
Journal of Intelligent and Robotic Systems **38:** 31–53, 2003.

Predicting the Performance of Randomized Parallel Search: An Application to Robot Motion Planning

DANIEL J. CHALLOU, MARIA GINI, VIPIN KUMAR* and
GEORGE KARYPIS
Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, U.S.A.; e-mail: gini@cs.umn.edu

Abstract. In this paper we discuss methods for predicting the performance of any formulation of randomized parallel search, and propose a new performance prediction method that is based on obtaining an accurate estimate of the k-processor run-time distribution. We show that the k-processor prediction method delivers accurate performance predictions and demonstrate the validity of our analysis on several robot motion planning problems.

Key words: randomized path planning, randomized parallel search, performance evaluation, parallel computers.

1. Introduction

Among the many skills autonomous robots require to support their activities is the ability to plan the paths they must take while conducting those activities. 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 (Latombe, 1991), but most are never used in practice. Motion planning is PSPACE-hard (Reif, 1979), which implies that, as the number of joints of the robot increases, the problem quickly becomes computationally unmanageable.

Parallel search algorithms have been shown to be effective for solving combinatorially explosive problems (Ferreira and Pardalos, 1996), in particular randomized search problems. When the search space grows as a small exponent of the problem size, parallel search can provide a speedup proportional to the number of processors, which means that significantly bigger instances of the problem can be solved.

We devised (Challou et al., 1993) a parallel formulation of the Randomized Path Planner (RPP) of Barraquand and Latombe (Latombe, 1991) and applied it to a variety of motion planning problems (Challou et al., 1998). We have focused our work on robots with multiple joints and a fixed-base, robots such as the one

* Work supported in part by ARO grant DA/DAAG55-98-1-0441, ARO grant DA/DAAG55-97-1-021, and NSF grant NSF CCR-9972519.

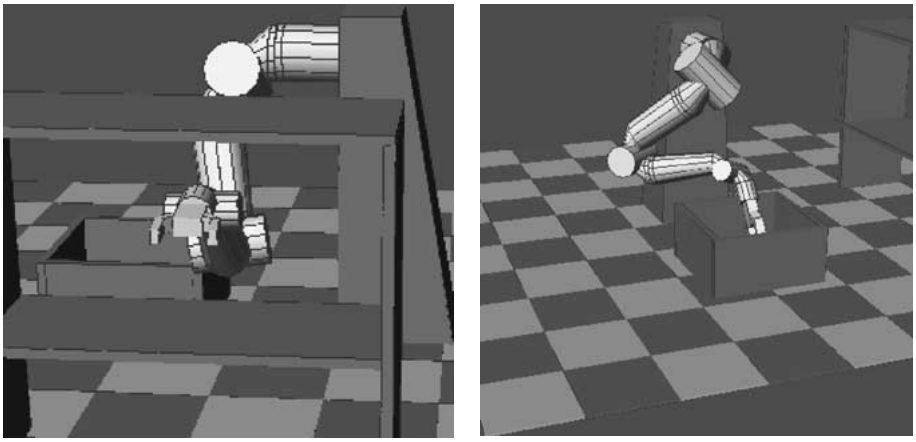


Figure 1. Start and goal configuration of a Robotics Research K-1207i 7 degrees of freedom arm. The arm has to move from the shelf cabinet into the box shown in the background. Moving into the box is difficult because of the limited space for the elbow to move.

shown in Figure 1. The robot shown has 7 degrees of freedom and has a reach comparable to that of a human arm.

Our parallel formulation has proved extremely effective, sometimes delivering superlinear speedup, as in the examples illustrated later in Figures 6 and 9. In our parallel formulation each processor runs the same program. The only inter-processor communication is an initial broadcast of the workspace and goal to all processors, and checks for a message indicating that another processor has found a solution.

However, as the number of processors used is increased, the speedup decreases. Furthermore, the number of processors required to deliver good performance varies from problem to problem. Thus, the following question keeps recurring: "How many processors does the method need to deliver acceptable performance?"

The main question we address in this paper is how to predict the performance of randomized parallel search formulations. Performance prediction methods capable of determining the following two quantities are of particular interest: (1) the time needed 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. The second of these can be obtained from the first, so we will focus on the first.

These quantities could be obtained easily if the probability distribution of the run-times were known. However, since it is not, we need a way to estimate it. We present a performance prediction method that computes accurate estimates of the time needed to solve a problem, and present results for a variety of motion planning problems.

2. Parallel RPP

2.1. A MOTIVATING EXAMPLE

Let us consider the example shown in Figure 1. The example is motivated by the set of benchmarks proposed as part of the SKALP (SCalable ALgorithms for highly parallel motion Planning) project at the University of Karlsruhe (Henrich et al., 1998) and it is of particular interest when using humanoid robots as helpers in a house. Think of the problems an arm will have to face when placing food from a shopping bag into a refrigerator or when picking up an object behind another object.

Given the growing availability of processing time via Internet connected networks of workstations, the algorithm discussed in this paper can be used to solve such problems quickly and ensure minimum use of such capability in order to do so.

In the example shown there are many local minima in configuration space, where the arm can easily get trapped. This makes the problem hard for most motion planning algorithms. However, the random search components of the algorithm we use, RPP, allow the robot to escape from the local minima.

2.2. DESCRIPTION OF OUR PARALLEL IMPLEMENTATION OF RPP

An outline of our parallel formulation of RPP is given in Figure 2.

Consider a search beginning at the start configuration. The gradient descent forces the search in the direction of the node that appears more promising. Successors of a node are generated in a random manner until a successor is found that has a better heuristic value than the current node. Thus, the first legal successor with a better value than its parent is adopted as the next step in the path.

However, since the heuristic is often misleading, at some point, the search might reach a local minimum, where every successor of the current node is worse than the current node. When this occurs, random search with a randomly chosen depth bound is executed. This step is called a random walk.

This is followed by Randomized Gradient Descent, which continues until a local minimum is reached. The sequence of random walk followed by gradient descent is repeated for a predetermined number K of trials or until a better node is found. When a better configuration is found the new part of the path found is appended to the previous path and the process resumes. If, after K trials, no better node has been found, then Random Backtrack begins from a randomly picked configuration in the current path. The cycle of random walks followed by gradient descent is then resumed.

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.

```
ALGORITHM. Parallel RPP
repeat
  Construct initial path via Randomized Gradient Descent
  until a local minimum occurs
  while goal not found and no time-out
    repeat K times or until improvement found
      Form new temporary path from end of initial path by executing
        Random Walk to escape local minimum
        if TERMINATION MESSAGE RECEIVED
          then return NO SOLUTION
        Randomized Gradient Descent until a local minimum
        if TERMINATION MESSAGE RECEIVED
          then return NO SOLUTION
      if improvement found
        then append new temporary path to end of initial path
      if no improvement found
        then Randomly Backtrack to a new point in initial path
    end while
  until goal found or global time-out
  if SOLUTION FOUND then
    BROADCAST TERMINATION MESSAGE TO ALL PROCESSORS
  return solution or no solution
```

Figure 2. Outline of our parallel formulation of RPP. The capitalized statements show the message passing needed to run on multicomputers.

