombe, "Robot motion planning: A distributed representation approach," *International Journal of Robotics Research*, vol. 10, no. 6, pp. 628–649, December 1991.

[6] K. E. Bekris, B. Y. Chen, A. M. Ladd, E. Plaku, and L. E. Kavraki, "Multiple query motion planning using single query primitives," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2003, pp. 656–661.

[7] S. Carpin and E. Pagello, "On parallel RRTs for multi-robot systems," in *8th conference of the Italian Association for Artificial Intelligence*, 2002, pp. 834–841.

[8] D. J. Challou, D. Boley, M. L. Gini, and V. Kumar, "A parallel formulation of informed randomized search for robot motion planning problems," in *IEEE International Conference on Robotics and Automation*, 1995, pp. 709–714.

[9] D. J. Challou, M. L. Gini, and V. Kumar, "Parallel search algorithms for robot motion planning," in *IEEE International Conference on Robotics and Automation*, 1993, pp. 46–51.

[10] S. A. Ehmann and M. C. Lin, "Geometric algorithims: Accurate and fast proximity queries between polyhedra using convex surface decomposition," *Computer Graphics Forum - Proceedings of Eurographics*, vol. 20, pp. 500–510, 2001.

[11] D. Henrich, C. Wurll, and H. Wörn, "Multi-directional search with goal switching for robot path planning," in *International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 1998, pp. 75–84.

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

[13] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, June 1996.

[14] A. M. Ladd and L. E. Kavraki, "Using motion planning for knot untangling," *International Journal of Robotics Research*, vol. 23, no. 7-8, pp. 797–808, 2004.

[15] ——, "Fast exploration for robots with dynamics," in *Workshop on Algorithmic Foundations of Robotics*, 2004.

[16] F. Lamiraux and L. E. Kavraki, "Planning paths for elastic objects under manipulation constraints," *International Journal of Robotics Research*, vol. 20, no. 3, pp. 188–208, 2001.

[17] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, May 2001.

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

[19] M. Moll and L. E. Kavraki, "Path planning for minimal energy curves of constant length," in *IEEE International Conference on Robotics and Automation*, 2004, pp. 2826–2831.

[20] G. Sánchez and J.-C. Latombe, "On delaying collision checking in PRM planning: Application to multi-robot coordination," *International Journal of Robotics Research*, vol. 21, no. 1, pp. 5–26, 2002.

[21] M. Yim, D. G. Duff, and K. D. Roufas, "PolyBot: a modular reconfigurable robot," *IEEE International Conference on Robotics and Automation*, pp. 514–520, 2000.