Proc. 1995 IEEE International Symposium on Assembly and Task Planning, pp 201–206

Very Fast Motion Planning for Dexterous Robots*

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

Abstract

We show that paths for dexterous robots can be generated in a few seconds or less using parallel informed randomized search on multicomputers. The experimental results we present have been obtained for a simulated 7-jointed arm operating in realistic 3D workspaces. We also present a new method for predicting the solution times that our parallel formulation can deliver on increasing numbers of processors.

1 Introduction

Motion planning is the process of computing paths that will allow a robot to move to different positions in its environment without hitting obstacles. Many motion planning algorithms have been developed [13], but most are never used in practice because of their computational complexity [8].

The intent of this paper is to show that plans for multi-jointed dexterous robot arms which operate in realistic environments can be synthesized very quickly by parallel algorithms, and to show how to estimate the number of processors necessary to obtain results in an acceptable time frame.

Figure 1 shows a simulated model of a Robotics Research K-1207i 7-degree of freedom (*dof*) arm. We expect to execute our planned paths on the real robot shortly.

The ability to plan paths quickly is important to make motion planning useful in application areas, such as industrial robotics, teleoperation [15], control of redundant robots [11]. Real-time motion 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, it will allow robots to react to unanticipated events and to quickly replan their paths whenever needed.

*This work was supported in part by Contract numbers DA/DAAH04-93-G-0080 and DA/DAAH04-93-G-0131 between the Army Research Office and the University of Minnesota for the Army High Performance Computing Research Center. Additional support was furnished by NSF/CDA-9022509, NSF/CCR-9405380, and the Center for Advanced Manufacturing, Design and Control of the University of Minnesota.

2 Motion Planning

Research in the area of robot motion planning can be traced back to the late sixties, but most of the work has been carried out more recently. Over the last few years the theoretical and practical understanding of the issues has increased rapidly, and a variety of solutions have been proposed. Latombe [13] provides an extensive overview.

There are two different spaces associated with motion planning algorithms, the workspace and the configuration space (C-space). The workspace is the world that the robot must move through, the C-space is the set of all robot configurations. The dimensionality of the C-Space is the number of parameters required to fully specify a configuration of the robot. The search for a path is performed in C-space because the robot becomes a point in C-space. The motion planning problem then becomes one of computing a decomposition of the C-space and searching through sequences of cells to find a path that involves no collisions with obstacles.

For example, assume we have a robot arm with k joints, where each degree of freedom (joint) is quantized into n discrete levels. Such an arm has a C-space consisting of n^k unique configurations. Now assume that the planner allows each joint to be in any one of 90 unique positions, so $n = 90$. Thus, an arm with four joints ($k = 4$) has over sixty five million configurations, and an arm with six joints ($k = 6$) has over five hundred billion configurations. Consequently, even if it were possible to compute all of the C-space, the amount of primary memory required to store it would be prohibitive.

To obtain acceptable performance, some methods do a significant amount of preprocessing of the configuration space (C-space) [10], or place landmarks in C-space that are then used by a local planner [3, 6]. 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 [18, 19], but not for path planning.

In an effort to decrease the computation time, some researchers have devised parallel methods. Lozano-Perez [14] 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 excellent performance [3]. The method places landmarks in free space until it is able to generate a path using local methods.

Our method is a parallel formulation of the Randomized Path Planner proposed by Barraquand and Latombe [2]. Space is represented with bitmap arrays. The configuration space is discretized and searched using heuristic search with random walks to escape local minima.

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, for instance, by [17], [11].

We have shown that our parallel formulation is capable of generating plans in very short time frames on various parallel architectures [4], including a 1024 processor nCUBE2[1], a 512-processor CM5[2], and a 16-processor network of Sun workstations [5].

3 Parallel Motion Planning

In recent years parallel search algorithms have been shown to be effective for solving combinatorially explosive problems.

Two classes of parallel search algorithms have been developed: (1) communication-based formulations that partition the search space among the processors, and (2) randomized formulations in which each processor explores the entire search space randomly with no interprocessor communication. The latter formulations are commonly referred to as randomized search methods. Though purely randomized search methods can be successful, they can often benefit from heuristic knowledge [12].

