A Parallel Formulation of Informed Randomized Search for Robot Motion Planning Problems[*]

Daniel Challou Daniel Boley Maria Gini Vipin Kumar
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

Abstract

We show how paths for articulated robots with many degrees of freedom can be generated in a few seconds or less using nonsystematic parallel search. We present experimental results obtained on a multicomputer for an accurate model of a 7-jointed manipulator arm operating in realistic 3D workspaces. We then present and discuss a fast method for smoothing the paths delivered by the parallel algorithm.

1 Introduction

Among the many skills robots require is the ability to plan the paths they must take while executing their tasks. The intent of this paper is to provide experimental evidence that very fast path planning can be performed using parallel algorithms. By very fast we mean systems that produce solutions in fractions of seconds or a few seconds for complex environments and robots with over four degrees of freedom (*dof*).

We have developed a parallel formulation of the Randomized Path Planner proposed by Barraquand and Latombe [2]. Our parallel formulation has proved extremely effective in solving variants of the robot motion planning problem that are computationally impractical on sequential machines [5]. Our formulation has yielded excellent performance gains on virtually every problem we have tried, sometimes delivering even superlinear speedup [6].

From networks of workstations to parallel processors on a single chip, a wide variety of cheap parallel platforms will be available in the near future, making our approach a viable option.

[*]This work was supported in part by Contract DAAL03-89-C-0038 between the Army Research Office and the University of Minnesota for the Army High Performance Computing Research Center, NSF/CDA-9222922, NSF/CCR-9405380, IST/SDIO 28408-MA-SDI, and the Center for Advanced Manufacturing, Design and Control of the University of Minnesota.

2 Background and Related Work

Many algorithms have been developed for motion planning [12], but most are never used in practice because of their computational complexity [8]). Since the complexity increases with the number of *dof*, there are very few algorithms capable of generating paths in reasonable time frames (i.e., times on the order of minutes) for robots with more than three *dof*.

Most motion planning methods decompose the search space into discrete components called cells. The motion planning problem then becomes one of computing a decomposition and searching through sequences of contiguous cells to find a a sequence of configurations that involves no collisions with obstacles.

To obtain acceptable performance, some methods do a significant amount of preprocessing of the configurations space (C-space) [9], or place landmarks in C-space that are then used by a local planner [3, 7]. Other methods make assumptions on the type of robot (for instance, [1] takes advantage of the symmetry of the workspace with respect to the first axis of the robot), or use a coarse discretization of C-Space. Real time has been achieved in detection of imminent collisions [15] but not for path planning.

In an effort to decrease the computation time, some researchers have devised parallel methods. Lozano-Perez [13] was the first to develop a parallel algorithm to compute the discretized C-space for the first three links of a six *dof* manipulator. This method works well, but it is limited to relatively coarse C-space discretizations due to memory limitations. A genetic based approach has been implemented using 128 T800 transputers with impressive performance [3]. The method keeps placing landmarks in free space until it is able to generate a path using local methods.

With our parallel formulation of the algorithm proposed in [2] we are able to solve complex motion planning problems in a few seconds or less.

3 Search and Parallel Search

Langley [11] classified search methods as *systematic* and *nonsystematic*. Systematic search methods enumerate each node in the search space. They produce each possible path only once, thus minimizing the redundant work.

Nonsystematic (or randomized) search methods select a node at random at each choice point in the search and record it as a step in the path. This process is repeated until a solution is found or a depth limit is reached. These methods do not retain a complete memory of nodes previously visited, and thus may generate the same paths more than once. They are only probabilistically complete (and thus cannot report that a problem has no solution with perfect certainty), but are capable of outperforming uninformed systematic schemes. Nonsystematic search is particularly effective on tasks whose solutions are many and deep, and though purely randomized search methods can be successful, they can often benefit from heuristic knowledge [11].