There are a number of parameters that control the random parts of the algorithm. For instance, the duration of the random walk (i.e., the number of collision-free configurations to be generated and inserted into the path without regard to their heuristic value) depends on the level of discretization of the work space (the finer the discretization is the longer the random walk), the total number of attempts to find a solution, and the total number of attempts to find a solution starting with a particular initial path.

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 randomization in the state generation process, random walks, and randomized backtracking are the means by which each processor explores a probabilistically different part of the search space.

ALGORITHM. Randomized Gradient Descent

```
Set the Path to the Current_configuration.
Obtain the robot's control point positions by computing the forward kinematics of
  the Current_configuration.
Set the Heuristic_value to the numerical potential field value associated with the
  current control point position(s).
Set the number of Successor_trials to 0.
Set the New_member flag to false.
while (Goal Not Found and Successor_trials < Max_successor_trials)
  while (Successor_trials < Max_successor_trials and New_member flag is false)
    Set the Candidate_configuration to a
      randomly generated successor to the Current_configuration.
    Obtain the robot's control point positions by computing
      the forward kinematics of the Candidate_configuration.
    Set the Candidate_heuristic_value to the numerical potential
      field value associated with the current control point position(s).
    if (Candidate_heuristic_value < Heuristic_value
       and the Candidate_configuration is Collision Free)
       then
         Set the New_member flag to true.
    Increment the number of Successor_trials.
  end while
  if (New_member flag is true)
    then
      Append the Candidate_configuration to the Path.
      Set the Current_configuration equal to the Candidate_configuration.
      Set the Heuristic_value equal to the Candidate_heuristic_value.
      Set the number of Successor_trials to 0.
      Set the New_member flag to false.
  end while
return Path, Heuristic_value.
```

Figure 3. Outline of the Randomized Gradient Descent procedure. The procedure attempts to construct a solution by generating configurations, one at a time, and appending them to the path if they are suitable.

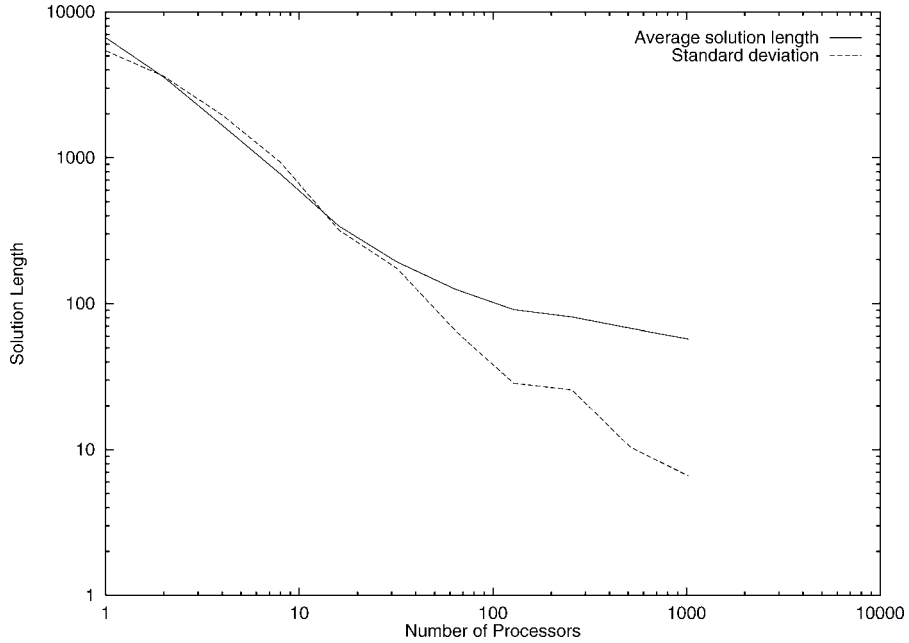


Figure 4. Length of solution paths found for the problem shown in Figure 1. The length decreases dramatically as the number of processors increases.

2.3. ADVANTAGES OF PARALLEL MOTION PLANNING

There are many reasons for using parallel randomized search for motion planning. Speeding up the search process is one of them, finding better solutions by obtaining shorter paths is another. Due to the random component of the search algorithm, solutions can be much too long and include many unnecessary moves. However, when solutions are found with parallel search, better solutions are found more rapidly, and since the search stops as soon as the first solution is found, the quality of the solutions improves with the number of processors.

We can see this by examining the example in Figure 4, which shows the average length of solutions for different numbers of processors for the problem shown in Figure 1.

To obtain good performance, other motion planning methods have been devised, which reduce the computation time by doing a significant amount of preprocessing. The most well known example is the PRM algorithm (Kavraki and Latombe, 1998), where a roadmap that covers the free space is computed upfront and the motion planning problem is then solved by finding a path from the start point to the roadmap, traversing the roadmap, and finding a path from the roadmap to the goal point. Since preprocessing in PRM is computationally expensive, the use of parallel processing has been proposed to speed it up (Hsu et al., 1999; Amato and Dale, 1999). Other methods, such as the one proposed in (Kim and Boley, 2001) to construct a network of local minima, are also easily parallelizable.

3. Parallel Search and Performance Prediction Methods

Two types of parallel search formulations have been developed in the literature. In both formulations the first processor to find a solution sends a termination signal to the remaining processors, and then reports its solution.

- Communication-based schemes distribute the task to be performed among processors by using interprocessor communication. Many researchers have demonstrated the utility of communication-based formulations for solving a variety of state-space search problems (Rao and Kumar, 1993; Cook and Varnell, 1998). Grama and Kumar provide a extensive review of parallel techniques for systematic search (Grama and Kumar, 1999).
- In randomized allocation schemes each processor runs the same randomized search, and no interprocessor communication is used to partition the search space. Randomized allocation schemes do not insure that there is no duplication of work, but, in some sense, they probabilistically partition the search space among processors.

In general, communication-based formulations are more efficient than random-allocation schemes (Rao and Kumar, 1993), but randomized schemes require little or no interprocessor communication, and thus they are much easier to implement on a variety of architectures (Ertel, 1993; Challou et al., 1998).

Recall from the introduction that we are interested in performance prediction methods that predict the time needed to solve a particular problem given a fixed number of processors.

Many researchers have shown that, when the run-time distribution is known, classic probabilistic methods are useful for predicting the range of performance that can be delivered by randomized parallel search (Mehrotra and Gehringer, 1985; Janakiram et al., 1988; Karp and Zhang, 1993) and other problems in which each processor runs the same program (Alanberg-Navony et al., 1994). Unfortunately, for any non trivial problem, the run-time distribution is not known in advance. Therefore, some experimental basis is necessary to obtain an accurate prediction.

Characterizing the run-time distribution has to be done on individual problem instances, as opposed to be done on a collection of problems. As detailed by Hoos (1998), averaging over exponentially distributed random variables yields a random variable of a different distribution type.

