Pergamon

Transpn Res.-C, Vol. 5, No. 2, pp. 95–107, 1997

0968-090X/97 $17.00 + 0.00

PII: S0968-090X(97)00004-1

DESIGN AND IMPLEMENTATION OF PARALLEL TIME-DEPENDENT LEAST TIME PATH ALGORITHMS FOR INTELLIGENT TRANSPORTATION SYSTEMS APPLICATIONS

ATHANASIOS ZILIASKOPOULOS*
Department of Civil Engineering, Northwestern University, Evanston, IL 60208, U.S.A.

DIMITRIOS KOTZINOS
Foundation for Research and Technology Hellas, Institute of Computer Science, Heraklion, Greece

and

HANI S. MAHMASSANI
Department of Civil Engineering, The University of Texas at Austin, Austin, TX 78712-1076, U.S.A.

Abstract—The development of Intelligent Transportation Systems (ITS) and the resulting need for real-time traffic management and route guidance models require fast shortest-path algorithms that can account for the dynamics of traffic networks. The objective of this paper is to introduce parallel designs for time-dependent shortest-path algorithms that can be used in real-time ITS applications. In this paper, two shared-memory and one message-passing algorithms are designed, implemented, coded and computationally tested on actual and random networks. The reported tests are performed on CRAY supercomputers, but the algorithms can be readily ported to lower-end multiprocessor machines. © 1997 Elsevier Science Ltd

1. INTRODUCTION

The development of Intelligent Transportation Systems (ITS) and the resulting need for real-time traffic management and route guidance models has renewed the interest in shortest-path algorithms. One of ITS' main premises is to reduce a traveler's trip time by using current information about prevailing traffic conditions in the network; raw data are collected from the network, processed by a central controller or by an on-board computer and presented to the user in an easily understood form, usually a route. If the optimum paths are computed for individual users, stand-alone time-dependent least-time path (TDLTP) algorithms are needed to compute the path in reasonable time after the request. If coordinated paths are sought by a central controller for many users in the system, dynamic traffic assignment algorithms (DTA) must be employed. Most DTA algorithms require iterative computation of least-time paths on dynamic networks from all origins and departure times to all destination nodes at every assignment interval, which is computationally expensive. For instance, the DTA algorithms built by Mahmassani *et al.*, (1994) spend more than 70% of the overall CPU time on path computations, making TDLTP algorithms the bottleneck to any efforts for real-time implementation. Besides ITS operations, many other areas in telecommunications and logistics require fast-path computations that account for network dynamics (Kaufman and Smith, 1993; Orda and Rom, 1990). To the best of our knowledge, there appears to be no algorithm reported in the literature that can account for both of these ITS needs:

*Author for correspondence

time-dependency and potential for real-time execution. In fact, as discussed below, only a few successful parallel implementations have been reported even for the static shortest-path problem.

The objective of this paper was to introduce parallel designs, both shared memory and message-passing, for TDLTP algorithms that can be used in real-time ITS applications. The algorithms are based on the sequential TDLTP algorithm introduced by Ziliaskopoulos and Mahmassani (1993, 1996); they assume the existence of discrete travel times on each arc over a given time period of interest (e.g. peak period), discretized into small time steps. A vector of labels, one for each time interval, is maintained for every node and updated in a label-correcting fashion, i.e. the labels are upper bounds to the optimum path label until the algorithm terminates. The shared memory designs are implemented on a CRAY Y-MP/8, although no machine specific features are used and the algorithms are, in principle, implementable on any shared memory machine. The message-passing algorithm is implemented on a CRAY T3D with 32 processors arranged in a 3D-folded torus architecture, using Parallel Virtual Machine (PVM) functions for communication. PVM allows processing elements to communicate with one another by sending messages containing all the necessary information. A version of PVM exists for a network of UNIX workstations, which makes the algorithm portable to lower-end machines. The algorithms were tested on various street-like networks (average degree approximately 3); we limited the experiments to street-like networks because the primary purpose of this paper was to assess the applicability of the designed algorithms on ITS applications. No testing was performed on larger degree networks and no inferences can be made on the algorithms' performance on such networks.

In the next section, we briefly discuss parallel approaches proposed in the literature for shortest-paths algorithms. In Section 3, we discuss the TDLTP problem and include a sequential algorithm. Also, the shared memory algorithms are designed, implemented and computationally tested. Section 4 introduces the message-passing design and evaluates its performance computationally. Section 5 concludes this paper and provides pointers for further research in the area.

