Parallel Shortest Paths Methods for Globally Optimal Trajectories *

D.P. Bertsekas[a], F. Guerriero[b] and R. Musmanno[b]

[a]Laboratory for Information and Decision Systems, M.I.T.,
Cambridge, MA, 02139, U.S.A.

[b]Dipartimento di Elettronica, Informatica e Sistemistica,
Universita' della Calabria, 87036 Rende, Italy

In this paper we consider a special type of trajectory optimization problem that can be viewed as a continuous-space analog of the classical shortest path problem. This problem is approached by space discretization and solution of a discretized version of the associated Hamilton-Jacobi equation. It was recently shown by Tsitsiklis [1] that some of the ideas of classical shortest path methods, such as those underlying Dijkstra's algorithm, can be applied to solve the discretized Hamilton-Jacobi equation. In more recent work, Polymenakos, Bertsekas, and Tsitsiklis [2] have carried this analogy further to show that some efficient label correcting methods for shortest path problems, the SLF and SLF/LLL methods of [3] and [4], can be fruitfully adapted to solve the discretized Hamilton-Jacobi equation. In this paper we discuss parallel asynchronous implementations of these methods on a shared memory multiprocessor, the Alliant FX/80. Our results show that these methods are well suited for parallelization and achieve excellent speedup.

1. INTRODUCTION

We consider a trajectory optimization problem that arises in a variety of contexts involving the planning of a motion within a, perhaps irregular, region of two-dimensional or three-dimensional space. A vehicle starts at an initial point x_0 located in some region G of $\Re^m$ and follows a trajectory $x(t)$ such that

$$dx/dt = u(t), \tag{1}$$

where $u(t) \in \Re^m$ is a control vector that must satisfy the constraint $\|u(t)\| \leq 1$ for all t. After a certain time T the vehicle arrives at the boundary ∂G of G where a cost $q(x(T))$ is incurred. There is also a traveling cost $\int_0^T r(x(t))dt$ that depends on the trajectory followed by the vehicle. The objective is to find a trajectory $x(t)$ that starts at the given initial point $x(0)$, ends at the boundary ∂G of G, and minimizes

$$q(x(T)) + \int_0^T r(x(t))dt, \tag{2}$$

*Research supported by National Science Foundation under Grants 9108058-CCR, 9221293-INT, and 9300494-DMI

subject to the constraint $\|u(t)\| \leq 1$ for all t.

We assume that $\inf_{x \in G} r(x) > 0$, which forces the vehicle to reach the boundary of G in finite time. Note that this problem formulation includes the case where we want to reach a given destination point x_f with minimum traveling cost $\int_0^T r(x(t))dt$; we may just take $G = \Re^m - \{x_f\}$ and $q(x_f) = 0$. More generally, we can use the terminal cost $q(x)$ to provide a preference for reaching some portions of the boundary of G over others. In particular, points of the boundary of G with a very high value of $q(x)$ are essentially "forbidden". This allows the introduction of "obstacles" that the vehicle must avoid.

Note that when $r(x(t))$ is identically equal to 1, when G contains several obstacles, and when there is a fixed destination x_f, the problem becomes the shortest path problem of finding a minimum length trajectory that starts at x_0 ends at x_f and avoids the obstacles. This problem has been extensively studied in the robotics and theoretical computer science literature. More generally, the integral cost $\int_0^T r(x(t))dt$ may be viewed as a "generalized length" of the trajectory, and the problem may be viewed as a continuous-space shortest path problem.

Our problem can be approached by classical continuous-time optimal control techniques. However, when the region G contains many "obstacles" that the vehicle must avoid, and/or the cost functions $q(x)$ and $r(x)$ are nonconvex, the problem may be essentially combinatorial and may have multiple local minima. In this case, solution methods based on dynamic programming and discretization of the associated Hamilton-Jacobi equation, which provide globally optimal solutions, are typically preferable.