Two types of parallel formulations of search methods have been developed: (1) communication-based formulations that partition the search space among the processors, and (2) formulations in which each processor explores the entire search space randomly with no interprocessor communication. In both formulations the first processor to find a solution sends a termination signal to the remaining processors, and then reports its solution.

We can give a brief theoretical explanation for the success of parallel randomized allocation schemes. Let $P_1(t)$ be the probability that a single processor will find a solution within time t, and let $P_k(t)$ be the probability that a k-processor parallel randomized allocation scheme will find a solution within time t. Let the random variable T_1^i be the time it would take processor p_i to find a solution, if allowed to run to completion. Since this is equivalent to running multiple trials on a single processor, the T_1^i's are independent and identically distributed. The probability $1 - P_k(t)$ that the solution time on k processors will exceed t is just the probability that none of the k processors will find a solution within time t:

$$1 - P_k(t) = (1 - P_1(t))^k \tag{1}$$

To interpret this formula, suppose a single processor has only a 10% probability that it can solve the problem within a given time $t_{10\%}$. Then a 32 processor system has over a 96% probability of finding a solution within $t_{10\%}$. A 64 processor system has over a 99% chance of doing the same. Space does not permit a more detailed analysis, which can be found in [4].

We define $E[T_1]$ as the average uniprocessor solution time; $E[T_k]$ as the average k processor solution time, and speedup as:

$$S = \frac{E[T_1]}{E[T_k]}. \tag{2}$$

If $E[T_k]$ is less than $E[T_1]$, then, on average, the k processor randomized allocation formulation will deliver speedup over the uniprocessor algorithm. If $E[T_k]$ is less than $\frac{1}{k} \cdot E[T_1]$ then, on k processors, the parallel randomized allocation formulation will yield, on average, superlinear speedup over the uniprocessor algorithm. This is because on k processors the first processor to find a solution stops all the others, so there is no need to wait for solutions that take a long time. On a single trial on a uniprocessor a bad choice made early might significantly delay the completion of the search. However, on k processors a bad choice made by one processor does not prevent the other processors from making a better choice.

4 Parallel Motion Planning

We have developed a parallel formulation of the Randomized Path Planner [2] and implemented it on a variety of architectures including the nCUBE2[1] (with up to 1024 processors), the CM-5[2], and a network of workstations.

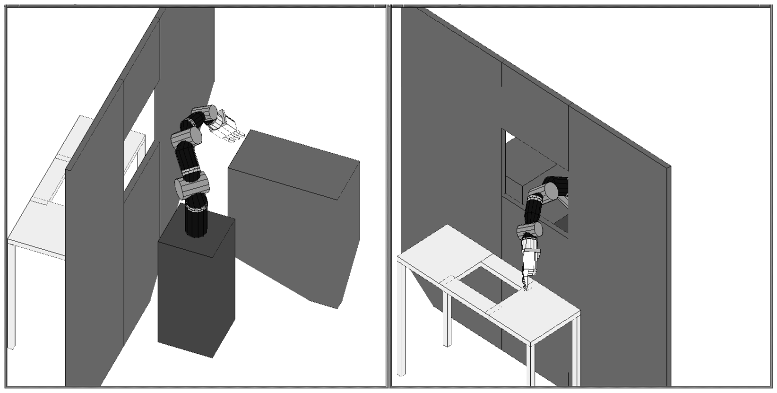
An outline of the algorithm is shown below. The "*"'s indicate points where the algorithm checks for a termination message from other processors. Such a termination message terminates the computation.

Step I: Compute the heuristics used to guide the search.
Step II: Search using the heuristics :

```
Gradient Descent until local minimum *
repeat until goal reached or global time-out
    repeat K times or until improvement found
        Random Walk to escape local minimum *
        Gradient Descent until a local minimum *
    if no improvement
        then Randomly Backtrack
    if improvement found
        then append new path to previous path
if goal found
    then broadcast termination message
```

[1]registered trademark of the nCUBE Corporation

[2]registered trademark of the Thinking Machines Corporation. The results obtained on the CM-5 presented here are based upon a beta version of the software and, consequently, are not necessarily representative of the performance of the full version of the software.