2. LITERATURE REVIEW

Designing parallel network optimization algorithms has attracted considerable attention in the last 15 years, parallel shortest-path (SP) algorithms being no exception. A number of algorithms, both abstract machine-independent (Tseng *et al.*, 1990; Paige and Kruskal, 1985) and machine-specific (Day and Srimani, 1989; Habbal *et al.*, 1994) have been proposed for various sequential SP approaches. The most investigated class of SP problems is the all-pairs problem (Habbal *et al.*, 1994; Paige and Kruskal, 1985; Deo *et al.*, 1980). Habbal *et al.* (1994) proposed a decomposition scheme for Floyd's algorithm suitable for massively parallel architectures and implemented it with good results on a Connection Machine. Several attempts have been reported to parallelize one-to-all SP algorithms, with little apparent success (Paige and Kruskal, 1985; Bertsekas and Tsitsiklis, 1989; Deo *et al.*, 1980). Adaptations of Moore's algorithm appear theoretically promising (Deo *et al.*, 1980; Lakhani, 1984) but, when implemented, the use of a common shared queue proved to be a bottleneck, limiting the achievable speed-up. Paige and Kruskal (1985) presented parallel versions of Dijkstra's and Ford's algorithms, deriving mainly theoretical bounds without, however, implementation and actual computational results. Regarding the TDLTP problem, the first paper dealing with this subject appears to be the one by Cooke and Hasley (1966) who proposed an approach based on Bellman's (1958) principle of optimality. A recent overview of this literature can be found in Ziliaskopoulos (1994). The algorithms introduced in this paper are based on a sequential scheme proposed by Ziliaskopoulos and Mahmassani (1993, 1996). The main characteristic of this algorithm is the simultaneous scanning of all labels of a node for all possible departure times. The main problem with parallelizing one-to-all single SP approaches is that they are sequential iterative approaches, where the number of iterations is large, but each iteration takes little time, yielding very small-size grains. One way to overcome this problem is to increase the size of the grain by performing more computations at every iteration. The proposed implementation achieves larger grains by computing the labels for all possible departure times for every node. Next, the formulation and a sequential algorithm for the TDLTP problem are discussed.

3. FORMULATION OF THE TDLTP PROBLEM AND A SEQUENTIAL ALGORITHM

Let $G = (V, E)$ be a directed graph, where V is the set of nodes and E the set of arcs connecting the nodes. Let $d_{ij}[t]$ be the non-negative time required to travel from node i to node j when departure from node i is at t; $d_{ij}[t]$ is defined for every $t \in S$, where $S = \{t_0, t_0 + \delta, t_0 + 2\delta, t_0 + 3\delta, \ldots, t_0 + M\delta\}$, t_0 is the earliest possible departure time from every origin node in the network, δ is a small time interval during which some meaningful change in traffic conditions may occur, and M is a large integer such that the interval from t_0 to $t_0 + M\delta$ spans the time period of interest (e.g. the morning peak period). We assume that $d_{ij}[t]$ for $t > t_0 + M\delta$ is constant and equal to $d_{ij}[t + M\delta]$. This is a reasonable assumption for urban traffic networks where, after the peak hour, somewhat stable conditions can be assumed. Nevertheless, this is not a restrictive assumption since M is user-defined and can always be increased to include the time period with variable travel times for all arcs. Another assumption is that $d_{ij}[\tau] = d_{ij}[t_0 + k\delta]$, $\forall \tau \in (t_0 + k\delta, t_0 + (k+1)\delta)$; this is not very restrictive either, because δ is specified by the user and it is by definition very small. Node N denotes the destination node.

The algorithms proposed in this paper calculate the time-dependent shortest paths from every node i in the network and for every departure time step t to node N. At each step of the computation, the travel time of the current shortest path between node N and node i at time t is denoted by $\lambda[t]$. Let $\Lambda_i = [\lambda_i[t_0], \lambda_i[t_0 + \delta], \lambda_i[t_0 + 2\delta], \ldots, \lambda_i[t_0 + M\delta]]$ be an M-vector label that contains all the labels for every time step t for node I. Every finite label $\lambda_i[t]$ from node i to node N is identified by the ordered set of nodes $P_i = \{i = n_1, n_2, \ldots, n_m = N\}$. According to Ziliaskopoulos and Mahmassani (1992, 1993, 1996), $\lambda_i[t]$ is defined by the following equation:

$$\lambda_i[t] = \begin{cases} \min_{j1}\{\lambda_i[[t + d_{ij}[t]] + d_{ij}[t]\}, & \text{for } i = 1, 2, \ldots, N-1;\ t \in S \\ 0, & \text{for } i = N;\ t \in S \end{cases} \tag{1}$$

The optimality eqn (1) is the building block of all the algorithms introduced in this paper. The TDLTP problem is extensively discussed in Ziliaskopoulos and Mahmassani (1993, 1996) and Ziliaskopoulos (1994). Next, we include the sequential algorithm for the TDLTP algorithm for reference.

3.1. The sequential TDLTP algorithm

This algorithm is essentially a label-correcting procedure. A scan-eligible (SE) list is created that maintains all the nodes with the potential to improve at least one label of any node in the network. Initially, it contains only the destination node N. The labels of node N are set equal to zero and those of the remaining nodes to infinity. In the first iteration, all nodes that can directly reach N are updated according to the optimality equation above, and inserted in the SE list.

$$\lambda_i[t] = d_{iN}[t], \forall i \in \Gamma^{-1}(N), \forall t \in S$$

where $\Gamma^{-1}(N)$ is the set of nodes such that $(i, N) \in E$.

Next, the first node j of the SE list is scanned, updating every node i that can directly reach node j ($\forall i \in \Gamma^{-1}(j)$) according to relation (1). If at least one of the components of Λ_i is modified, node i is inserted in the SE list. This scheme is repeated until the SE is empty, and the algorithm terminates. The steps of the TDLTP algorithm are as follows:

3.2. The TDLTP algorithm

Step 1: Initialize the label vectors at the following values:
$\Lambda_N = (0, 0, \ldots, 0)$
$\Lambda_i = (\infty, \infty, \ldots, \infty), \forall i \in \mathcal{V}\backslash N$
Create the SE list and initialize it by inserting into it the destination node N.

Step 2: If the SE list is empty, go to Step 4; otherwise, select the first node j from the SE list and delete it from the list.