There is an interesting general method to address the discretization issues of continuous-time optimal control. The main idea in this method is to discretize, in addition to time, the state space $\Re^m$ using some kind of grid, and then to approximate the cost-to-go $V(x)$ of nongrid states by linear interpolation of the cost-to-go values of the nearby grid states. By this we mean that if a nongrid state x is expressed as

$$x = \sum_{i=1}^{m} \xi^i x^i \tag{3}$$

in terms of the grid states $x^1, \ldots, x^m$, where the positive weights ξ^i add to 1, then the cost-to-go of x is approximated by

$$\sum_{i=1}^{m} \xi^i V(x^i), \tag{4}$$

where $V(x^i)$ is the cost-to-go of x^i. When this idea is worked out, one ends up with a *stochastic* optimal control problem having as states the finite number of grid states, and transition probabilities that are determined from the weights ξ^i above. If the original continuous-time optimal control problem has fixed terminal time, the resulting stochastic control approximation has finite horizon. If the terminal time of the original problem is free and subject to optimization, as in the problem of this paper, the stochastic control approximation becomes a Markovian Decision Problem, known as a *first passage* or *stochastic shortest path problem* (see [5] and [6] for a discussion of such problems). We refer to the monograph [7], the papers [8] and [9], and the survey [10] for a description and analysis of continuous-time optimal control discretization issues.

Tsitsiklis [1] showed that the problem of this paper, when appropriately discretized, maintains much of the structure of the classical shortest path problem on directed graphs. In particular, a finitely terminating adaptation of the Dijkstra shortest path algorithm was developed in [1]. In [2], other shortest path methods were adapted to the problem of this paper. These adaptations are extensions to the SLF label correcting method of [3] and the SLF/LLL method of [4]. Computational results given in [3] show that these adaptations vastly outperform (by orders of magnitude) the classical dynamic programming methods, which are based on Jacobi and Gauss-Seidel value iterations.

In the present paper we focus on the efficient parallelization of the Dijkstra, and SLF/LLL methods of [3] and [4] on a shared memory machine. We draw motivation from our earlier work on parallel asynchronous label correcting methods, where we found that the SLF and SLF/LLL approaches lend themselves well to parallel computation, and result in very efficient shortest path solution methods. We concentrate on asynchronous algorithms. Generally, dynamic programming iterations, which contain label correcting methods as a special case, can be executed in a totally asynchronous fashion, as shown in [16]. The textbook [6] contains an extensive discussion and analysis of asynchronous algorithms, including theory that establishes the validity of the asynchronous implementations of the present paper.

Our computational results suggest that the Dijkstra and SLF/LLL approaches are well suited for parallel solution of the problem, and result in excellent speedup. The SLF/LLL method is faster for the problems that we tried, both in a serial and in a parallel environment. This is consistent with the serial computational results of [2]. Since the solution of the problem of this paper is very computationally intensive, we conclude that the gains from parallelization can be significant.

The remainder of the paper is organized as follows. In Section 2 we formulate the trajectory optimization problem, and we describe the discretized version of the Hamilton-Jacobi equation. In Section 3 we describe the Dijkstra method and the SLF/LLL method. In Section 4 we present parallel asynchronous implementations of these methods and computational results.

2. PROBLEM FORMULATION

A *trajectory* starting at $x_0 \in G$ is a continuous function $x : [0,T] \in \Re^m$, where T is some positive scalar, such that $x(t) \in G$, for all $t \in [0,T)$ and $x(T) \in \partial G$. The trajectory is said to be *admissible* if there exists a function $u : [0,T] \to \Re^m$ such that:

$$x(t) = x(0) + \int_0^t u(s)ds, \tag{5}$$

and

$$\|u(t)\| \leq 1, \qquad \forall\, t \in [0,T]. \tag{6}$$

Let G a bounded connected open subset of $\Re^m$ and let ∂G be its boundary. Let also $r : G \to (0,\infty)$ and $q : \partial G \to (0,\infty)$ be two positive-valued cost functions. The cost of an admissible trajectory is given by

$$q(x(T)) + \int_0^T r(x(t))dt. \tag{7}$$

The optimal cost-to-go function $V^* : G \cup \partial G \to \Re$ is defined as follows. If $x \in \partial G$, we let $V^*(x) = q(x)$, otherwise, if $x \in G$, we let $V^*(x)$ be the infimum of the costs of all admissible trajectories starting at x.