No. Processors	1	32	64	128	256
Avg Search Time	57.28	7.05	4.69	3.50	2.55
Std Dev	59.96	3.96	2.35	1.48	0.91
Avg Path Length	11892	6335	4638	3543	2412
Std Dev	4925	3635	2725	1905	1141
Avg Speedup	1.00	8.12	12.21	16.37	22.46

Figure 1: Start and Goal Configurations for a 7 *dof* arm operating in a 128^3 cell workspace. The robot is reaching from the dark table behind it, through on opening in the wall on his left, to the light table with the hole on it. The table shows data for at least 64 runs on a CM-5 multicomputer. All times are in seconds.

Consider the search starting at the start robot configuration. The gradient descent forces the search in the direction of the goal node that appears closest. If the heuristic is misleading then, at some point, every successor is worse than the current node. When this occurs, random search with a randomly chosen depth bound is executed. We call this step a random walk.

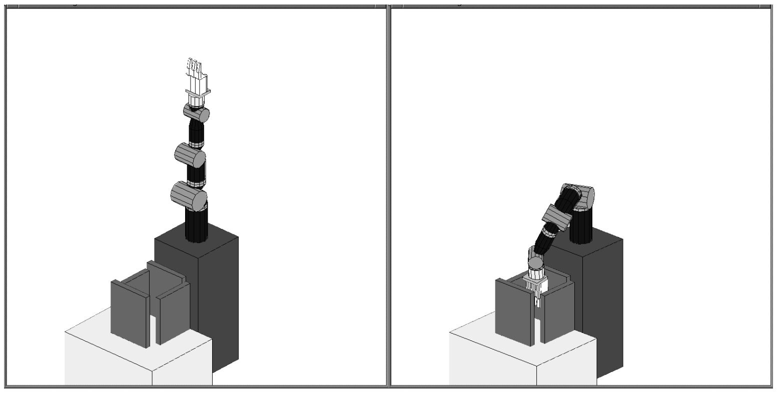
Gradient descent resumes from the state at which the random walk terminates. The sequences composed of a random walk followed by gradient descent search are repeated for a predetermined number (K) of trials or until a better node is found. If, after K trials, no better node has been found, then backtracking is performed to a randomly picked point in the current path. The cycle of random walks followed by gradient descent is then resumed. When a better node is found the new part of the path found is appended to the previous path and the process resumes.

The idea behind the random walks and randomized backtracking is to find a place in a different region of the search space where the heuristic is more reliable. In that event the gradient descent search can quickly descend toward a goal configuration. Successors of a node are generated in a random manner until a successor is found that has a better heuristic value than the current configuration. Thus, the first legal successor with a better value than its parent is adopted as the next step in the path.

The randomization in the state generation process, random walks, and randomized backtracking are the means by which the search space is allocated to each processor. Within each processor, the randomized search is controlled by a random number generator with an initial seed guaranteed to be unique among all the processors. Each processor is assigned a disjoint range of initial seed values, and a specific value within this range is selected using the processor clock. This probabilistically ensures each processor searches a different part of the search space.

The performance of this scheme varies significantly from run to run. For many problems the method delivers a solution in a few seconds or minutes on one run, but on other runs no solution is delivered for minutes, hours, or even days [4]. This is because on some runs the random walks help escape from dead ends in the search space more effectively than on other runs.

There are many reasons for our selection of the algorithm to parallelize. First, parallel formulations of randomized search can easily be developed using randomized allocation schemes. Second, the grid-based representation of the workspace is especially convenient when sensors are used to construct it, as shown in [14], [10].