Step 3: Scan node j as follows:
For every node $i \in \Gamma^{-1}(j)$ do the following:
For every time step $t \in S$ do the following:
Check if $\lambda_i[t]$ is greater than $d_{ij}[t] + \lambda_j[t + d_{ij}(t)]$.
If it is, then replace $\lambda_i(t)$ in the label vector Λ_i at position t with the new value.
If at least one of the M labels of node i has been improved, insert node i in the SE list. The details of the SE list structure, and the associated operations of creation, insertion, and deletion are described in Ziliaskopoulos and Mahmassani (1993, 1996).
If all nodes $i \in \Gamma^{-1}(j)$ have been updated, go to Step 2; otherwise, go back to Step 3.

Step 4. Terminate the algorithm. The M-dimensional vectors Λ_i for every node i in the network contain the travel times of the time-dependent least-time paths from node i to the destination node N for each time step $t \in S$.

The complexity of this algorithm is $O(M^2 \mid V \mid 3)$ (see Ziliaskopoulos, 1994). Next, based on the formulation and the sequential algorithm above, we present the proposed parallel TDLTP algorithms.

4. SHARED MEMORY PARALLEL ALGORITHMS

Two main factors contribute to the potential benefit of implementing an algorithm in parallel. The first is the computing hardware, which consists of the computing processors, the interconnecting architecture and the communication protocol among the different processors. The second factor is the design of the algorithm, which should take advantage of the hardware capabilities. The major obstacle in parallelizing an algorithm is to avoid computational and storage dependence among the various concurrent components of the algorithm. Computational dependence consists of data dependence and control dependence. Data dependence is an ordering relationship between statements that use or produce the same data. Control dependence refers to situations where the order of execution of statements cannot be determined *a priori*. This typically happens when conditional statements (IF-THEN commands) are encountered in the program. Finally, storage dependence is related to the independence of the workspace. Each parallel computational task has access to variables, and the fetching and storing of the variables in one task must not interfere with that in another task. Dependence is avoided by using locking functions to protect the potential shared areas of the program.

The primary development machine in this part of the study is a CRAY Y-MP/8 supercomputer, a multiple-instruction multiple-data (MIMD) machine with eight vector processors and shared memory communication environment. Several parallel processing capabilities are available on this machine, such as macrotasking, microtasking, autotasking, vectorization and I/O subsystem parallelization. Macrotasking is the only feature utilized in this paper, mainly to facilitate portability to lower end machines. Macrotasking is a form of multitasking that uses multiple processors at the subroutine level. The whole operation is controlled by the programmer who is responsible for explicitly partitioning his program into tasks, each of which is eligible to run on a central CPU. Typically, these tasks may take the form of different subroutines that can be executed concurrently, or involve separate invocations of the same subroutine. Task is a piece of code and data that can be scheduled for execution on a CPU.

Macrotasking is applied at two levels in this section: at the level of many destinations (program level–P1-TDLTP algorithm) and at the level of an individual destination (SE list level–P2-TDLTP algorithm). At the many destinations level, each destination is assigned to a different processor and the algorithm runs independently on every processor. At the single destination level, one instance of the algorithm for one destination runs on many processors.

Next, we discuss the design and implementation of the P1-TDLTP and P2-TDLTP algorithms.

4.1. Macrotasking at the program level (P1-TDLTP algorithm)

At the program level, we assume that the optimum paths from every node and time interval to several destination nodes are sought. The design idea is almost trivial: assign each destination to a different processor and run the TDLTP algorithm independently on each one of them. Repeat this

procedure until all the destinations have been addressed. Each task is completely independent from the others, which makes this design attractive. Every processor keeps a copy of the program, and maintains its own SE list and optimum path tree — stored in the private memory of the processor. The only time the shared memory is accessed is to obtain travel-time data. This step, theoretically, does not interfere with the other processors, because the data are accessed in read-only fashion. However, in practice, this operation can create memory-contention problems, especially if more than one processor is simultaneously attempting to access the same data. The above scheme is summarized next in the P1-TDLTP algorithm. Two library functions of the CRAY are used:

TSKSTART(TID,S) which assigns the subroutine S to processor TID, and
TSKWAIT(TID) which waits for the processor TID to become idle.

4.2. P1-TDLTP algorithm

Step 1: Read network structure and travel time data and initialize the parameters and variables.
Set TID = 1.

Step 2: Let Destination(TID) be the current destination.
Call TSKSTART(TID,TDLTP) that assigns Destination(TID) and a copy of the TDLTP algorithm to processor TID.

Step 3: If TID is less than the maximum number of available processors, then set TID = TID + 1 and go to Step 2; otherwise go to Step 4.

Step 4: For TID = 1 to the maximum number of available processors, do the following
Call TSKWAIT(TID).

Step 5: Terminate.

Destination(TID) is an array that holds the destination nodes. Subroutine TDLTP is identical to the sequential algorithm outlined in Section 3. Computational results that demonstrate the performance of this design are discussed below.

4.3. Macrotasking at the SE list level (P2-TDLTP algorithm)

In contrast to the P1-TDLTP algorithm, parallelization at the single destination level is more involved and does not perform as well as in the previous approach. The main idea for the P2-TDLTP algorithm is to parallelize the SE list by exploiting the label correcting property of the SE list structure. Specifically, the label correcting property suggests that the nodes are selected from the SE list in an order not determined by their label values; thus, scanning the second or third node of the SE list, instead of the first, does not substantially affect the total computation time. Therefore, a design where many nodes are selected and simultaneously scanned by an equal number of processors, would theoretically produce a promising parallel scheme. However, this design creates the following problems:

1. The SE list may be updated simultaneously from many processors, thereby destroying its pointer structure.
2. One of the processors may insert a node in the SE list when another processor is scanning it.
3. Two processors update different copies of the same node, setting different label values and predecessor nodes.