Most stochastic search methods start their searches in one (usually randomly chosen) location in the search space and then do some sort of random walk in the space. The consensus (see, for example, (Reeves, 1993)) is that the best use of time is to run several shorter searches starting from multiple locations in the space, rather than spending all the available time on one search.

If the run-time distribution of the algorithm is exponential, the probability of finding a solution on k processors when running the algorithm for time t/k is the same as the probability of finding a solution when running the algorithm on

a single processor for time t (Hoos and Stützle, 1998). If the run-time distribution of the algorithm is less steep than an exponential distribution, running the algorithm on a parallel computer will increase performance. In this case the algorithm produces superlinear speedups, which means that doing multiple restarts on a single processor is equally advantageous. However, if the run-time distribution is steeper than exponential, parallelizing the computation (or, equivalently, restarting the search) does not produce the same improvement, and the speedup obtained is sublinear (Hoos and Stützle, 1998).

The simplest method for obtaining an experimental basis for predictions is to solve the problem using increasing numbers of processors (i.e., solve the problem with $1, 2, \ldots, k$ processors, where k is the maximum number of processors that are available). The run-times obtained on each group of processors will enable computation of the optimal number of processors required to solve the problem and the number of processors required to solve the problem in less than a given time bound.

There are problems with this approach however. First, randomized methods are nondeterministic, so the run-times obtained on each group of processors varies. Therefore, many solutions must be obtained on each group of processors in order to obtain an accurate measure of the average run-time and its variance. Thus, formulating accurate performance predictions with this approach requires a large amount of time and computing resources.

One possible alternative is to accurately estimate the single processor solution distribution by computing a large number of solutions on a single processor and then estimating the performance on larger numbers of processors by using the single processor distribution. We refer to this method as the T_1 estimation method. Ertel (1992, 1993) proposed such a scheme and used it to compute accurate predictions of the performance of his 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.

3.1. THE T_1 ESTIMATION METHOD

Let T_1 be a random variable denoting the run-times for a uniprocessor randomized search, and T_k be a random variable which denotes the times taken by k-processors to find a solution.

We can estimate the probability of finding a solution on a single processor in time t_i using the frequency of the appearance of t_i (denoted $n(t_i)$) in the set of N sequential run-times observed. More formally,

$$P[T_1 = t_i] \approx \frac{1}{N} \cdot n(t_i), \quad \text{where } N = \sum_{i=0}^{m} n(t_i). \tag{1}$$

Once the probabilities have been estimated and associated with each run-time in the sample, we can use them to estimate the probability distribution function. This is accomplished as follows.

Let $p_1(t)$ denote the estimated probability distribution function of T_1. We can compute $p_1(t)$, or $P[T_1 \leqslant t]$ for a particular t_i by summing up all the estimated probabilities associated with times $t_0, \ldots, t_i$. In other words,

$$p_1(t) = P[T_1 \leqslant t_i] = \sum_{0}^{i} P[T_1 = t_i].$$

If an accurate estimate of $p_1(t)$ is known, we can compute an accurate estimate of $p_k(t)$, the probability distribution of T_k, as follows:

$$\begin{aligned} p_k(t) = P[T_k \leqslant t] &= P\left[\min(T_1^1, T_1^2, \ldots, T_1^k) \leqslant t\right] \\ &= 1 - P\left[\min(T_1^1, T_1^2, \ldots, T_1^k) > t\right] \\ &= 1 - P\left[T_1^1 > t \text{ and } T_1^2 > t \text{ and } \ldots\ T_1^k > t\right] \\ &= 1 - P[T_1 > t]^k \\ &= 1 - (1 - P[T_1 \leqslant t])^k \\ &= 1 - (1 - p_1(t))^k, \end{aligned} \tag{2}$$

where T_1^i is a random variable denoting the run-times for uniprocessor randomized search on processor i. Note that T_1^i for $i = 1, 2, \ldots, k$ are independent identically distributed random variables with probability distribution $p_1(t)$.

This formula can be used to explain the success of parallel randomized allocation schemes. Suppose a single processor has only a 10% probability that it can solve a 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.

We define $E[T_1]$ as the average uniprocessor run-time, $E[T_k]$ as the average k processor time, and speedup as

$$S = \frac{E[T_1]}{E[T_k]}. \tag{3}$$

If $E[T_k]$ is less than $E[T_1]$, then, on average, the k processor solution will deliver speedup over the uniprocessor algorithm. If $E[T_k]$ is less than $(1/k)E[T_1]$ then, on k processors, the parallel randomized allocation formulation will yield superlinear speedup over the uniprocessor algorithm (on average) (Li and Wah, 1986). 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.

An estimate of the expected run-time on k processors for $k = 1, \ldots, \infty$ can be computed as follows:

$$E[T_k] = \sum_t P[T_k = t] \cdot t. \tag{4}$$

Alternatively, if we assume that the probability that a solution will be found in a finite time is one, we can compute the expected value with the following equation (Hoel et al., 1971):

$$\begin{aligned} E[T_k] &= \sum_t P[T_k > t] = \sum_t 1 - P[T_k \leqslant t] \\ &= \sum_t 1 - p_k(t) = \sum_t (1 - p_1(t))^k. \end{aligned} \tag{5}$$

3.2. THE T_k ESTIMATION METHOD

In the same way as we estimate $E[T_1]$ from solutions samples on a single processor, we can estimate $E[T_k]$ directly using solutions samples from the actual T_k solution set, where $k > 1$. This is done experimentally by estimating the probability distribution function p_k with a set of samples from the actual T_k. We call this the T_k estimation method.

The experimentally estimated $p_k(t)$ can then be used to predict the performance on an m-processor system where $m \neq k$. When $m < k$ we call this *downward* prediction, when $m > k$ we call it *upward*. This can be done as follows.

First, as with the T_1 estimation method, a set of k-processor run-times are experimentally obtained. Next, approximate probabilities are computed for the experimentally obtained times in a manner similar to Equation (1). Then, for each time t_i, $p_k(t_i)$ is estimated by using its definition (i.e., $p_k(t_i) = P[T_k \leqslant t_i] = \sum_0^i P[T_k = t_i]$). As with the single and k processor case, the possible run-times on the m processor system are the same as for the k-processor system, only the run-time probabilities differ.

Having experimentally estimated $p_k(t)$, we can now compute $p_m(t)$ in the following manner. First we solve Equation (2) for $p_1(t)$. This yields

$$p_1(t) = 1 - (1 - p_k(t))^{1/k}. \tag{6}$$

We then derive the equation necessary for predicting the probability distribution function $p_m(t)$ by substituting Equation (6) into Equation (2). Doing so yields the following result:

$$\begin{aligned} p_m(t) &= 1 - \left(1 - \left(1 - (1 - p_k(t))^{1/k}\right)\right)^m \\ &= 1 - (1 - p_k(t))^{m/k}. \end{aligned} \tag{7}$$

Hence, once an estimate of $p_k(t)$ has been computed, $p_m(t)$ can be estimated using Equation (7). The average run-time on m processors can then be predicted using Equation (5).