[1] nCUBE2 is a registered trademark of the nCUBE corporation

[2] CM-5 is a registered trademark of the Thinking Machines Corporation. The results obtained on the CM-5 that are presented in this paper 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.

To implement the randomized motion planning algorithm on parallel architectures, we broadcast the workspace bitmap and desired goal location to all processors, and check for a message indicating that a processor has found a solution. Each processor runs the same basic program. The only interprocessor communication is the initial broadcast and the termination check. Randomized search and random walks are the means by which the search-space is partitioned, as they probabilistically insure that each processor searches different parts of C-space.

Our parallel formulation of randomized heuristic search has proved extremely effective in solving motion planning problems, such as the example shown in Figure 1. The formulation has yielded impressive performance gains on every problem we have employed it on, sometimes delivering superlinear speedup [4].

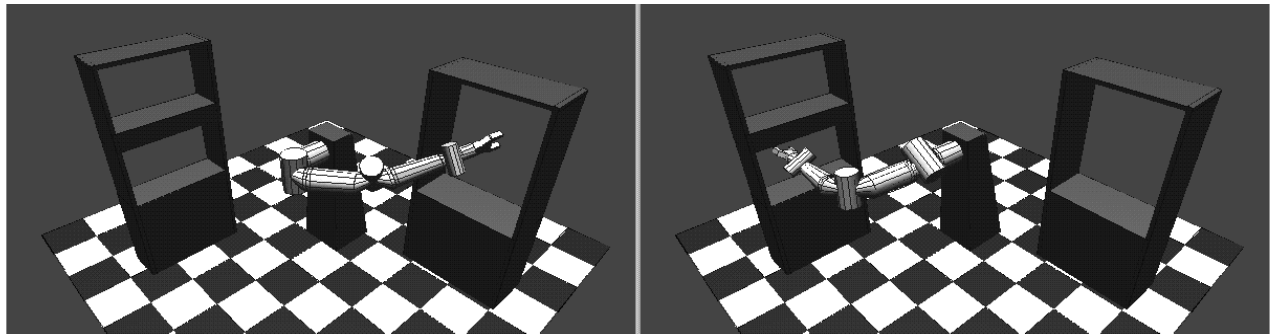
For the problem shown in Figure 1, just 16 processors are required to cut the average solution an order of magnitude to under ten seconds, and 64 processors cut the average solution time to under five seconds.

In addition to delivering paths in shorter time frames, the parallel algorithm, when executed with a larger number of processors, tends to produce better solutions by producing shorter paths. We have observed this behavior in all the experiments we have performed to date. The variance in time to solution behaves similarly, that is, it falls off as the number of processors increases.

A brief theoretical explanation for the success of parallel randomized allocation schemes is as follows. 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 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 $t_{10\%}$, and a 64 processor system has over a 99% chance of doing so. Space does not permit a more detailed analysis, which can be found in [4].



No Processors	1	4	8	16	32	64	128	256	512
Avg Search Time	127.77	26.75	16.63	9.62	6.01	4.81	2.78	2.08	1.59
Std Dev	142.00	20.92	10.41	7.02	3.71	2.69	1.53	0.78	0.33
Avg Path Length	8618	6211	5361	4550	2558	2776	1660	1179	722
Std Dev	5425	4487	4566	4189	2356	1989	1207	746	295
Speedup	1.00	4.78	7.68	13.28	21.26	26.56	45.96	61.43	80.36

Figure 1: Start and Goal Configurations for a Robotics Research K-1207i 7–*dof* arm operating in a 128^3 cell workspace. The robot is reaching from the cabinet on the right into the cabinet on the left. The table shows data for at least 64 runs on a CM-5 multicomputer. All times are in seconds, path lengths in number of steps.

We define speedup as:

$$S = \frac{E[T_1]}{E[T_k]}. \quad (2)$$

where $E[T_1]$ is the average uniprocessor solution time, and $E[T_k]$ the average k processor solution time. 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. On k processors a bad choice made by one processor does not prevent the other processors from making a better choice.