In order to resolve the first two problems, we have to protect the SE list from simultaneous access by two or more processors. Securing the SE list turns out to be expensive computationally, because it is an operation that is activated and de-activated in every scanning iteration — with significant overhead each time. This is the reason this scheme would not work well for conventional shortest-path algorithms, where the parallel component (scanning operation) is very short and the benefit of parallelism is wasted in overhead. For the TDLTP algorithm, the scanning operation is more expensive and this scheme performs adequately. This design is well suited to the structure of multidimensional networks, because the larger the dimension of the problem, the larger is the grain of the task expected to be.

The third problem mandates the protection of the label data structure Λ_i from simultaneous access by more than one processor. This is accomplished by maintaining a binary type array in the shared memory (SECURE(node)) that can be 0 or 1 at any point in the computation process. When a node is scanned or updated by a processor, its SECURE() value is one (1), otherwise it is zero (0). This array, along with the SE list, are locked using CRAY's library functions LOCKON (Lock) and LOCKOFF(Lock). The LOCKON(Lock) command turns the value of the variable Lock ON, allowing only one processor at a time to work on that part of the code. The LOCKON(Lock) stays on until the LOCKOFF(Lock) command is encountered, which allows the next processor to move in. Calling these functions is expensive computationally and only large granularity problems can typically benefit from it.

The steps of the algotithm that incorporate the above features are described below:

4.4. P2-TDLTP algorithm

Step 1: Initialize the label vectors at the following values:
$\Lambda_N = (0, 0, \ldots, 0)$;
$\Lambda_i = (\infty, \infty, \ldots, \infty)$, for all $i \in \mathcal{V}$ except N;
Create the SE list and initialize it by inserting into it the destination node N;
Set TID = 1.

Step 2: Set SECURE(i) = 0, for all $i \in \mathcal{V}$.
For all processors (numbered TID), call TSKSTART(TID, ScanNodes).

Step 3: Call ScanNodes for the master processor.

Step 4: For all processors, call TSKWAIT(TID).

Step 5: Terminate.

4.5. Subroutine ScanNodes

Step 1: If the SE List is empty, then terminate; otherwise, go to Step 2.

Step 2: Call LOCKON(SELock).
Set j equal to the first node in the SE list;

While (SECURE(j) equal to 1 and j not the last node in the SE list), do the following:
j equals to the node next to j in the SE list.

If SECURE(j) equal to 1 and j is the last node in the SE list then
call LOCKOFF(SELock) and go to Step 1;
Delete node j from the SE list;
Set SECURE(j) = 1;
Call LOCKOFF(SELock);

Step 3: For all $i \in \Gamma^{-1}(j)$, do the following
If SECURE(i) equals 0 then
call LOCKON(SELock);
set SECURE(i) = 1;
call LOCKOFF(SELock);
go to Step 4;
otherwise,
call node j insertable in the SE list.

Step 4: For every time step $t \in S$ do the following:
check if $\lambda_i[t]$ is greater than $d_{ij}[t] + \lambda_j[t + d_{ij}(t)]$;
if it is, then replace $\lambda_i(t)$ in the label vector Λ_i at position t with the new value;
if at least one of the M labels of node i has been improved, mark node i as insertable to the SE list;
Set SECURE(i) = 0.

Step 5: Set SECURE (j) = 0.
Call LOCKON(SELock);
Insert all nodes $i \in \Gamma^{-1}(j)$ that have been marked insertable;

Insert node j in the SE list, if it has been marked insertable;
Call LOCKOFF(SELock);
Go to Step 1.

Note that Step 4 of the Subroutine ScanNodes is the only part computed concurrently by all processors. The computational time spent on Step 4 depends on the number of time intervals spanning the time period S. Greater speed-up is attained with a larger number of time intervals.

One problem we encountered with the above design was that the load was not well balanced among the processors, creating computational inefficiencies. The problem was essentially resolved by introducing an additional step between Steps 1 and 2 (Step 1′) in the P2-TDLTP algorithm, which runs subroutine ScanNodes on only one processor, until the SE list contains more nodes than the available processors:

Step 1′: Call ScanNodes. If cardinality of the SE list is greater than the number of processors then go to Step 3.

The computational results in the next section were produced from an implementation that includes the modification above.

4.6. Algorithm implementation

The three fundamental issues in implementing shortest-path algorithms are:

1. representing the network,
2. structuring the scan eligible (SE) list, and
3. representing and storing the paths.

We use the backward star structure to store the network and a simple list structure to store the paths as it is typical for static shortest-path algorithms and is explained in Ziliaskopoulos (1994). The structure of the SE list for the label-correcting algorithms has been extensively studied in the literature (see Pallotino, 1984). All SE list structures proposed for label-correcting algorithms are appropriate for the above designs: a simple list with any priority rule (i.e. first-in-first-out (FIFO), last-in-first-out (LIFO)), a queue, a 2-queue and a double-ended queue (deque). The computational results in the next section are based on a deque implementation.