3.3. COMPARING THE T_1 AND T_k ESTIMATION METHODS

If the estimated $p_1(t)$ or $p_k(t)$ closely approximate the actual probability distribution function associated with T_1 or T_k, then either method can be used to accurately predict the performance on a larger number of processors. The accuracy of the estimated $p_1(t)$ or $p_k(t)$ can be insured by obtaining a large number of samples from the actual T_1 or T_k. This process is time consuming, particularly when estimating $p_1(t)$.

For example, in order to insure a good prediction using the T_1 estimation method, between one thousand and ten thousand solutions are used to formulate the predictions shown in (Ertel, 1992, 1993). Computing one thousand uniprocessor solutions from T_1 in succession would require over 16 hours for a problem that requires an average of 60 seconds to solve. Moreover, the examples shown in (Ertel, 1992, 1993) and in Section 4 show that such long uniprocessor run-times are not uncommon.

The question is: what is the effect on the accuracy of the prediction of using a smaller number of solutions (e.g., one or two hundred) to estimate $p_1(t)$? Unfortunately, when few solutions are used for T_1, the method has poor performance prediction capability, and tends to yield pessimistic predictions, as reported by Ertel.

On the other hand, in most cases a smaller number of solutions is sufficient 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 larger numbers of processors than a small sample from T_1 does. The reason is that we get many large run-times when sampling T_1. However, as the number of processors k increases, the probability that a k-processor system will yield a large run-time decreases with k.

In other words, when sampling T_1, a significant amount of time is required to accumulate information that yields minimal information about the average run-time on a k-processor system. Sample points from T_k tend to be small run-times because we take the minimum time of k independent runs. These smaller run-times have a higher probability of occurring in the run-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. This is supported by our experimental results shown in Section 4.

There are two drawbacks to the T_k prediction method. First, note that when we use equation 7 to predict performance on a smaller numbers of processors (i.e., for $m < k$), our method will tend to yield optimistic predictions (i.e., predict average

run-times faster than actually available). This will occur because the experimentally computed $p_k(t)$ has little or no information about the larger run-times present in the run-time distribution on smaller numbers of processors, and these larger run-times are necessary to predict the performance on smaller numbers of processors accurately.

Second, if a large number of processors is used to predict performance, the T_k approximation method can yield little meaningful information. This is due to the inability of the T_k prediction method to predict performance accurately on a smaller number of processors. 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 only if the run-times are too long. This is useful when making predictions on a network of workstations, since it is not always easy to have very large numbers of them.

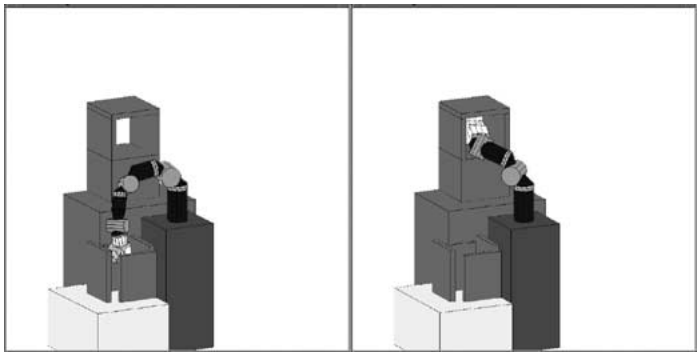
In any case, at this point it should be clear that in order to be useful, a performance prediction method should be fast and accurate. These requirements are somewhat conflicting. Any method that can deliver fast predictions must base its predictions on a small number of solutions, but accuracy requires a larger number of solutions. We will show that in our experiments the T_k method produces accurate results with relatively few solutions.

There is one more point we should discuss. When using multiple processors, even if fewer runs are needed, is the total CPU time needed to run the smaller number of experiments for T_k higher than the CPU time needed when using T_1? Our experiments show that the total CPU time needed to run enough experiments for T_k is comparable to the CPU time needed to run a number of experiments for T_1 that is too small to get good predictions. In other words, if we had to pay the same amount for the CPU time on different architectures, it would be more economical to use T_k than T_1.

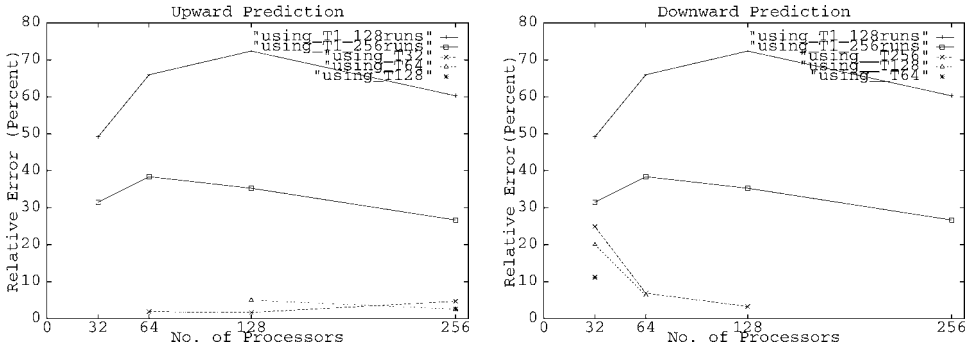
4. Experimental Results

Each of Figures 5–9 shows a picture of a problem instance. The problems are shown in decreasing order of difficulty (i.e., the problem that requires the largest average run-time is shown first, and the problem that requires the smallest average run-time is shown last). In each case we use a model of a seven-jointed arm. Each joint on the robot has 128 discrete positions and the workspace consists of 128^3 cells, each representing a volume of approximately 2.1 cubic centimeters.

Below the picture in each figure is a table showing the average run-times obtained experimentally and the average run-times predicted for each group of processors. The average run-times obtained experimentally for T_1 and each T_k are the diagonal entries in each table and are in bold. Entries in a row to the left of the bold-font diagonal entry predict the run-time required by fewer processors, and times in a row to the right of a bold-font diagonal entry predict the run-time required

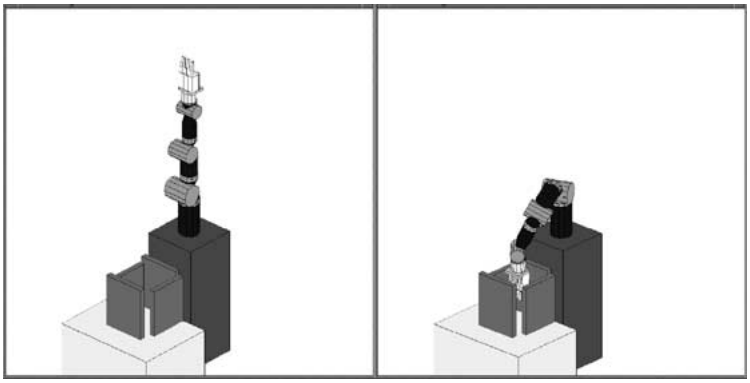


Number of processors	1	32	64	128	256
$E[T_1]$ (128 runs)	**107.72**	12.52	8.90	5.81	3.72
$E[T_1]$ (256 runs)	**102.34**	11.04	7.42	4.56	2.94
$E[T_{32}]$ (64 runs)	–	**8.39**	5.46	3.43	2.21
$E[T_{64}]$ (64 runs)	–	7.45	**5.36**	3.54	2.38
$E[T_{128}]$ (64 runs)	–	6.71	5.02	**3.37**	2.26
$E[T_{256}]$ (64 runs)	–	6.30	4.99	3.48	**2.32**
Std. dev. (σ)	108.33	5.24	3.26	2.17	1.26
Avg. speedup ($E[T_1]/E[T_K]$)	1.00	12.02	19.09	30.37	44.11