We have observed superlinear speedup when using a small number of processors in the majority of our experiments. This shows that it pays to use our parallel algorithm, even when only 4 or 8 processors are available. The speedup decreases and eventually becomes sublinear as the number of processors is increased. Furthermore, the number of processors required to deliver good performance varies from problem to problem. Thus, the following question about our parallel formulation keeps recurring: "How many processors does the method need to deliver acceptable performance?"

4 Performance Prediction

Performance prediction methods capable of determining the following two quantities are of particular interest: the time needed by a parallel algorithm to solve a problem given a fixed number of processors, and the number of processors needed to deliver a solution within a given time bound. The second of these can be obtained from the first, so we will focus on the first.

Many researchers have shown that, when the solution distribution is known, classic probabilistic methods are useful for predicting the performance that can be delivered by randomized parallel search formulations [16, 9]. Unfortunately, for any non trivial problem, the solution distribution is not known in advance. Therefore, some experimental basis is necessary to obtain an accurate prediction.

One possible method is to accurately estimate the solution distribution by computing a large number of solutions on a single processor and then use the distribution on the single processor to estimate the performance on larger numbers of processors. Ertel [7] has shown that accurate estimates can be obtained when a large number of solution times are used. We refer to this method as the T_1 estimation method. Ertel used

the T_1 estimation method to compute accurate predictions of the performance of his uninformed randomized parallel formulation on various theorem proving problems after obtaining a large sample (i.e., between 1000 and 10000 solutions) from the actual single processor solution distribution.

We have developed a method, called the T_k estimation method, capable of computing accurate estimates of both the time needed by a parallel algorithm to solve a particular problem and the number of processors needed to deliver a solution within a given time bound. The method is based on experimentally estimating the probability distribution function of the solution time on k-processor by successively executing randomized parallel search on a fixed number of processors. This is done as follows.

Let T_1 be a random variable which denotes the time taken by uniprocessor randomized search to find a solution. Assume that we obtain N samples from T_1. To obtain each sample, we execute randomized search until it finds a solution. Also assume that the sequential solution times are labelled $t_0, \ldots, t_m$, and that they are sorted in increasing order (i.e., t_0 is the smallest, t_m is the largest). The estimated probability associated with a particular solution time t_i is just the frequency of its appearance in the set of N sequential solution times sampled. Then, for each time t_i, $P_k(t_i)$ is estimated by using its definition $P_k(t_i) = P[T_k \leq t_i] = \sum_0^i P[T_k = t_i]$.

Having experimentally estimated $P_k(t)$, we can now compute the probability distribution function $P_m(t)$ on m processors in the following manner. First we solve equation 1 for $P_1(t)$. This yields:

$$P_1(t) = 1 - (1 - P_k(t))^{1/k}. \quad (3)$$

We can then derive the equation necessary for predicting the probability distribution function $P_m(t)$ by substituting equation 3 into equation 1. Doing so yields the following result:

$$\begin{aligned} P_m(t) &= 1 - \left(1 - (1 - (1 - P_k(t))^{1/k})\right)^m \\ &= 1 - (1 - P_k(t))^{m/k}. \quad (4) \end{aligned}$$

In most cases a relatively small number of solutions is required to obtain an accurate estimate of $P_k(t)$. This is because a small sample from the solution set associated with T_k contains more information about the solution distribution on a larger number of processors than a small sample from T_1 does. The reason is that we get many large solution times when sampling T_1. However, as k increases, the probability that a k-processor system will yield a large solution time decreases exponentially with k.