The deque structure was introduced by D'Esopo and extensively tested in the literature (see Pallotino, 1984). The main feature of this structure is that it allows node insertion at both ends of the SE list — according to a predetermined strategy — and removal always from the beginning of the SE list, as shown in Fig. 1

A one-dimensional array of size equal to the number of nodes — referred to as deque — holds an integer number for each node that can take the following values:

$$Deque(i) = \begin{cases} 0 & \text{if node } i \text{ has never been on the list,} \\ -1 & \text{if the node } i \text{ was previously on the list but is no longer there,} \\ j & \text{if node } i \text{ is on the list and } j \text{ is the next node of the list} \\ \infty & \text{if node } i \text{ is the last node on the list.} \end{cases}$$

Furthermore, two pointers are kept: one pointing to the first element (*FirstNode*) and the other to the last element (*LastNode*) in the deque.

This structure is used in both implementations: P1-TDLTP and P2-TDLTP. In P1-TDLTP, every processor maintains its own deque structure. In P2-TDLTP, all deque operations are performed sequentially by locking the part of the code that accesses the deque.

4.7. Computational results

The main application machine for the shared memory designs is a CRAY Y-MP/8, a shared memory MIMD machine with eight tightly coupled vector processors (Cray Research, 1991). Each processor owns a local memory and has access to the shared memory. The processors interact through the shared memory.

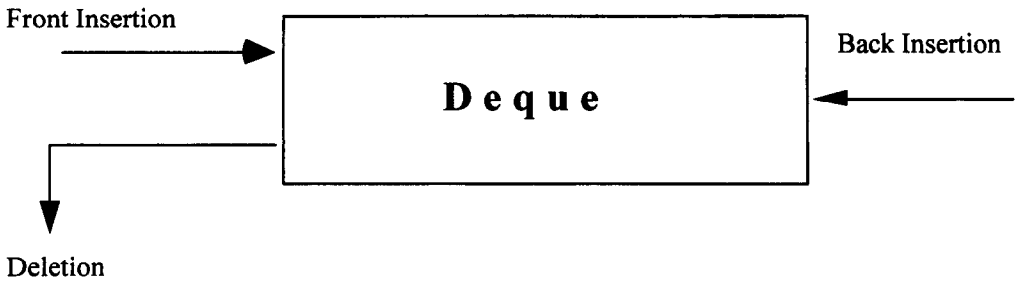

Fig. 1. Double ended queue (Deque).

The main obstacle in performing tests of the parallel implementations was the difficulty of having all the processors available in a dedicated environment. Such access was possible only a limited number of times, during which these tests were performed.

The P1-TDLTP algorithm was tested on a set of street-like networks (average node degree of approximately 3), with sizes ranging from 50 to 2500 nodes, all for 900 time intervals. The results were remarkably stable and insensitive to the size of the network. Four processors were utilized in a dedicated environment yielding speed-up of approximately 3.6, in almost every case.

The P2-TDLTP algorithm was tested on the 500 node-1250 arc network for various time intervals as shown in Table 2 and depicted in Fig. 2. Although the computational savings are not as impressive as in the P1-TDLTP case, they are substantial. Note that, as expected, the speed-up increases for higher dimension problems.

The worst case bounds for these algorithms is expected to be the same with the sequential algorithm, since in general a $|V|$ shared memory processor is not practically implementable. Thus, since the number of processors is rather small, the complexity is the same as that of the sequential algorithm.

5. THE MASSIVELY PARALLEL ALGORITHM

The algorithm first reads the network, which is originally stored in a backward star representation, by splitting the nodes evenly among the processors. Every processor keeps the backward star representation only for the nodes that belong to it, including the time-dependent travel times from every node to its successors and the labels that keep the shortest path for every time interval for the time period under inspection.

Initially all the labels for all nodes except the starting node N are set equal to infinity. The labels for node N are set equal to 0 since this node should never be updated. In the beginning, node N starts the process by sending a message to all the nodes that can directly reach it. The message contains the new labels calculated as follows:

$$\lambda_i[t + d_{iN}[t]] = d_{iN}[t] + \lambda_i[t], \qquad i \in \Gamma^{-1}(N) \tag{2}$$

where $\Gamma^{-1}(N)$ is the set of nodes that can directly reach node N.

Next, the nodes that receive the message scan their labels as follows:

$$\lambda_i[t] = \min(\lambda_i[t], \textit{received-label}) \tag{3}$$

Table 1. Computational times in milliseconds for the P1-TDLTP algorithm on various networks with 900 time intervals

	Network				
No. of CPU	50N 162A	500N 1250A	625N 1724A	1500N 5000A	2500N 8000A
1 CPU	51.33	142.20	368.3	581.23	904.5
4 CPU	13.95	40.135	102.3	164.00	254.1
Speed-up	3.677	3.543	3.601	3.544	3.561

Table 2. Computational times in milliseconds for 1 and 4 CPUs on the CRAY for the 500 node 1250 arc network and the P2-TDLTP algorithm for an increasing number of time intervals

	Time intervals				
CPUs	120	240	360	480	600
1 CPU	23.89	35.73	50.93	62.39	109.21
4 CPU	19.72	23.79	31.62	35.08	59.27
Speedup	1.211	1.501	1.610	1.770	1.840

If node i is updated, it sends a new message to its immediate successors with the updated labels calculated according to eqn (3).

This sequence is repeated until no processor can update any nodes that belong to it, at which time the whole process terminates by issuing a termination message that makes all processors quit simultaneously. If no processor works, which means that no message is pending, it is guaranteed that all possible updates have been performed.