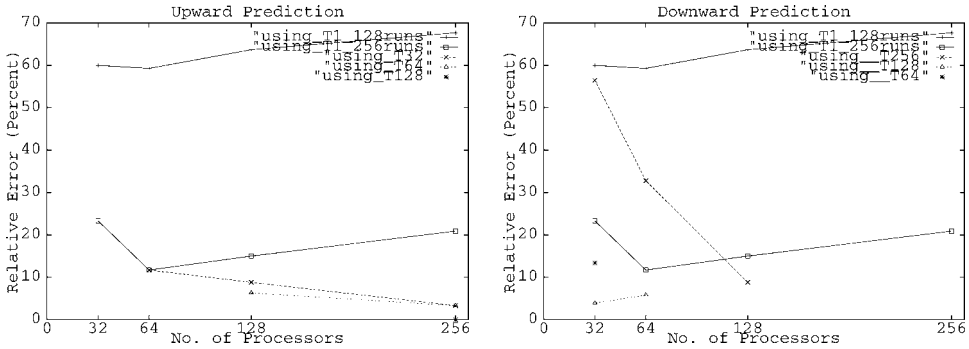


Number of processors	32	64	128	256
% relative error T_1 predictions (128 runs)	49.2	66.0	72.4	60.3
% relative error T_1 predictions (256 runs)	31.5	38.4	35.3	26.7
% relative error T_k upward predictions	–	1.8	5.0	4.7
% relative error T_k downward predictions	24.9	11.8	3.6	–

Figure 5. Start and goal configurations, average and predicted run-times (in seconds), graphs comparing relative prediction errors. (on the left upward predictions, on the right downward predictions), relative error for the T_1 method and worst relative error for the T_k method.

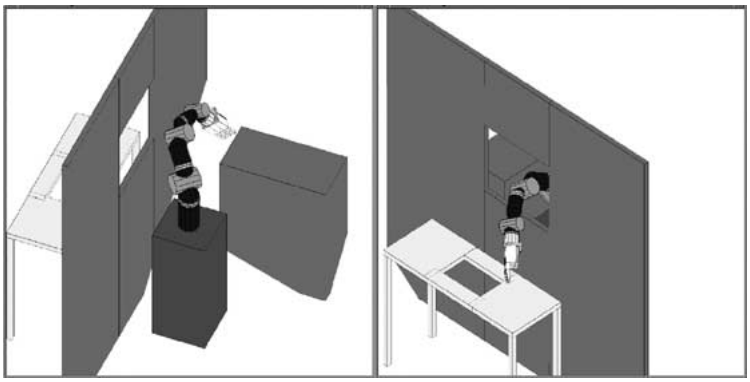


Number of processors	1	32	64	128	256
$E[T_1]$ (128 runs)	**88.46**	3.57	2.04	1.31	1.04
$E[T_1]$ (256 runs)	**83.02**	2.75	1.43	0.92	0.75
$E[T_{32}]$	–	**2.23**	1.13	0.73	0.60
$E[T_{64}]$	–	2.53	**1.28**	0.75	0.60
$E[T_{128}]$	–	2.10	1.33	**0.80**	0.62
$E[T_{256}]$	–	0.97	0.86	0.73	**0.62**
Std. dev. (σ)	104.13	2.41	1.39	0.53	0.14
Avg. speedup ($E[T_1]/E[T_K]$)	1.00	37.23	64.86	102.49	133.90

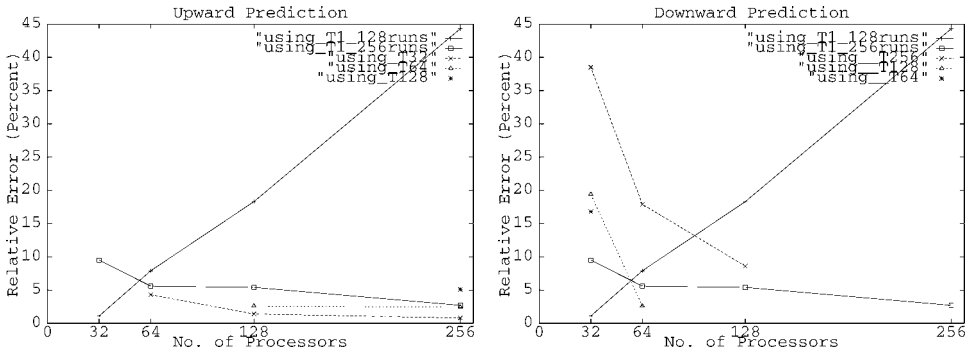


Number of processors	32	64	128	256
% relative error T_1 predictions (128 runs)	60.0	59.3	63.7	67.7
% relative error T_1 predictions (256 runs)	23.3	11.7	15.0	20.9
% relative error T_k upward predictions	–	11.7	8.8	3.3
% relative error T_k downward predictions	56.5	32.8	8.8	–

Figure 6. In this problem, the robot is reaching down into the small box in front of it. The rest of the figure contains the same information as Figure 5.

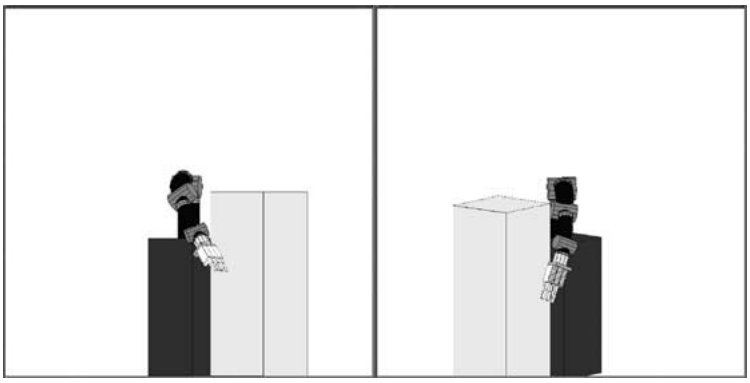


Number of processors	1	32	64	128	256
$E[T_1]$ (128 runs)	**62.80**	6.97	5.06	4.14	3.68
$E[T_1]$ (256 runs)	**57.28**	6.38	4.42	3.31	2.62
$E[T_{32}]$	–	**7.05**	4.89	3.45	2.53
$E[T_{64}]$	–	5.86	**4.69**	3.41	2.61
$E[T_{128}]$	–	5.68	4.57	**3.50**	2.68
$E[T_{256}]$	–	4.33	3.85	3.20	**2.55**
Std. dev. (σ)	59.96	3.96	2.35	1.48	0.91
Avg. speedup ($E[T_1]/E[T_K]$)	1.00	8.12	12.21	16.37	22.46