Computed solution times					
No Processors	1	32	64	128	256
Avg Search Time	102.34	8.39	5.36	3.37	2.32
Std Dev	108.33	5.24	3.26	2.17	1.26
Predicted solution times					
No Processors	1	32	64	128	256
$E[T_1]$ 128 runs	**107.72**	12.52	8.90	5.81	3.72
$E[T_1]$ 256 runs	**102.34**	11.04	7.42	4.56	2.94
$E[T_{32}]$	-	**8.39**	5.46	3.43	2.21
$E[T_{64}]$	-	7.45	**5.36**	3.54	2.38
$E[T_{128}]$	-	6.71	5.02	**3.37**	2.26
$E[T_{256}]$	-	6.30	4.99	3.48	**2.32**

Figure 2: Computed and predicted solution times for a problem instance. The actual average solution times are shown at the top of the table and marked in bold font in the bottom part of the table. The times were computed using 128 solutions and 256 solutions from T_1, and 64 solutions from T_k. All times are in seconds.

In other words, when sampling T_1, a significant amount of time is required to accumulate information that yields minimal information about the average solution time on a k-processor system for small to moderately large values of k. Conversely, sample points from T_k tend to belong to the group of small solution times because each of them is the minimum time of k independent runs. These smaller solution times have a higher probability of occurring in the solution time probability distribution of an m-processor system, where $m > k$. Hence, most samples from T_k yield information about the distribution that is relevant for computing $P_m(t)$. Thus, sampling T_k yields better predictions for T_m ($m > k$) with fewer solutions than sampling T_1.

There is a drawback to T_k prediction method. When we use equation 4 to predict performance on a smaller number of processors ($m < k$), our method will tend to yield optimistic predictions (i.e., predict average solution times faster than actually available). This will occur because the experimentally computed $P_k(t)$ has little or no information about the larger solution times present in the solution time distribution on a smaller number of processors, and these larger solution times are necessary to predict the performance on smaller numbers of processors accurately. As a result, downward predictions of solution times tend to be optimistic. Optimistic predictions then yield pessimistic estimations of speedup.

So, if a large number of processors is used to predict performance on a smaller number of processors, the T_k approximation method yield little meaningful

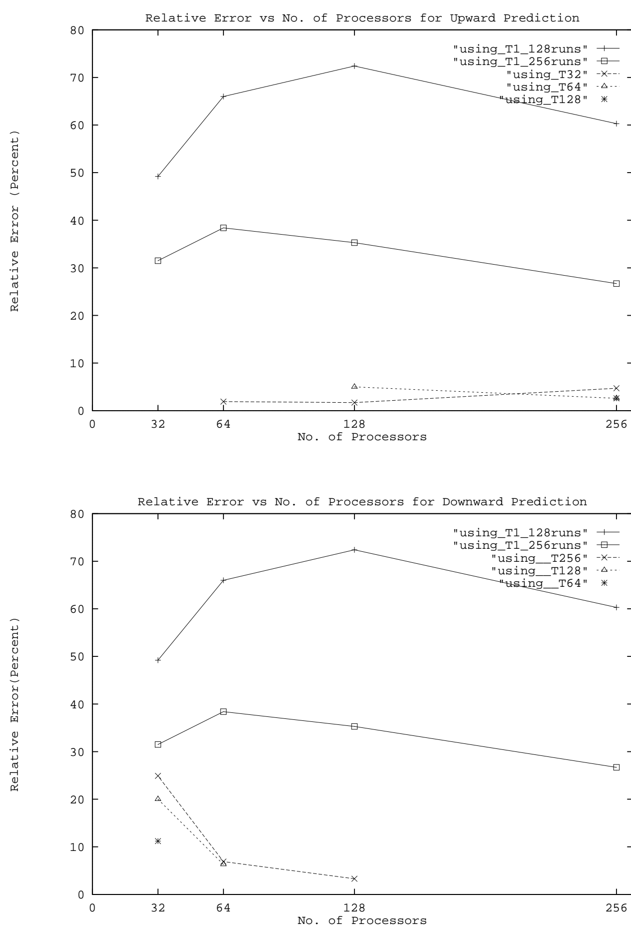


Figure 3: Percent relative errors for predicted solution times for the problem shown in Figure 2. The graphs compare the relative upward and downward prediction error delivered by the T_1 and T_k methods. The first graph shows the relative error when predicting performance upward, the second graph downward.

information. One way to avoid this situation is to use a relatively small number of processors (e.g., 32 as in our experiments), and increase the number of processors only if some of the solution times are unacceptably large.