The cost-to-go function V^*, under appropriate conditions [11], satisfies the Hamilton-Jacobi equation

$$V^*(x) = \min_{u \in \Re^m,\ \|u\| \leq 1} r(x) + u' \nabla V^*(x), \qquad x \in G. \tag{8}$$

A discretized version of this equation was given and analyzed in [1]. We follow closely the framework of that reference.

Let $h > 0$ some discretization step. Let S and B be discretizations of the sets G and ∂G, whose elements are of the form (ih, jh), where i and j are integers. For each point $x \in S$, we denote by $N(x)$ the set of neighbors of x defined by

$$N(x) = \{x + h\alpha_i e_i \in S \cup B \mid i \in \{1, ..., m\}, \alpha_i \in \{-1, 1\}\}, \tag{9}$$

where e_i is the i-th unit vector of $\Re^m$.

We assume that we have two functions $g : S \to (0, \infty)$ and $f : B \to (0, \infty)$, which represent the discretizations of the traveling and terminal cost functions r and q, resopectively. The function g can usually be defined by $g(x) = r(x)$ for every $x \in S$. The choice of f may depend on the nature of θG because B can be disjoint from θG even if B is a good approximation to θG. We also introduce a function $V : S \cup B \to \Re$, which represents an approximation of the optimal cost-to-go function V^*.

The discretized Hamilton-Jacobi equation (8) is given by

$$V(x) = \min_{\alpha \in \{-1,1\}^m} \min_{\theta \in \Theta} \left[hg(x)\|\theta\| + \sum_{i=1}^{m} \theta_i V(x + h\alpha_i e_i) \right], \qquad x \in S, \tag{10}$$

$$V(x) = f(x), \qquad x \in B, \tag{11}$$

where:

$$\|\theta\| = \sqrt{\sum_{i=1}^{m} \theta_i^2}, \qquad \theta \in \Theta, \tag{12}$$

and Θ is the unit simplex in $\Re^m$,

$$\Theta = \left\{ \theta \in \Re^m \mid \sum_{i=1}^{m} \theta_i = 1,\ \theta_i \geq 0 \right\}. \tag{13}$$

The manner in which Eqs. (10) and (11) approximate the Hamilton-Jacobi equation is explained in [1]. In particular, consider a vehicle that starts at some point $x \in S$ and moves with a unit speed along a direction d, until the point $x + h \sum_{i=1}^{m} \theta_i \alpha_i e_i$ is reached. The direction d is determined by α, which specifies the quadrant within which d lies, and by the choice of θ, which specifies the direction of motion within that quadrant. The total time required to reach the final point is $h\|\theta\|$. The traveling cost is $hg(x)\|\theta\|$, since $g(x)$

is the traveling cost per unit time. With these approximations, it is seen that the optimal cost-to-go function $V^*(x)$ satisfies

$$V^*(x) \approx \min_{\alpha \in \{-1,1\}^m} \min_{\theta \in \Theta} \left[hg(x)\|\theta\| + V^* \left(x + h \sum_{i=1}^m \theta_i \alpha_i e_i \right) \right]. \tag{14}$$

Now if in the above equation we use the approximation

$$V^* \left(x + h \sum_{i=1}^m \theta_i \alpha_i e_i \right) \approx \sum_{i=1}^m \theta_i V^*(x + h\alpha_i e_i), \tag{15}$$

we obtain Eq. (10).

The equations (10) and (11) represent a special case of discretization based on finite elements. In [7], these equations are viewed as Dynamic Programming Equations for a Markov Decision Problem, which can be solved by using methods like value or policy iteration. For a more detailed description of these methods see [7] and [9]. However, as remarked in [1] and [2], these methods do not exploit the special structure of our problem and are relatively slow. In the next section we discuss methods that are much more efficient.