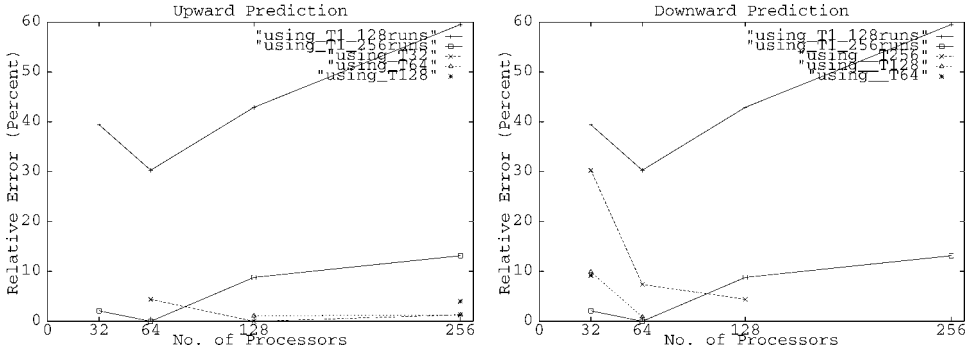


Number of processors	32	64	128	256
% relative error T_1 predictions (128 runs)	1.1	7.9	18.3	44.3
% relative error T_1 predictions (256 runs)	9.5	5.6	5.4	2.7
% relative error T_k upward predictions	–	4.3	1.1	5.1
% relative error T_k downward predictions	38.5	17.9	8.6	–

Figure 7. In this problem instance, the robot is reaching from the solid dark table behind it, through an opening in the wall on its left, and down to the light table with the hole in it. The rest of the figure contains the same information as Figure 5.

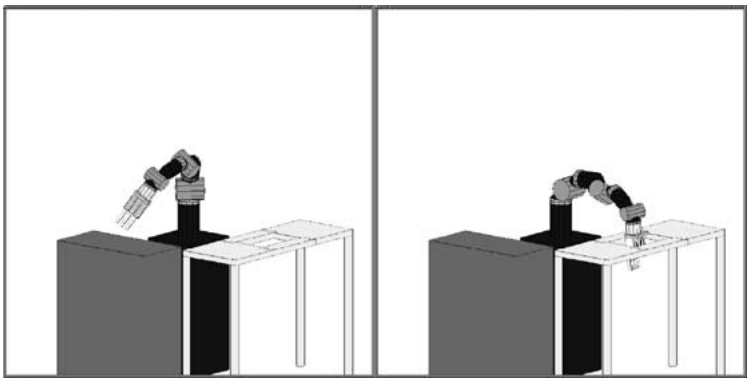


Number of processors	1	32	64	128	256
$E[T_1]$ (128 runs)	**22.85**	1.98	1.55	1.30	1.18
$E[T_1]$ (256 runs)	**21.61**	1.45	1.08	0.83	0.64
$E[T_{32}]$	–	**1.42**	1.13	0.91	0.75
$E[T_{64}]$	–	1.29	**1.08**	0.90	0.75
$E[T_{128}]$	–	1.28	1.09	**0.91**	0.77
$E[T_{256}]$	–	1.09	1.00	0.87	**0.74**
Std. dev. (σ)	23.04	0.52	0.31	0.26	0.20
Avg. speedup ($E[T_1]/E[T_K]$)	1.00	15.22	20.01	23.75	29.20

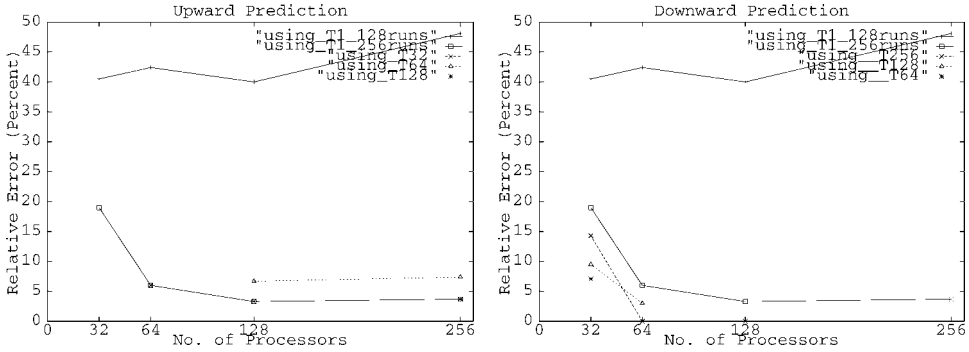


Number of processors	32	64	128	256
% relative error T_1 predictions (128 runs)	39.4	30.3	42.9	59.5
% relative error T_1 predictions (256 runs)	2.1	0.0	8.8	13.15
% relative error T_k upward predictions	–	4.4	1.1	3.9
% relative error T_k downward predictions	30.3	7.4	4.4	–

Figure 8. In this problem, the robot is reaching around the big box in front of it. The rest of the figure contains the same information as Figure 5.



Number of processors	1	32	64	128	256
$E[T_1]$ (128 runs)	**20.59**	0.59	0.47	0.42	0.40
$E[T_1]$ (256 runs)	**19.86**	0.50	0.35	0.31	0.28
$E[T_{32}]$	–	**0.42**	0.35	0.31	0.28
$E[T_{64}]$	–	0.39	**0.33**	0.28	0.25
$E[T_{128}]$	–	0.38	0.34	**0.30**	0.26
$E[T_{256}]$	–	0.36	0.33	0.30	**0.27**
Std. dev. (σ)	35.42	0.16	0.08	0.06	0.05
Avg. speedup ($E[T_1]/E[T_K]$)	1.00	47.29	60.18	66.20	73.56



Number of processors	32	64	128	256
% relative error T_1 predictions (128 runs)	40.5	42.4	40.0	48.1
% relative error T_1 predictions (256 runs)	19.0	6.0	3.3	3.7
% relative error T_k upward predictions	–	6.0	6.6	7.4
% relative error T_k downward predictions	14.3	3.0	0.0	–

Figure 9. In this problem, the robot is moving its hand from over the dark table next to it down and through the small opening in the table in front of it. The rest of the figure contains the same information as Figure 5.

by larger numbers of processors. 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.

Average speedup is calculated from the average run-times for $N = 256$ samples from T_1, and $N = 64$ samples from each T_k, $k = 32, \ldots, 256$. We used 64 solutions to estimate T_k, because, for the examples shown here, the sample standard deviation indicated that 64 solutions were enough to provide an accurate estimate of the actual k processor distribution.