In Figure 2 we show the results obtained by the T_k method on one of the many problem instances we examined. The table shows the experimentally computed average solution times and the predicted average solution times for each group of processors. For the T_1 method we show the results obtained with 128 and 256 solutions. For the T_k method we used 64 solutions. The average solution times that we computed experimentally are the diagonal entries in the table and are marked in bold font. Entries in a row to the left of the bold-font diagonal entry are the predicted

Relative Error per No Processors	32	64	128	256
T_1 Predictions (128 runs)	49.2	66.0	72.4	60.3
T_1 Predictions (256 runs)	31.5	38.4	35.3	26.7
T_k Upward Predictions	-	1.8	5.0	4.7
T_k Downward Predictions	24.9	11.8	3.6	-

Figure 4: Percent relative errors for predicted solution times for the T_1 and T_k methods with different numbers of processors.

solution times required by fewer processors (downward predictions), and entries in a row to the right of a diagonal entry are the predicted solution times required by larger numbers of processors (upward predictions). The number of processors for which a time is experimentally estimated or predicted is determined by the number of processors listed at the top of its column.

Graphs of the relative error delivered by the T_1 and T_k prediction methods on increasing and decreasing numbers of processors are shown in Figure 3. The percent relative error is calculated using the following formula:

$$\frac{|\text{computed value} - \text{predicted value}|}{\text{computed value}} \cdot 100 \qquad (5)$$

Each graph shows the percent relative error for T_1 estimated with $N = 128$ and $N = 256$ solutions. The first graph shows the percent relative error for upward predictions (i.e., predictions for increasing numbers of processors), the second for downward predictions (i.e., predictions for decreasing numbers of processors). We show also the relative error for the T_1 predictions, and for the T_k predictions for $k = 32, 64$, and 128.

The table in Figure 4 shows the percent relative error delivered by T_1 estimated with 128 and 256 solutions. The table also shows the highest and lowest percent relative error delivered by the T_k prediction method for upward and downward predictions. The highest and lowest percent relative error shown for each group of processors is selected from all predictions made for that group of processors. The data from the table in Figure 2 are used to compute the relative error.

5 Conclusions and Future Work

We have presented a fast performance prediction method that can be used to predict the solution times that our parallel motion planner can deliver on a larger number of processors.

Although the motion planner presented is only probabilistically complete, the experimental results we have obtained with a large variety of robots and environments indicate that the method always finds a solution.

Given enough processors a solution is found in short time frames. 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.

The number of different solution paths increases dramatically with the number of *dof* of the robot. Because of the way the planner escapes local minima and generates successors, this increased solution density enables the parallel planner to escape local minima very effectively even in instances that would be difficult for other methods.

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 Journal 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. Ph.D. dissertation, The University of Minnesota, 1995.

[5] 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.

[6] 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.

[7] W. Ertel. OR-parallel theorem proving with random competition. In A. Voronokov, editor, *LPAR'92: Logic Programming and Automated Reasoning*, pages 226–237. Springer-Verlag (LNAI 624), 1992.

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

[9] V. Janakiram, D. Agrawal, and R. Mehrotra. A randomized parallel backtracking algorithm. *IEEE Trans. Computers*, 37(12), December 1988.

[10] 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.

[11] 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.

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

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

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

[15] V. Lumelsky and E. Cheng. Real-time collision avoidance in teleoperated whole sensitive robot arm manipulators. *IEEE Trans. Systems, Man, and Cybernetics*, SMC-23(1):194–203, Jan/Feb 1993.

[16] R. Mehrotra and E. F. Gehringer. Superlinear speedup through randomized algorithms. In *Proc. Int'l Conf. on Parallel Processing*, pages 291–300, 1985.

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

[18] C. A. Shaffer. A real-time robot arm collision avoidance system. *IEEE Trans. Robotics and Automation*, RA–8(2):149–160, 1992.

[19] 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.