The steps of the algorithm that run on every individual processor are as follows:

5.1. The massively parallel algorithm: steps

Step 1: Read the network in parallel. Initialize labels for all nodes in all processors, respectively.
Starting from node N send message with the updated labels for every individual node that can be reached from N.

Step 2: Check for messages in the queue. If there is a termination message go to Step 4; else, if there is a message concerning one of the nodes in the processor go to Step 3.

Step 3: Scan the node specified in the message (node i). Specifically, for every time step t check whether $\lambda_i[t]$ needs to be updated according to eqn (3). If at least one of the labels of the node i is updated then send the appropriate messages and go to Step 2.

Step 4: If the number of terminating messages already received is equal to the number of the participating processors minus 1 (the current processor), then terminate the algorithm; otherwise, increment the termination message counter by one and go to Step 2. When the algorithm terminates, the M-dimensional vectors Λ_i for every node i in the network contain the travel times of the time-dependent shortest paths from node N to every other node in the network for each time step $t \in S$.

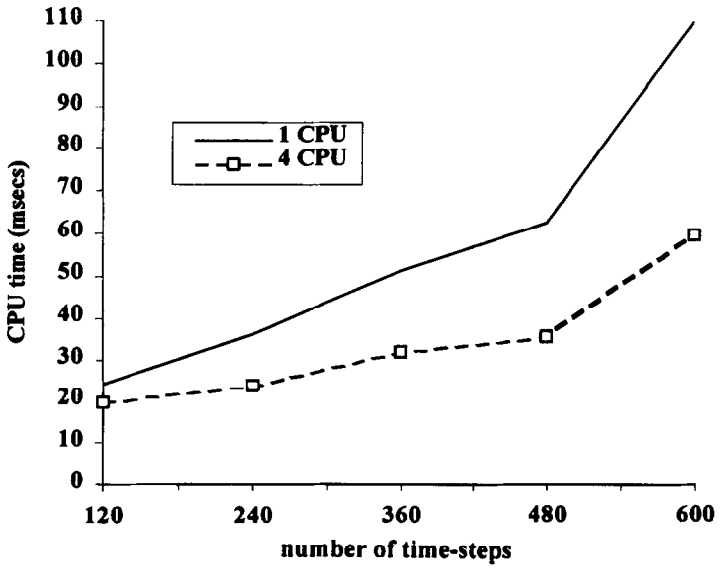

Fig. 2. Graphical representation of the results in Table 2.

Table 3. Computational results in seconds for part of Columbus street network (50 nodes–108 arcs) for various time intervals

No. of time steps	Computational time	Total time (including I/O)
10	0.017626	1.097778
100	0.491195	1.590957
250	2.59117	3.929461
500	9.218471	10.98817
1000	22.18286	25.33126
2000	55.20308	60.73199

5.2. CRAY T3D, PVM and message passing

The algorithm was implemented on a CRAY-T3D (Cray Research, 1993) available at the Ohio Supercomputer Center. This CRAY-T3D has 32 DEC Alpha processors, each of them running at a 150 MHz clock speed. Each processing element (PE) has 8 Megawords (MW) of local memory, which means that a total of 256 MW of RAM is available to programmers. The processing elements (PEs) are organized in nodes. Each node contains two PEs. The nodes are organized in a 3D-Folded Torus topology which gives minimum diameter* and maximum bisection† width. This topology also allows the programmer to reconfigure it easily as a grid or a hypercube.

The nodes are connected with a fast network (300 MB sec-six directions) which gives the ability to send many sizable messages concurrently without substantial overhead except in highly congested situations.

The algorithm was implemented using the Parallel Virtual Machine (PVM) functions (Cray Research, 1994). PVM is a parallel message passing environment that allows PEs to communicate with one another by sending messages containing all the necessary information. PVM can also be found in a network version, for a network of UNIX workstations, which makes the algorithm easily portable to lower-end architectures.

Messages exchanged among PEs contain information in both the header and the body. The header is used to identify the recipient of the message, in the case that the receiving PE has more than one network nodes to handle. After the recipient node is identified, the corresponding PE starts processing the body of the message. The body contains information retrieved in a two-dimensional array. The first dimension of the array identifies which labels need to be checked for update and the second contains the values for the corresponding labels. Finally, the parent node is

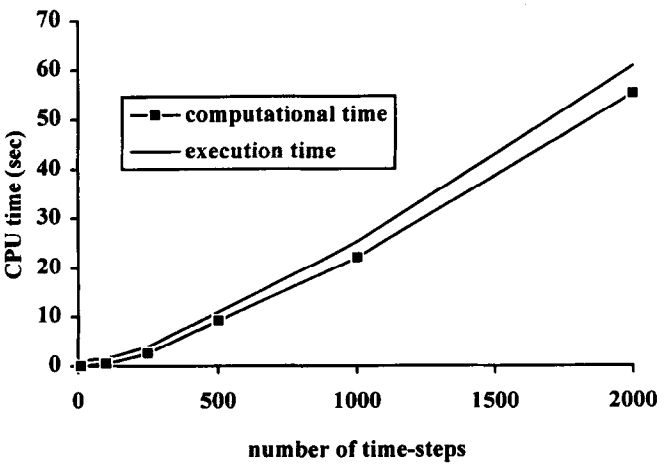

Fig. 3. Computational times in seconds for part of Columbus street network with 50 nodes and 108 arcs and various time intervals.

*Diameter: number of wires that have to be traversed in order to get from one node to the other.
†Bisection width:minimum number of wires that have to be removed in order to disconnect the network into two halves with identical number of nodes.