3. LABEL CORRECTING METHODS

The analysis and methodology of [1] and [2] rests on the following fundamental lemma:

Lemma 3.1 *Let $x \in S$, and let $\alpha \in \{-1,1\}^m$, $\theta \in \Theta$, be such that $V(x) = hg(x)\|\theta\| + \sum_{i=1}^m V(x + h\alpha_i e_i)$. Then, $V(x + \alpha_i e_i) < V(x)$, for all i such that $\theta_i > 0$.*

This lemma, which is proved in [1], can be used to show that the prototype label correcting algorithm to be described shortly *terminates in a finite number of iterations* (see [2]). This algorithm maintains a vector $V(x)$ of labels, where $x \in S$, and a candidate list L of states. At the start of the algorithm, the list L contains just an element x_1 of B at which $f(x)$ is minimized, that is:

$$V(x_1) \leq V(x), \qquad \forall\ x \in S \cup B. \tag{16}$$

Also the initial labels are given by

$$\bar{V}(x) = \begin{cases} V(x) & \forall x \in L \cup B, \\ \infty & \text{otherwise.} \end{cases} \tag{17}$$

The algorithm terminates when L is empty and upon termination the optimal cost-to-go of x is given by $\bar{V}(x)$. Assuming L nonempty at a typical iteration, the vector of labels and the candidate list are updated as follows:

1. Let y be a state in L.
   Remove y from L;

2. For each $x \in N(y)$, where
   $N(y) = \{y + h\alpha_i e_i \mid i \in \{1, ..., m\}, \alpha_i \in \{-1, 1\}\}$, compute
   $\hat{V}(x) = \min_\alpha \min_\theta \left[hg(x)\|\theta\| + \sum_{i=1}^m \theta_i \bar{V}(x + h\alpha_i e_i)\right]$.
   If $\hat{V}(x) < \bar{V}(x)$, set
   $\bar{V}(x) = \hat{V}(x)$,
   and add x to L if x does not already belong to L.

The analog of Dijkstra's method is obtained when the state y exiting L is the state with the minimum value of $\bar{V}$. Reference [1] proves the remarkable fact that in this method, each state will enter and exit L at most once. An efficient way to implement Dijkstra's algorithm is to maintain the list L partially ordered in a binary heap.

If the state exiting the candidate list at each iteration is not a state with minimum label, the required number of iterations can be shown to be finite under our assumptions (see [2]), but some states many enter and exit the candidate list more than once. However, such a method avoids the overhead associated with finiding the node of minimum label. A particularly effective strategy for selecting the state to exit the candidate list was proposed in [3] in the context of the classical shortest path problem. This strategy, called Small Label First method (SLF for short), maintains the candidate list in a double-ended queue Q and inserts a node to the bottom or to the top of Q depending on whether the label of the node is larger than the label of the top node of Q or not. In [2], this method was adapted to the trajectory optimization problem as follows:

1. Let x be a state that enters Q.
   Let y be the top state of Q.

2. If $\bar{V}(x) \leq \bar{V}(y)$ then insert x at the top of Q,
   else insert x at the bottom of Q.

The state removed from L at each iteration is always the top state of Q. In [4], a more sophisticated state removal strategy, called Large Label Last strategy (LLL for short), is proposed for the classical shortest path problem. In this strategy, the top state of Q is repositioned to the bottom whenever its label is larger than the average node label in Q. In [2], this method was adapted to the trajectory optimization problem as follows:

1. Let $s = \frac{\sum_{x \in Q} \bar{V}(x)}{\|Q\|}$.
   Let y be the top state of Q.

2. If $\bar{V}(y) > s$ then move y at the bottom of Q.
   Repeat until a state y such that $\bar{V}(y) \leq s$ is found and is removed from Q.

The SLF and LLL strategies have also been combined to solve the trajectory optimization problem in [2]. The serial implementations of the methods of the present section, given in [2], have served as the starting point for the parallel implementations described in the next section.