Graphs of the relative error delivered by the T_1 and T_k prediction methods for upward and downward predictions are shown. The percent relative error is calculated using the following formula:

$$\frac{|\text{experimental avg. run-time} - \text{predicted avg. run-time}|}{\text{experimental avg. run-time}} \times 100. \tag{8}$$

Each graph shows the percent relative error for T_1 estimated with $N = 128$ and $N = 256$ solutions. The graph on the left shows the percent relative error for upward predictions (i.e., predictions for increasing numbers of processors). In addition to the relative error for the T_1 predictions, the predictions made from T_k for $k = 32, 64$, and 128 processors are also shown. The graph on the right shows the percent relative error for downward predictions (i.e., predictions for decreasing numbers of processors). In addition to the relative error for the T_1 predictions, the predictions made from T_k for $k = 64$, 128, and 256 processors are shown as well.

The table at the bottom of each figure shows the percent relative error delivered on each number of processors by T_1 estimated with $N = 128$ and $N = 256$ 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 form the table included at the top of each figure is used to compute the relative error.

The times shown were obtained on a CM-5 parallel computer (Corporation, 1992). The reason for the choice was convenience of access to a large number of processors. However, the CM-5 processors are slow and with limited memory (32 Megabytes), so the timing results should not be taken as the best timing the algorithm can produce today on faster processors. Since many of the runs were made in time sharing mode, the run-times were obtained by multiplying the average node expansion time by the number of node expansions. All times are in seconds.

5. Discussion

5.1. PREDICTING PERFORMANCE ON INCREASING NUMBERS OF PROCESSORS

As our analysis predicts, the T_k prediction method requires fewer solutions in order to formulate an accurate prediction than the T_1 prediction method. For instance,

consider the example in Figure 5. The table at the bottom of the figure shows that the *minimum* relative error delivered by T_1 approximated with 256 solutions is 26.7% and the *maximum* relative error delivered delivered by T_k approximated with 64 solutions is 5%. For the problem instance in Figure 6, the *minimum* relative error delivered by T_1 approximated with 256 solutions is 11.7% and the *maximum* relative error delivered by the T_k approximated with 64 solutions is 11.7%. Thus the most inaccurate predictions delivered by the T_k method estimated using 64 solutions are more accurate than, or equal to the accuracy of the best predictions delivered by the T_1 method estimated with 256 solutions for the most time consuming problems shown. In addition, the maximum relative error delivered by the T_k prediction method in the remaining examples is 7.4%, while the maximum relative error delivered by the T_1 method estimated with 256 solutions exceeds 10% in 2 cases.

Furthermore, 11 of the 15 predictions delivered by the T_k method are equal to, or more accurate than, the predictions delivered by T_1 prediction method based on 256 solutions. The only exceptions include one prediction for one group of processors on each of the problems depicted in Figures 7 and 8, and two predictions made from T_{64} for the problem depicted in Figure 9.

Consider the relative error graph for upward prediction in each example when only 128 solutions are used to estimate T_1. In each problem shown, the upward performance predictions delivered by the T_k method are significantly more accurate than then predictions delivered by the T_1 method. Specifically, the *minimum* relative error delivered by the T_1 method when T_1 is estimated with 128 solutions is approximately twice the maximum relative error delivered by the T_k method when T_k is estimated with 64 solutions (e.g., see the error delivered by the T_k and T_1 methods for 64 processors in Figure 7).

Hence, from our results it should be clear that the T_k prediction method requires significantly fewer solutions in order to formulate accurate performance predictions for a larger number of processors (i.e., when $m > k$) than the T_1 method. Finally, consider the predictions formulated from T_1 with 128 solutions shown in each example. With the exception of 32 processors for problem 7, they are pessimistic for each group of processors in every problem shown. Thus, the data from our examples overwhelmingly confirm that predictions formulated with the T_1 method are pessimistic (and inaccurate) when T_1 is estimated with too few samples.

5.2. PREDICTING PERFORMANCE ON DECREASING NUMBERS OF PROCESSORS

Consider the downward predictions formulated by the T_k method shown in the table below the figure in each of the examples. The data show, as our analysis predicts, that the T_k prediction method tends to predict lower than actual runtimes on decreasing numbers of processors. Recall that this is because the k-processor

solution has little information about the longer run-times that tend to be present in the run-time distributions on smaller numbers of processors.

Nonetheless, consider the graphs showing the relative error for downward predictions for each example. They show that the T_k method is reasonably effective for predicting performance on one-half the number of processors used to compute the estimate. In particular, the T_k method delivers more accurate predictions on one half the number of processors used to compute the estimate than the T_1 method estimated with 256 solutions in 13 out of 15 cases. The two cases are the performance predictions for 32 processors formulated from T_{64} for problems 7 and 8. In both cases the predictions are less accurate than the predictions formulated for 32 processors from T_1 estimated with 256 solutions.

In all other cases, the relative error for predictions on one-half the actual number of processors used to estimate T_k is a maximum of a little over 10%. In addition, for problems 5 and 9, all the downward predictions formulated from T_k are more accurate than the predictions formulated by T_1 estimated with 256 solutions. Thus, our experimental results indicate that the T_k prediction method is useful for predicting performance on one half as many processors, but not for formulating "long-range" performance predictions on decreasing numbers processors.

5.3. WHICH METHOD DELIVERS ACCURATE PREDICTIONS FASTER: T_1 OR T_k?

Our analysis in Section 3.3 and our discussion above show that the T_1 method requires more solutions than the T_k method in order to formulate predictions with the same degree of accuracy.

For the problems in Figures 5 and 8 the minimum relative error delivered by T_1 estimated with 128 solutions is over 30%. In addition, in problem 7, the error ranges between a little over one percent to over 44%. Thus, the predictions formulated with 128 solutions from T_1 are virtually meaningless for each of the aforementioned problems.

If the number of solutions from T_1 is raised to 256, then the accuracy of the T_1 estimation method becomes, in some cases, close to the accuracy of the T_k method. The relative error in the predictions formulated by the T_1 method decreases dramatically for problems 7 and 8, but remains relatively high for problem 5 (i.e., over 25% in all cases).

For each problem, there is a number of processors k for which both the T_1 and T_k methods require the same total CPU time in order to deliver equally accurate predictions. We call this the *crossover point*. The crossover point depends on the speedup (and efficiency) available on a particular problem instance. For example, in our examples, problem 7 delivers the lowest overall speedups (and efficiencies). The total CPU times for each number of processors show that the crossover point is approximately 32 processors. Thus, for this problem, when 32 processors or fewer are used predict the performance, the T_k estimation method will outperform the T_1 estimation method.

Examples 8 and 5 also show that as the available speedup increases, then the number of processors on which the T_k method outperforms the T_1 method in terms of speed and accuracy also increases. For example, for problem 8, the available speed up on the problem instance is higher than for problem 7 and it outperforms the T_1 method on up to 64 processors. In addition, for problem 5 it appears that the T_k method will outperform the T_1 method in terms of speed and accuracy on even larger numbers of processors.

If a large number of processors is available, then it may seem that a faster method for predicting performance might be to simply obtain solutions from T_1 using each processor, and then use the T_1 method to formulate predictions. However, again consider the problem which yields the minimum speedup in our experiments pictured in Figure 7. Next, assume 256 processors are available for formulating predictions. On average 57.28 sec. would be required to compute 256 run-times using the T_1 method.