No. Processors	1	32	64	128	256
Avg Search Time	83.02	2.23	1.28	0.81	0.62
Std Dev	104.13	2.41	1.39	0.53	0.14
Avg Path Length	6211	896	583	302	235
Std Dev	6049	1871	1322	187	134
Avg Speedup	1.00	37.23	64.86	102.49	133.90

Figure 2: Start and Goal Configurations for a 7 *dof* arm operating in a 128^3 cell workspace. The robot is reaching down into the small box in front of it. The table shows data for at least 64 runs on a CM-5 multicomputer. All times are in seconds.

5 Discussion of Results

This section presents results for an accurate scale model of a 7-jointed Robotics Research arm. In these examples, we discretize the C-space into 128^7 cells, so that each joint has 128 discrete positions, each $2.8125° = 360°/128$ apart. The workspace is discretized into a $128 \times 128 \times 128$ array of cells. Each cell in the workspace represents a volume of 2.1 cm^3.

Figure 1 shows the start and goal configurations for a test case. The table indicates the benefits of parallelizing the planner. For this problem just 32 processors cut the average solution an order of magnitude to under ten seconds, and 128 processors cut the average solution time to under five seconds. For the problem shown in Figure 2, 128 processors cut the average solution time to under a second.

In addition to delivering paths in shorter time frames, the parallel algorithm produces better solutions when executed with a larger number of processors by generating shorter paths. In the example in Figure 1, 32 processors yield a solution path length about one half as long as the average solution path length delivered by one processor. In the example in Figure 2, 64 processors are sufficient to reduce the length more than an order of magnitude.

The variance in time to solution behaves similarly, that is, it falls off as the number of processors increases.

As the number of processors increases, we reach a point where the performance falls off and the average time to solve the problem moves toward a constant value. This is because the probability that the random component of the algorithm will ensure that different processors explore different parts of the search space decreases with the number of processors.

An interesting trend supported by the data in Figure 1 is that a significant gain in performance is delivered by a reasonably small number of processors. Thus, the following question keeps recurring: "How many processors does the method need to deliver acceptable performance?"

The following two quantities are of particular interest: (1) the time needed by the parallel algorithm to solve a particular problem given a fixed number of processors, and (2) the number of processors needed to deliver a solution within a given time bound.

Recently we developed a fast performance prediction method that yields an accurate estimate of both quantities using a small base of solution runs obtained with a few processors [4]. Such a method enables a user to quickly ascertain how many processors are necessary to obtain a desired level of performance on a particular problem.

6 Smoothing

In this section we discuss a smoothing technique to be applied to the paths produced by the parallel randomized planner. To aid in the illustration of this technique, we use an example of a redundant manipulator in 2D. Figure 3 shows the starting robot configuration for our example with dashed line segments, the final configuration with solid line segments, the path from the parallel randomized planner with dots, and the smoothed path with a solid curve. The remaining rectangles are obstacles. In this example, we discretize the C-space into 128^6 cells and the workspace into 128^2 cells.