4. PARALLELIZATION

In this section, we discuss our parallelization of the Dijkstra, SLF, and combined SLF/LLL methods described in the preceding section. Our implementations are similar to the corresponding ones described in [4] for the classical shortest path problem. The prototype label correcting method of the preceding section can be easily parallelized at least for shared memory machines. The basic idea is that several states can be simultaneously extracted from the candidate list and the labels of adjacent states can be updated in parallel. On a shared memory multiprocessor, the label of each state is stored in a unique memory location, shared among all processors. This means that when more processors try to modify simultaneously the label of the same state, they must lock the corresponding memory location to guarantee that only one processor at a time modifies the label of that state. We assume the availability of p processors. For the SLF/LLL method, we have p queues shared among the processors. Each processor i uses only the i-th queue when the LLL state removal strategy is applied and one of the p queues when a state has to be inserted according to the SLF state insertion strategy. In particular, each processor extracts the state x from the top of its queue (or moves it to the bottom of the queue, following the LLL procedure), updates the labels for the adjacent states, and uses a heuristic procedure for choosing the queue to insert a state that enters L. This queue is chosen on the basis of the minimum current value for the sum of states assigned to the queues. As remarked in [4], this heuristic is very easy to implement and ensures a good load balancing among the processors.

In order to parallelize the Dijkstra version of the label correcting method, we maintain a separate binary heap for each processor. Each processor extracts the state at the top of its own binary heap, and whenever a state must be inserted in L, the appropriate binary heap is chosen according to the same heuristic procedure used for the SLF/LLL method. Furthermore, when a processor updates the label of a state already present in L, the same processor reorganizes the corresponding binary heap, in order to keep it ordered.

Even though all the p binary heaps are ordered, the entire list L is not fully sorted. This means that in our parallel implementation of Dijkstra's method a state may exit and reenter L several times in the course of the algorithm. Nonetheless, we refer to this parallel method as the parallel Dijkstra's method, even though the corresponding parallel version is not trully a label setting algorithm. It is worth observing that the overhead for inserting and deleting a state in a binary heap strongly depends on the number of states in the binary heap. This means that by using multiple binary heaps in the parallelization scheme, this overhead is substantially reduced.

In our parallel implementations, both Dijkstra's and the SLF/LLL methods are executed asynchronously, in the sense that a new state may be removed from the list L by some processor while other processors are still updating the labels of other states. For a more detailed discussion on parallel asynchronous iterative methods see [6]. More formally, let $\bar{V}(x,t)$, $t = 0, 1, \ldots$ be the value of the label of state x at time t. This is the value of $\bar{V}(x)$ which is kept in the shared memory location. The label $\bar{V}(x,t)$ is updated at a subset of times $T(x) \subset \{0, 1, ...\}$ by some processor by using the following formula:

$$\bar{V}(x,t+1) = \min_\alpha \min_\theta \Big[hg(x)\|\theta\| + \textstyle\sum_{i=1}^m \theta_i \bar{V}(x + h\alpha_i e_i, \tau_i(x,t)) \Big], \quad \text{if } t \in T(x), \qquad (18)$$

$$\bar{V}(x,t+1) = \qquad \bar{V}(x,t), \qquad \text{otherwise.} \qquad (19)$$

In this formula $\tau_i(x,t))$ represents the time at which the label $\bar{V}(x+h\alpha_i e_i)$ has been read from shared memory by the processor updating $\bar{V}(x)$ at time t. The asynchronism is due to the fact that we may have $\tau_i(x,t) < t$ and $\bar{V}(x+h\alpha_i e_i, \tau_i(x,t)) \neq \bar{V}(x+h\alpha_i e_i, t)$.

The convergence of the algorithms can be shown under very weak assumptions. The proof closely resembles the proofs given in [6], Section 6.4, and it will not be given here.

Dijkstra's and the SLF/LLL methods and its parallel asynchronous versions have been implemented and tested on an Alliant FX/80, a vector-parallel computer with 8 processors, each with 23 Mflops of peak performance, having a core memory of 32 MBytes. All the codes are written in Fortran and compiled with the FX/Fortran 4.2 compiler.