Conversely, if the T_k method is emulated for $k = 32$ then, on average, 8 solutions can be obtained from T_{32} using 256 processors in an average time of $E[T_{32}]$, or 7.05 sec. In order to compute an accurate prediction from T_{32}, 64 solutions are required. The time required to obtain 64 solutions from T_{32} in this manner is $E[T_{32}] \cdot 8$, or 56.4 sec. Thus, the T_k method outperforms the T_1 method even when a fairly significant number of processors and only low speedups are available, provided that a small k is used. Finally, the analysis and results also show that if a parallel machine is not available, emulating a k processor randomized parallel search on a single processor is the fastest way to compute a useful set of solutions.

6. Concluding Remarks

To conclude, both our analysis and experimental results show that the T_k estimation method delivers more accurate predictions with fewer solutions than the T_1 estimation method. The results above also confirm our analysis showning that, if too few solutions are used to estimate T_1, then the T_1 method delivers pessimistic predictions. Furthermore, our results also show that the margin by which the T_k method outperforms the T_1 method is determined by the speedup available on the number of processors k used to formulate the prediction.

The estimation methods presented are useful for determining the parallel resources required by the robot motion planning system. The resources necessary to deliver a particular level of performance can be determined as follows. Select a reasonable number of typical reaches in the given environment. Obtain an estimate of how many processors are necessary to obtain acceptable performance for each instance and select the minimum number of processors for all reaches attempted. If no solution can be found in the desired amount of time on the number of processors available, the method is also useful for indicating whether the desired performance can be obtained by adding more processors.

Acknowledgements

We would like to thank Jean Claude Latombe at Stanford University for providing access to implementations of the Random Path Planner; Curtis Olson for porting the software to a network of Unix/Linux workstations, and for writing visualization and animation tools; the University of Minnesota Army High Performance Computing Research Center for supporting the work.

References

Alanberg-Navony, N., Itai, A., and Moran, S.: 1994, Average and randomized complexity of distributed problems, Technical Report, Technion, Haifa, Israel.

Amato, N. M. and Dale, L. K.: 1999, Probabilistic roadmap methods are embarrassingly parallel, in: *Proc. IEEE Internat. Conf. on Robotics and Automation*, pp. 688–694.

Challou, D., Boley, D., Gini, M., Kumar, V., and Olson, C.: 1998, Parallel search algorithms for robot motion planning, in: K. Gupta and A. del Pobil (eds), *Practical Motion Planning in Robotics: Current Approaches and Future Directions*, Wiley, New York, pp. 115–131.

Challou, D., Gini, M., and Kumar, V.: 1993, Parallel search algorithms for robot motion planning, in: *Proc. of IEEE Internat. Conf. on Robotics and Automation*, Vol. 2., pp. 46–51.

Cook, D. J. and Varnell, R. C.: 1998, Adaptive parallel iterative deepening search, *J. Artificial Intelligence Res.* **9**, 139–166.

Corporation, T. M.: 1992, The connection machine CM-5 Technical Summary, Thinking Machines Corporation, Cambridge, MA.

Ertel, W.: 1992, OR-parallel theorem proving with random competition, in: A. Voronokov (ed.), *LPAR'92: Logic Programming and Automated Reasoning*, Lecture Notes in Artificial Intelligence 624, Springer, Berlin, pp. 226–237.

Ertel, W.: 1993, Massively parallel search with random competition, in: *Working Notes of the 1993 AAAI Spring Symposium for Innovative Applications of Massive Parallelism*, pp. 62–69, AAAI Press, Menlo Park, CA; available as Technical Report No. TR SS-93-04.

Ferreira, A. and Pardalos, P. (eds): 1996, *Solving Combinatorial Optimization Problems in Parallel: Methods and Techniques*, Lecture Notes in Computer Science 1054, State-of-the-Art Surveys, Springer, New York.

Grama, A. and Kumar, V.: 1999, State of the art in parallel search techniques for discrete optimization problems, *IEEE Trans. Knowledge Data Engrg.* **11**(1), 28–35.

Henrich, D., Wurrl, C., and Woern, H.: 1998, Multi-directional search with goal switching for robot path planning, in: A. P. del Pobil, J. Mira and M. Ali (eds), *Tasks and Methods in Applied Artificial Intelligence*, Lecture Notes in Artificial Intelligence 1416, Springer, Berlin, pp. 75–84.

Hoel, P., Port, S., and Stone, C.: 1971, *Introduction to Probability Theory*, Houghton Mifflin Company, Boston, MA.

Hoos, H. H.: 1998, Stochastic Local Search – Methods, Models, Applications, PhD Thesis, the Darmstadt University of Technology, Germany.

Hoos, H. H. and Stützle, T.: 1998, Evaluating Las Vegas algorithms – Pitfalls and remedies, in: *Proc. of the 14th Conf. on Uncertainty in Artificial Intelligence*, pp. 238–245, Morgan Kaufmann, Los Altos, CA.

Hsu, D., Latombe, J., Motwani, R., and Kavraki, L.: 1999, Capturing the connectivity of high-dimensional geometric spaces by parallelizable random sampling techniques, in: P. Pardalos and S. Rajasekaran (eds), *Advances in Randomized Parallel Computing*, Combinatorial Optimization Series, Kluwer Academic Publishers, Dordrecht, pp. 159–182.

Janakiram, V., Agrawal, D., and Mehrotra, R.: 1988, A randomized parallel backtracking algorithm, *IEEE Trans. Computers* **37**(12), 1665–1675.

Karp, R. and Zhang, Y.: 1993, Randomized parallel algorothms for backtrack search and branch -and-bound computation, *J. ACM* **40**(3), 765–789.

Kavraki, L. E. and Latombe, J. C.: 1998, Probabilistic roadmaps for robot path planning, in: K. Gupta and A. del Pobil (eds), *Practical Motion Planning in Robotics: Current Approaches and Future Directions*, Wiley, New York, pp. 33–53.

Kim, S. W. and Boley, D.: 2001, Building and navigating a network of local minima, *J. Robotic Systems* **18**(8), 405–419.

Latombe, J. C.: 1991, *Robot Motion Planning*, Kluwer Academic Publishers, Norwell, MA.

Li, G.-J. and Wah, B. W.: 1986, Coping with anomalies in parallel branch-and-bound algorithms, *IEEE Trans. Computers* **35**.

Mehrotra, R. and Gehringer, E. F.: 1985, Superlinear speedup through randomized algorithms, in: *Proc. of Internat. Conf. on Parallel Processing*, pp. 291–300.

Rao, V. N. and Kumar, V.: 1993, On the efficiency of parallel backtracking, *IEEE Trans. Parallel Distributed Systems* **4**(4), 427–437.

Reeves, C. R.: 1993, *Modern Heuristic Techniques for Combinatorial Problems*, Wiley, New York.

Reif, J.: 1979, Complexity of the Mover's problem and generalizations, in: *Proc. of IEEE Symposium on Foundations of Computer Science*, pp. 421–427.