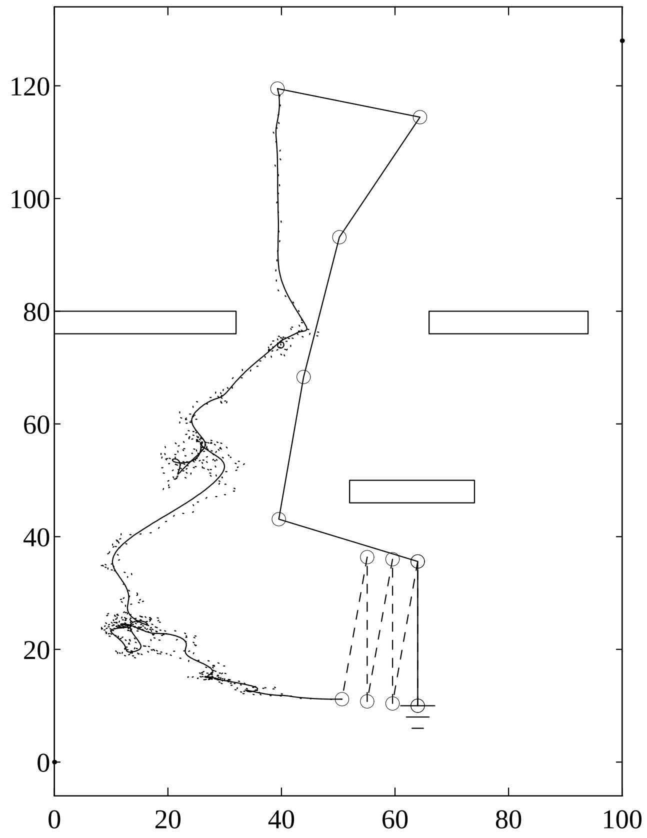


Figure 3: 2D example with 6 *dof* showing raw path (dotted curve) and smoothed path (solid curve).

In order to ensure that progress is made at every search step, the parallel randomized planner tends to move every joint at every step. At the next search step, a joint may end up moving back to its previous position, where each such move is exactly 2.8125°. As a result, in the path delivered by the parallel planner joints can "jitter" between two or three neighboring positions and the path is not very practical. Hence there is a need to smooth the raw paths produced by the parallel randomized planner.

Since the raw paths can easily have several thousand points, we use a fast one-pass smoothing technique based on a gaussian kernel. The raw paths are represented by sequences of joint angles, one for each joint. The smoothing kernel is applied to each sequence of joint angles independently. Each joint angle is replaced with a weighted average of all its neighboring joint angles in the same sequence, using gaussian weights.

Specifically, if $\theta_j^{(k)}$ denotes the j-th joint angle in the sequence for the k-th joint and n denotes the length of the path, then the smoothed joint angle $\widetilde{\theta}_j^{(k)}$ is computed by

$$\widetilde{\theta}_j^{(k)} = \sum_{-s \leq i \leq +s} \theta_{j+i} \cdot \frac{1}{\sqrt{s\pi}} e^{-i^2/s}, \text{ for } j = 0, 1, 2, \ldots, n$$

where s is a user-defined parameter specifying the smoothing window size.

Figure 4 shows the typical result using joint 3. The dots show the raw sequence of joint angles produced by the parallel randomized planner, and the solid curve shows the smoothed sequence of joint angles using $s = 255$. With this value of s, only the neighboring 83 joint angles have relative weights above 10^{-3}, hence an effective window size of 83 is more than sufficient to smooth the joint angles with this value of s.

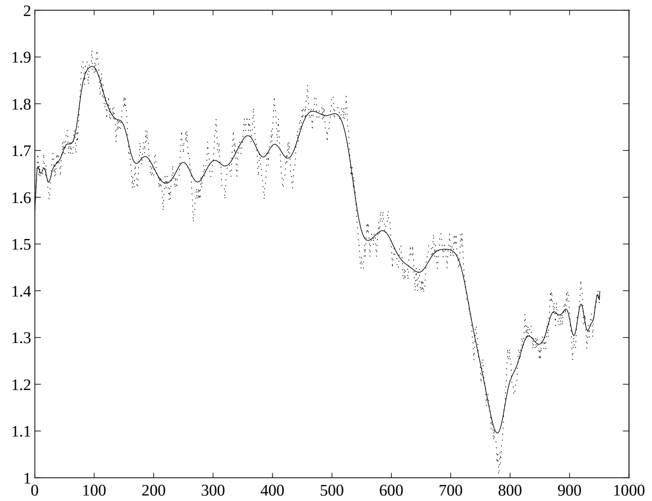


Figure 4: Joint angles for joint 3 for Figure 3: raw (dotted curve) and smoothed (solid curve).

Of course, the smoothed path does not match the raw path from the parallel randomized planner, and only the raw path is guaranteed to be free of collisions. We propose the following paradigm to ensure that the smoothed path is free of collisions. We incorporate a *safety margin* into the parallel randomized planner, so that the manipulator will keep away from every obstacle by at least this safety margin in the raw path. We then dynamically adjust the window size s to keep the discrepancy between the smoothed path and the raw path below this safety margin. One must compute the discrepancy in the position of every link of the manipulator resulting from the raw path and the smoothed path, but it suffices to compute this discrepancy just for the joints. Hence this is a fast computation. By adjusting the window size s during the smoothing process, one can keep the discrepancy bounded while producing a smooth path for the robot to follow.