We have considered two different sets of randomly generated test problems, which are 2-D and 3-D grids obtained from discretization of a square and of a cube with sides of length that is a multiple of the discretization step h. This ensures that the distance between any two adjacent states in the same direction is always h. We have used a similar approach for generating our test problem to that used in [2]. G is the set of the interior points of the square or the cube, whereas ∂G is the set of the points on the border. S and B are the states on the grids. The values of g in all points in S are randomly generated, according to a uniform distribution in the range [1,1000]. The cost f of all border states is assumed to be infinity, except for two adjacent states in a corner of the square or of the cube. In order to consider test problems that are more realistic, we add some *obstacles*, that is, we assume that $g(x) = \infty$ for some $x \in S$. It is easy to show that these types of problems allow the use of Dijkstra's and the SLF/LLL methods to find the optimal cost-to-go from all interior points to a point on the border B.

The full list of all test problems is reported in Table 1. The percentage of obstacles listed in this table is the fraction of the number of states x in S for which $g(x) = \infty$.

Table 1. List of test problems

Problem	states	Percentage of Obstacles
2-D.1	500x500	0
2-D.2	500x500	0.05
2-D.3	500x500	0.10
2-D.4	500x500	0.15
2-D.5	500x500	0.20
2-D.6	750x750	0
2-D.7	750x750	0.05
2-D.8	750x750	0.10
2-D.9	750x750	0.15
2-D.10	750x750	0.20
2-D.11	1000x1000	0
2-D.12	1000x1000	0.05
2-D.13	1000x1000	0.10
2-D.14	1000x1000	0.15
2-D.15	1000x1000	0.20
3-D.1	25x25x25	0
3-D.2	25x25x25	0.05
3-D.3	25x25x25	0.10
3-D.4	25x25x25	0.15
3-D.5	25x25x25	0.20
3-D.6	50x50x50	0
3-D.7	50x50x50	0.05
3-D.8	50x50x50	0.10
3-D.9	50x50x50	0.15
3-D.10	50x50x50	0.20
3-D.11	75x75x75	0
3-D.12	75x75x75	0.05
3-D.13	75x75x75	0.10
3-D.14	75x75x75	0.15
3-D.15	75x75x75	0.20

The numerical results are collected in 4 different tables, reported below, one for each algorithm and category of test problems (2-D or 3-D grid problems). In these tables, time in secs, and number of iterations required to solve the test problems is reported for each algorithm. For the parallel methods we report also the speed-up values, defined by the ratio between the sequential and parallel execution time.

Table 2. Results of Dijkstra's method for 2-D grid problems

Problem	Sequential	Parallel	Speed-up
2-D.1	248004 / 97.03	254300 / 18.79	5.16
2-D.2	235603 / 88.08	243308 / 17.45	5.05
2-D.3	223169 / 79.68	226118 / 19.76	4.03
2-D.4	210649 / 71.53	213609 / 18.09	3.95
2-D.5	197959 / 63.75	200781 / 16.42	3.88
2-D.6	559504 / 225.71	565726 / 61.08	3.70
2-D.7	531524 / 205.08	537100 / 50.42	4.07
2-D.8	223169 / 79.55	234376 / 16.29	4.88
2-D.9	475314 / 166.91	480871 / 42.65	3.91
2-D.10	446669 / 149.23	451594 / 39.06	3.82
2-D.11	996004 / 410.33	1011577 / 80.56	5.09
2-D.12	946201 / 372.91	966309 / 75.67	4.93
2-D.13	896302 / 337.44	917883 / 69.55	4.85
2-D.14	846122 / 304.09	873970 / 65.07	4.67
2-D.15	795138 / 272.09	876824 / 62.13	4.38

