Intl. Trans. in Op. Res. 9 (2002) 279–302

INTERNATIONAL TRANSACTIONS IN OPERATIONAL RESEARCH

Parallel algorithms for dynamic shortest path problems

Ismail Chabini and Sridevi Ganugapati

Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139, USA
E-mail: chabini@mit.edu

Received 26 June 1999; received in revised form 19 October 2001; accepted 11 December 2001

Abstract

The development of intelligent transportation systems (ITS) and the resulting need for the solution of a variety of dynamic traffic network models and management problems require faster-than-real-time computation of shortest path problems in dynamic networks. Recently, a sequential algorithm was developed to compute shortest paths in discrete time dynamic networks from all nodes and all departure times to one destination node. The algorithm is known as algorithm DOT and has an optimal worst-case running-time complexity. This implies that no algorithm with a better worst-case computational complexity can be discovered. Consequently, in order to derive algorithms to solve all-to-one shortest path problems in dynamic networks, one would need to explore avenues other than the design of sequential solution algorithms only. The use of commercially-available high-performance computing platforms to develop parallel implementations of sequential algorithms is an example of such avenue. This paper reports on the design, implementation, and computational testing of parallel dynamic shortest path algorithms. We develop two shared-memory and two message-passing dynamic shortest path algorithm implementations, which are derived from algorithm DOT using the following parallelization strategies: decomposition by destination and decomposition by transportation network topology. The algorithms are coded using two types of parallel computing environments: a message-passing environment based on the parallel virtual machine (PVM) library and a multi-threading environment based on the SUN Microsystems Multi-Threads (MT) library. We also develop a time-based parallel version of algorithm DOT for the case of minimum time paths in FIFO networks, and a theoretical parallelization of algorithm DOT on an 'ideal' theoretical parallel machine. Performances of the implementations are analyzed and evaluated using large transportation networks, and two types of parallel computing platforms: a distributed network of Unix workstations and a SUN shared-memory machine containing eight processors. Satisfactory speed-ups in the running time of sequential algorithms are achieved, in particular for shared-memory machines. Numerical results indicate that shared-memory computers constitute the most appropriate type of parallel computing platforms for the computation of dynamic shortest paths for real-time ITS applications.

Keywords: dynamic shortest paths; parallel and distributed computing; computer algorithms; intelligent transportation systems; dynamic networks.

1. Introduction

Shortest path problems in networks have been the subject of extensive research for many years, resulting in a large number of scientific publications. The vast majority of these publications however have dealt with static networks that have fixed topology and fixed link costs. In recent years there has been a renewed interest in the study of shortest path problems, however, with a new twist: link costs and link travel times generally depend on the time of entry onto a link. This results in a new family of shortest path problems known as dynamic or (time-dependent) shortest path problems.

The development of intelligent transportation systems (ITS) and the resulting need for real-time dynamic management and route guidance models and algorithms are examples of applications where dynamic shortest path problems arise. One of the fundamental network problems in such applications is the computation of shortest paths from all nodes to a set of destination nodes for all possible departure times, in a given time-dependent network. This problem can be reduced to solving many dynamic shortest path sub-problems, each of which consists in finding shortest paths from all nodes to a certain destination node for all possible departure times. This last problem is the focus of this paper, and is referred to as the all-to-one dynamic shortest path problem.

The development of solution algorithms for the all-to-one dynamic shortest path problem is a research topic that has been studied by various researchers. Various solution algorithms have been proposed (Cooke and Halsey, 1966; Ziliaskopoulos and Mahmassani, 1993). Recently, Chabini (1997, 1998) proposed an efficient solution algorithm, called algorithm DOT, and showed that no sequential solution algorithm with a better worst-case computational complexity can be developed.