Figures 3 and 4 show the results when using a fixed window size $s = 255$, with reduced values near the ends to enforce the true start and goal positions. The resulting paths had 951 points each, and the maximum discrepancy for any joint through the entire motion was 6.96, well under the initial safety margin of 8.

7 Conclusions and Future Work

In summary, we have devised and implemented a parallel motion planning algorithm based on a combination of informed and randomized search. The method delivers excellent performance on realistic problems using robots with many *dof*. Coarser discretizations and the use of currently available processors faster than those used here would enable our system to deliver sub-second performance even with a modest number of processors [4].

The ability to plan paths very rapidly will open up new perspectives for path planning and will make it attractive for many application areas, such as industrial robotics, teleoperation, control of redundant robots. Real-time path planning coupled with real-time sensing will allow robots to adapt their planned paths to take into account the uncertainties of the real world. Even more important, this will allow robots to react to unanticipated events and to quickly replan their paths whenever needed.

8 Acknowledgements

We would like to sincerely acknowledge Mike Hennessey and Max Donath for helping us model the Robotics Research arm; Jean Claude Latombe at Stanford University for providing access to implementations of the Random Path Planner; David Strip and Robert Benner at Sandia National Laboratories for providing access to the nCUBE2.

References

[1] P. Adolphs and H. Tolle. Collision-free real-time path-planning in time varying environment. In *Proc. IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems*, pages 445–452, 1992.

[2] J. Barraquand and J. C. Latombe. Robot motion planning: A distributed representation approach. *Int'l J. of Robotics Research*, 10(6):628–649, 1991.

[3] P. Bessiere, J.-M. Ahuactzin, E.-G. Talbi, and E. Mazer. The Ariadne's Clew algorithm: Global planning with local methods. In *Proc. IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems*, 1993.

[4] D. Challou. *Parallel Search Algorithms for Robot Motion Planning.* PhD thesis, Univ. of Minnesota, 1995.

[5] D. Challou, M. Gini, and V. Kumar. Parallel search algorithms for robot motion planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, volume 2, pages 46–51, 1993.

[6] D. Challou, M. Gini, and V. Kumar. Toward real-time motion planning. In H. Kitano, V. Kumar, and C. B. Suttner, editors, *Parallel Processing for Artificial Intelligence, 2.* Elsevier, 1994.

[7] P. C. Chen and Y. K. Hwang. SANDROS: a motion planner with performance proportional to task difficulty. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 2346–2353, 1992.

[8] Y. Hwang and N. Ahuja. Gross motion planning – a survey. *ACM Computing Surveys*, 24(3):219–291, 1992.

[9] L. Kavraki. Randomized preprocessing of C-space for fast path planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 2138–2145, 1994.

[10] T. Laliberte and C. Gosselin. Efficient algorithms for the trajectory planning of redundant manipulators with obstacle avoidance. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 2044–2049, 1994.

[11] P. Langley. Systematic and nonsystematic search strategies. In *Proc. Int'l Conf. on AI Planning Systems*, pages 145–152, College Park, Md, 1992.

[12] J. C. Latombe. *Robot Motion Planning.* Kluwer Academic Publ., Norwell, MA, 1991.

[13] T. Lozano-Perez. Parallel robot motion planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 1000–1007, 1991.

[14] H. P. Moravec. Sensor fusion in certainty grids for mobile robots. *AI Magazine*, 9(2):61–74, 1988.

[15] T. S. Wikman, M. Branicky, and W. S. Newman. Reflexive collision avoidance: a generalized approach. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, volume 3, pages 31–36, 1993.