Table 3. Results of SLF/LLL method for 2-D grid problems

Problem	Sequential	Parallel	Speed-up
2-D.1	283062 / 71.59	273319 / 14.35	4.99
2-D.2	266881 / 63.37	259438 / 13.12	4.83
2-D.3	249373 / 55.52	244863 / 15.24	3.64
2-D.4	232731 / 48.50	213609 / 13.81	3.51
2-D.5	216077 / 42.01	213939 / 12.47	3.37
2-D.6	640159 / 162.43	617794 / 32.63	4.98
2-D.7	601712 / 143.15	584866 / 37.87	3.78
2-D.8	249373 / 55.40	245057 / 11.88	4.66
2-D.9	526395 / 109.89	537960 / 26.44	4.16
2-D.10	489954 / 95.70	483180 / 29.79	3.21
2-D.11	1144104 / 291.05	1102015 / 62.95	4.62
2-D.12	1072272 / 255.82	1045384 / 58.12	4.40
2-D.13	1004776 / 224.80	985675 / 54.12	4.15
2-D.14	936121 / 196.13	946082 / 53.09	3.69
2-D.15	869957 / 170.54	861412 / 44.82	3.80

Table 4. Results of Dijkstra's method for 3-D grid problems

Problem	Sequential	Parallel	Speed-up
3-D.1	12168 / 119.92	14664 / 17.64	6.80
3-D.2	11560 / 108.26	14086 / 16.00	6.77
3-D.3	10951 / 92.56	13224 / 13.13	7.05
3-D.4	10342 / 76.58	12604 / 11.67	6.56
3-D.5	9733 / 64.37	11672 / 9.38	6.86
3-D.6	110593 / 1186.40	130801 / 167.56	7.08
3-D.7	105063 / 1003.41	124102 / 149.17	6.73
3-D.8	99534 / 874.61	118458 / 122.81	7.12
3-D.9	94002 / 725.35	111061 / 102.86	7.05
3-D.10	88464 / 626.08	104073 / 85.33	7.34
3-D.11	389018 / 4283.12	445048 / 590.98	7.25
3-D.12	369567 / 3649.76	420662 / 506.28	7.21
3-D.13	350116 / 3121.77	393070 / 437.26	7.14
3-D.14	330657 / 2619.21	386584 / 371.81	7.04
3-D.15	311176 / 2201.13	357460 / 307.05	7.17

Table 5. Results of SLF/LLL method for 3-D grid problems

Problem	Sequential	Parallel	Speed-up
3-D.1	20007 / 232.70	18635 / 19.87	11.71
3-D.2	18654 / 204.66	18390 / 17.16	11.93
3-D.3	17929 / 178.14	16881 / 13.76	12.95
3-D.4	17219 / 154.58	16132 / 9.94	15.55
3-D.5	14714 / 115.20	14673 / 9.58	12.02
3-D.6	180871 / 2194.08	173213 / 159.84	13.73
3-D.7	173189 / 1930.34	159966 / 138.56	13.93
3-D.8	152481 / 1505.92	151203 / 123.76	12.17
3-D.9	143647 / 1278.71	141053 / 89.55	14.28
3-D.10	130514 / 1030.12	126106 / 93.76	10.99
3-D.11	630325 / 7852.97	551956 / 624.99	12.56
3-D.12	592540 / 6639.18	528597 / 523.56	12.68
3-D.13	525829 / 5265.43	503412 / 473.25	11.13
3-D.14	499068 / 4513.38	497054 / 303.18	14.86
3-D.15	444179 / 3506.93	457312 / 292.57	11.99

In Tables 6 and 7 we aim to summarize the performance of the various methods. In particular, we compare the methods following an approach that is similar to the one proposed in [15], by giving to each method and for each test problem, a score that is equal to the ratio of the execution time of this method over the execution time of the fastest method for the given problem. Thus, for each method, we obtain an average score, which is the ratio of the sum of the scores of the method over the number of test problem. This average score, given in Tables 6 and 7, indicates how much a particular method has been slower on the average than the most successful method.