Numerical results show that on a Silicon Graphics Indy (SGI) workstation, algorithm DOT computes shortest paths from all nodes to one destination node for all departure times within one second, for dynamic networks containing 1000 nodes, 3000 arcs, and 100 times intervals. In the context of dynamic models of traffic flows on road networks, the number of destination nodes for networks of the above size is typically in the order of 300. Hence, the computation of dynamic shortest paths for 300 destinations would require five minutes of computation time on an SGI workstation. The solution of more evolved dynamic network models, such as the dynamic traffic assignment problem (for instance see Chabini and He, 2000), typically requires the solution of a series of all-to-many dynamic shortest path problems. Thus, computational times of sequential dynamic shortest path algorithms may not be low enough to meet the real-time operational requirement of ITS applications, at the heart of which arise the need to solve large-sized instances of evolved dynamic network models.

To achieve faster computation times, one needs to explore avenues other than design of sequential algorithms. The application of high performance computing to solve shortest path problems is an example of an avenue. The objective of this paper is to report on the design, implementation and computational testing of parallel algorithms that exploit possibilities offered by parallel and distributed computing platforms, to compute all-to-many shortest path problems in time-dependent networks.

We develop two shared-memory and two message-passing algorithm implementations. The algorithms are derived from algorithm DOT using the following parallelization strategies: decomposition by destination and decomposition by network topology. The algorithms are coded using two types of parallel computing environments: a message-passing environment based on the PVM library and a multi-threading environment based on the SUN Microsystems Multi-Threads (MT) library. We also develop a time-based parallel version of algorithm DOT for the case of minimum time paths in FIFO networks, and a theoretical parallelization of algorithm DOT on an ‘ideal’ theoretical parallel machine.

Numerical results are obtained using large-sized dynamic networks, and two parallel computing platforms: a distributed network of Unix workstations and a SUN shared-memory machine containing eight processors. Satisfactory speed-ups of sequential algorithms are achieved, in particular on shared-memory machines. Based on numerical results obtained, shared-memory platforms appear to be the most appropriate type of parallel computing platforms for the computation of dynamic shortest paths for real-time ITS applications.

This paper is organized as follows. In the next section, we overview the formulation of the all-to-one dynamic shortest paths problem, and the DOT solution algorithm. Section 3 gives an overview of concepts related to parallel computing and to the parallel implementations of this paper. Section 4 describes a message-passing algorithm and a shared-memory algorithm, which are derived from algorithm DOT using a parallelization strategy of decomposition by destinations. Section 5 describes a message-passing implementation and a shared-memory implementation of algorithm DOT using a parallelization strategy of decomposition by network topology. In Section 6 we develop a time-based parallel version of algorithm DOT for the case of minimum time paths in FIFO networks and a parallelization of algorithm DOT on an ‘ideal’ theoretical parallel machine. Section 7 summarizes results obtained from a computational study of the parallel implementations developed.

2. All-to-one shortest paths for all departure times in a time-dependent network

Chabini (1997, 1998) distinguishes various types of dynamic shortest path problems depending on the following:

(1) Discrete versus continuous representation of time,
(2) Fastest versus minimum cost (or shortest) path problems.
(3) Deterministic versus stochastic time-dependent network data.
(4) FIFO networks versus non-FIFO networks. In the latter networks one can depart later at the beginning of one or more arcs and arrive earlier at their end.
(5) Waiting policies at nodes: for such, waiting is allowed versus waiting is not allowed at nodes.
(6) Types of shortest path questions asked: one-to-all shortest paths for a given departure time or for all departure times, and all-to-one shortest paths for all departure times.

In this paper we are interested in the all-to-one and all-to-many minimum-time path problem for all departure times. The derivations in this paper can be extended to minimum-cost paths as well.

In the rest of this section, we overview a formulation and a solution algorithm for the all-to-one dynamic shortest paths problem in discrete-time dynamic networks. The main content in the rest of the section is borrowed from Chabini (1997, 1998), and is included here in order to have a self-contained paper.