Table 4. Speed-up for various time intervals for part of Columbus street network (50 nodes–108 arcs)

No. of time steps	4-PEs	1-PE	Speed-up
100	0.491195	1.667	3.393764
250	2.59117	8.892	3.431654
500	9.218471	33.322	3.614699

identified in the message so that the predecessor node can be identified for reconstructing the path. There are also messages that mark the beginning and the end of the busy period of a PE which are sent to all other PEs so that they are notified when one PE finishes its work.

5.3. Algorithm implementation

The part of the algorithm that is executed locally on each PE resembles the sequential algorithm in Section 3, although the notifying process has changed drastically due to the parallel implementation. One major difference between the proposed parallel algorithm and the sequential label correcting shortest-path algorithms is the absence of the SE list. There is no need to queue the nodes in an SE list because the computations are taking place in parallel so, even if there is some redundancy, the overall performance is considerably improved. Each node is informed of all the related updates so it can start updating its own labels immediately. Without the SE list, a bottleneck is avoided since no node has to wait for some other node to finish updating the list to start its own computations.

Each node is modeled as an object which has its own data and functions for communications with the other node-objects. The data maintained for every node include the 'backward-star' representation of the successor nodes of this node only and the necessary labels for storing the shortest path for every time interval. There are also functions that check the processor's queue for messages for the specific node only and decode them in case of arrival. This gives the algorithm a great deal of flexibility since it can easily run on a massively parallel supercomputer by putting only one object per PE or, as implemented in this paper, by letting one PE handle more than one object.

Next, we state, without proof, that the computational complexity of this algorithm, for a $|V|$ processor machine, is $O(M^2 \mid V \mid^2)$. The computational results, however, suggest that the actual performance of the algorithms is far better than this worst-case bound.

The implementation of the algorithm was coded in C++ using the corresponding PVM functions for message passing and run on the CRAY T3D supercomputer. The results from the tests are presented in the next section.

5.4. Computational results

Various networks were used to test this parallel algorithm. The tests had to be restricted by memory and time limitations imposed by the CRAY administrator. The first set of networks consists of an actual network representing part of Columbus, Ohio street network with 50 nodes and 108 arcs and was tested with different number of time steps each time. The results are presented in Table 3 and graphically depicted in Fig. 3. Only four PEs of the CRAY-T3D were used in this set of experiments.

It can be readily noted from Fig. 3 that the computational time is linear with the number of time intervals. It is also interesting that the algorithm does not consume much time for I/O, which is due to the fact that I/O is performed in a parallel manner. This is important for applications that would need to frequently update time-dependent travel times due to incidents that occurred on a

Table 5. Computational times in seconds for part of Columbus street network (50 nodes–108 arcs) for various number of processing elements

No. of PEs	CPU time (secs)	Speedup
1	40.000000	1
4	10.988170	3.697
8	5.329855	7.505

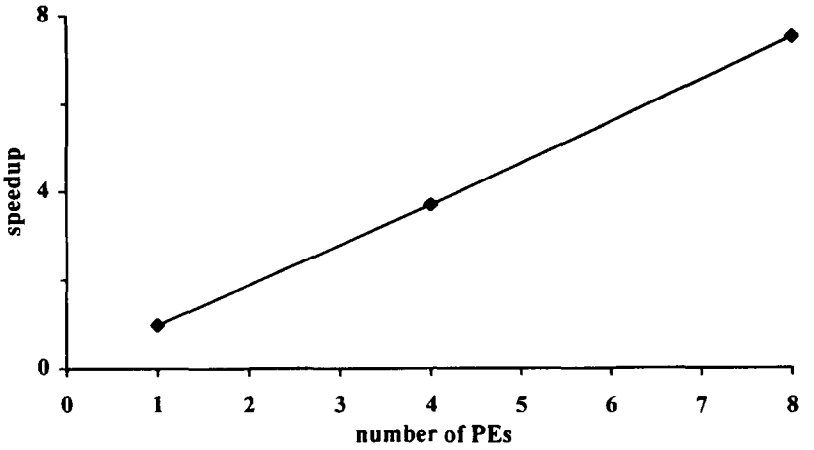

Fig. 4. Speed-up obtained for the part of Columbus Traffic Network, with 50 nodes, 108 arcs and 500 time-steps.

road in the area of interest. This would also be the case for most transportation applications, i.e. the network would be the same almost constantly but the time-dependent travel times would change due to changing real-time traffic conditions.

In Table 4, we test the performance of the algorithm on a network with increasing number of time intervals. The results demonstrate that the speed-up is improved for networks with a larger number of time intervals (i.e. finer discretization of the time period of interest) which is consistent with the results in Table 2 for the share memory design.

The results in Table 5 and Fig. 4 relate the performance of the algorithm to the number of processing elements. The speed-up obtained demonstrates that the algorithm works effectively in parallel manner taking advantage of the PEs that are in its disposal.

Although the speed-up is not quite linear, it is promising and, if extrapolated, we can expect that if sufficient number of processors were provided, real-time applicability of the algorithm is feasible with available computing architectures.

Finally, tests were performed on random networks to test the dependency of the algorithm on the number of arcs (Table 6).

From these tests, we can conclude that the number of the arcs does not drastically diminish the performance of the algorithm. Moreover, the impact is further reduced when the number of time steps increases, which agrees with the results in Tables 2 and 4.