Table 6. Ranking on 2-D grid problems

Algorithm	Number of Processors	Performance Index
DIJKSTRA	1	6.02
	8	1.37
SLF/LLL	1	4.12
	8	1.00

Table 7. Ranking on 3-D grid problems

Algorithm	Number of Processors	Performance Index
DIJKSTRA	1	7.34
	8	1.05
SLF/LLL	1	13.26
	8	1.04

Note that for 2-dimensional problems, the SLF/LLL method is faster than Dijkstra's method in a sequential environment, despite the smaller number of iterations of Dijkstra's method. This is due to the extra overhead for finding a state with minimal label in Dijkstra's method. For 3-dimensional problems, however, each iteration is much more costly than for 2-dimensional problems, and as a result the sequential Dijkstra's method is faster than the sequential SLF/LLL method. Nonetheless the parallel versions of the two methods are competitive for 3-dimensional problems because the SLF/LLL method exhibits greater speedup in our experiments. Furthermore, the parallel version of Dijkstra's method requires substantially more iterations than its serial counterpart. The superlinear speedup of the parallel SLF/LLL method is somewhat unexpected. It is due to the fact that the computation of the cost-to-go function in the 3-dimensional case often simplifies to a 2-dimensional computation. We have experimentally observed that this simplification occurs much more frequently in the parallel SLF/LLL method than in its sequential version.

REFERENCES

1. J. N. Tsitsiklis, Efficient Algorithms for Globally Optimal Trajectories, Tech. Rep. LIDS-P-2210, Laboratory for Information and Decision Systems, M.I.T., October 1993.
2. L. C. Polymenakos, D. P. Bertsekas, and J. N. Tsitsiklis, Extensions and Implementations of Efficient Algorithms for Globally Optimal Trajectories, unpublished report, Laboratory for Information and Decision Systems, M.I.T., Cambridge, Ma, U.S.A., 1994.
3. D. P. Bertsekas, A Simple and Fast Label Correcting Algorithm for Shortest Paths, Networks, 23, 703-709, 1993.
4. D. P. Bertsekas, F. Guerriero, and R. Musmanno, Parallel Asynchronous Label Correcting Methods for Shortest Paths, Journal of Optimization Theory and Applications, 1995 (*toappear*).
5. D. P. Bertsekas, and J. N. Tsitsiklis, An Analysis of Stochastic Shortest Path Algorithms, Math. Operations Res., Vol. 16, pp. 580-595, 1991.
6. D. P. Bertsekas, and J. N. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods, Prentice-Hall, Englewood Cliffs, N.J., 1989.
7. H. J. Kushner, and P. G. Dupuis, Numerical Methods for Stochastic Control Problems in Continuous Time, Springer-Verlag, New-York, 1992.
8. M. Falcone, Numerical approach to the infinite horizon problem of deterministic control theory, Applied Mathematics and Optimization, Vol. 15 (1), pp. 1-13, 1987.
9. R. Gonzales, and E. Rofman, On Deterministic Control Problems: an Approximation Procedure for the Optimal Cost, I, Stationary Problem, SIAM Journal on Control and Optimization, 23, pp. 242-266, 1985.
10. H. J. Kushner, Numerical methods for stochastic control problems in continuous time, SIAM Journal on Control and Optimization, Vol. 28 (5), pp. 999-1048, 1990.
11. W. Fleming, and R. Rishel, Deterministic and Stochastic Optimal Control, Springer-Verlag, New York, 1975.
12. R. Bellman, Dynamic Programming, Princeton University Press, Princeton, N.J., 1957.
13. F. Glover, R. Glover, and D. Klingman, The Threshold Shortest Path Algorithm, Networks, 14, 1986.
14. U. Pape, Implementation and Efficiency of Moore-Algorithms for the Shortest Path Problem, Mathematical Programming, 7, 212-222, 1974.
15. A. A. Brown, and M. C. Bartholomew-Biggs, Some Effective Methods for Unconstrained Optimization Based on the Solution of Systems of Ordinary Differential Equations, Tech. Rep. 178, Numerical Optimization Centre, The Hatfield Polytechnic, Hatfield, UK, 1987.
16. D. P. Bertsekas, Distributed Dynamic Programming, IEEE Transactions on Aut. Control, Vol. AC-27, pp. 610-616, 1982.