6. SUMMARY

In this paper, three new parallel time-dependent shortest-path algorithms were introduced: two shared memory and one massively parallel. The first shared memory design assigns one destination (or group of destinations) to each processor, resulting in a remarkable speed-up; this design, although trivial, has great applicability since most real-world applications are expected to compute paths for more than one destination. The second design is based on a conflict avoidance concept and although it does not perform as well as the first design, it is promising for networks with many time intervals, because the grain of parallelization becomes coarser for a higher number of time intervals. The algorithms were implemented, tested and their computational performance was evaluated on various networks.

Table 6. Computational times in seconds for part of Columbus street network (50 nodes–108 arcs) for various number of processing elements

	Computational time (secs)		Total execution time (secs) (including I/O)	
No. of time steps	276 arcs	462 arcs	276 arcs	462 arcs
10	0.023351	0.058803	1.027034	1.878858
100	12.607796	18.606588	14.803279	20.210135

The massively parallel design is based on the concept of message passing implemented with the PVM system. One important difference between this design and other label-correcting shortest-path algorithms is the absence of the SE list. There is no need to queue the nodes in an SE list, because a node that has improved at least one of its labels informs all the adjacent nodes to proceed with the necessary updates via the message exchanging system. Substantial speed-up is attained by this algorithm which is even more improved when a larger number of time intervals is computed. The algorithms, although implemented on CRAY supercomputers, can be easily ported to lower-end machines.

Acknowledgements—This research was partly funded by the Center for Intelligent Transportation Research at Ohio State University and the Ohio Supercomputing Center. Additional support was provided by a U.S. Department of Transportation contract to the University of Texas at Austin on Traffic Modeling to Support Advanced Driver Information Systems (ADIS)–DTFH61-90-R-00074, as well as a computer time allocation from the Center for High Performance Computing at the University of Texas at Austin. The contents of the paper remain, of course, the sole responsibility of the authors.

REFERENCES

Bellman, R. (1958) On a routing problem. *Quarterly Applied Mathematics* **16**, 87–90.

Bertsekas, D. P. and Tsitsiklis, J. N. (1989) *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ, U.S.A.

Cooke, K. L. and Hasley, E. (1966) The shortest route through a network with time-dependent intermodal transit times. *Journal of Mathematical Analysis and Applications* **14**, 492–498.

Cray Research Inc. (1991) *Parallel Processing Guide*. CF77 Vol. 4. Cray Research Inc., Mendota Heights, MN, U.S.A.

Cray Research (1993) *MPP Software Guide, SG-2508* 1.0. Cray Research Inc, Mendota Heights, MN, U.S.A.

Cray Research (1994) *PVM and HeNCE Programmer's Manual*. SR-2501 5.0, Cray Research, Inc., Mendota Heights, MN, U.S.A.

Day, S. and Srimani, P. K. (1989) Fast parallel algorithm for all-pairs shortest path problem and its VLSI implementation. *IEEE Proceedings Part E, Computers and Digital Techniques* **136**, 85–89.

Deo, N., Pang, C. Y. and Lord, R. E. (1980) Two parallel algorithms for shortest path problems. *IEEE Proceedings of the 1980 International Conference on Parallel Processing*, pp. 244–253.

Habbal, M. B., Koutsopoulos, H. N. and Lerman, S. R. (1994) A decomposition algorithm for the all-pairs shortest path problem on massively parallel computer architectures. *Transportation Science* **28**, 292–308.

Kaufman, D. E. and Smith, R. L. (1993) The fastest path in time-dependent networks for intelligent vehicle/highway systems. *IVHS Journal*. **1**, 91–95.

Lakhani G. D., (1984) an improved distributed algorithm for shortest path problems. *IEEE Transactions on Computers* **C-33**, 855–857.

Mahmassani, H. S., Hu T.-Y., Peeta S. and Ziliaskopoulos, A. (1994) Development and Testing of Dynamic Traffic Assignment and Simulation Procedures for ATIS/ATMS Applications. *Technical Report DTFH61-90-C-00074-FG. CTR*. The University of Texas at Austin, Austin, TX, U.S.A.

Orda, A. and Rom, R. (1990) Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the Association of Computing Machinery* **37**, 607–625.

Paige, R. and Kruskal, C. (1985) Parallel algorithms for shortest path problems. *IEEE Transactions on Computers* **C-34**, 14–20.

Pallotino, S. (1984) Shortest path methods: complexity, interrelations and new propositions. *Networks* **14**, 257–267.

Tseng, P., Bertsekas, D. P. and Tsitsiklis, J. N. (1990) Partially asynchronous parallel algorithms for network flow and other problems. *SIAM Journal Control and Optimization* **28**, 678–710.

Ziliaskopoulos, A. K. (1994) Algorithms for Optimum Paths on Multidimensional Networks: Analysis, Design, Implementation and Computational Experiments. Ph.D. Dissertation, The University of Texas at Austin, Austin, TX, U.S.A.

Ziliaskopoulos, A. K. and Mahmassani, H. S. (1992) Design and implementation of a shortest path algorithm with time-dependent arc costs. *Proceedings of the 5th Advanced Technology Conference*, Washington, DC, 1072-1093, U.S.A.

Ziliaskopoulos, A. K. and Mahmassani, H. S. (1993) A time-dependent shortest path algorithm for real-time intelligent vehicle/highway systems. *Transportation Research Record* **1408**, 94–104.

Ziliaskopoulos, A. K. and Mahmassani, H. S. (1996) Design and Implementation of Time-Dependent Optimum Path Algorithms. *Operations Research* (under 2nd revision).