2.1. Notation

Let $G = (N, A, D, C)$ be a directed network, where $N = \{1, \ldots, n\}$ is the set of nodes, and $A = \{1, \ldots, m\}$ is the set of links (or arcs). n and m are, respectively, the number of nodes and the number of arcs. Network $G = (N, A, D, C)$ is said to be time-dependent if its link travel times and costs vary as a function of time. Let t be an index to an interval of time. Let $d_{ij}(t)$ and $c_{ij}(t)$ respectively denote the travel time and travel cost experienced by a commodity traveling along arc (i, j), departing node i at time t. Sets D and C are, respectively, the set of link travel time and travel cost functions $d_{ij}(t)$ and $c_{ij}(t)$. Arc travel times are assumed nonnegative. We assume that the domain of functions $d_{ij}(t)$ and $c_{ij}(t)$ is the set of integers. Furthermore we assume that $d_{ij}(t)$ have an integer range and that $d_{ij}(t) = d_{ij}(M-1)$ and $c_{ij}(t) = c_{ij}(M-1)$ for $t > M-2$. We assume that travel costs $c_{ij}(M-1)$ can be negative, but should not lead to negative cycles. We denote by $A(i)$ the set of nodes having an incoming arc from node i, that is $A(i) = \{j \in N | (i, j) \in A\})$. $B(j) = \{i: (i, j) \in A\}$ denotes the set of nodes before node j. We use standard definitions (which can be found in standard books on algorithms) of O, θ, and Ω notations in the computational analysis of this paper.

Arc travel time functions can be such that commodities exit an arc in the same order in which they enter it. This is known as the First-In-First-Out (FIFO) property. An arc (i, j) satisfies the *FIFO condition* if function $a_{ij}(t) = t + d_{ij}(t)$ is non-decreasing as a function of t. As time is assumed discrete, the FIFO definition is equivalent to: $t + 1 + d_{ij}[t+1] \geqslant t + d_{ij}[t] \forall t \ \ in \ \ [0, M-1]$. Network $G = (N, A, D, C)$ is said to be FIFO if all its arcs are FIFO.

2.2. Problem formulation

Assume that waiting is not allowed anywhere in the network. Let $\pi_i(t)$ denote the minimum travel time to a destination node q departing node i at time index t. Functions $\pi_i(t)$ are the solution to the following system of functional equations:

$$\pi_i(t) = \begin{cases} 0 & if\ i = q \\ \min\limits_{j \in A(i)} \{c_{ij}(t) + \pi_j(t + d_{ij}(t))\} & if\ i \neq q \end{cases} \quad \forall i \in N, \forall t \in [0, M-1] \tag{1}$$

Optimal paths can be traced by saving for each node i and time t an outgoing link, the end node of which is an argument to the minimization operation in equation (1).

Equation (1) is called the dynamic shortest path optimality conditions. They are useful in designing solution algorithms, and were first proposed by Cooke and Halsey (1966). Cooke and Halsey developed a solution algorithm that extends Bellman-Ford's static shortest path algorithm, while Ziliaskopoulos and Mahmassani (1993) developed an algorithm that extends static label correcting algorithms. In both approaches only the minimum-time path problem was considered, and the label of a node, say i, is a vector of travel times from node i to destination node q for all possible departure times. The worst-case running time complexity of both algorithms is $O((n+M)(nM+mM))$.

In Chabini (1997, 1998), the acyclic nature in the time dimension of the discrete-time dynamic network is exploited to design an algorithm to solve functional equation (1). The algorithm computes optimal labels in Decreasing Order of Time, and is called algorithm DOT. The design of algorithm DOT is based on the following proposition:

Proposition 1. *Optimal labels $\pi_i(t)$ can be set in a decreasing order of departure time index t.*

Proof. Since all arc travel times are positive integers, labels corresponding to time steps t never update labels corresponding to time steps greater than t (see Equation 1). This result implicitly reflects the acyclic property, along the time dimension, of the equivalent time-space expansion of a discrete-time dynamic network. ■

2.3. *A decreasing order of time (DOT) algorithm*

For a departure time t greater than or equal to $M-1$, the computation of $\pi_i(t)$ is equivalent to a static shortest path problem. The main loop of Algorithm DOT is carried out in decreasing order of time, and assumes that a static shortest path procedure, StaticShortestPaths(l_{ij}, q), is available. The procedure computes a shortest path tree to a destination node q in network (N, A), using l_{ij} as link costs. The running time complexity of this procedure is denoted *SSP*. The statements of the algorithm DOT follow.

```
Step 0  (Initialization).
  $\pi_i(t) = \infty, \forall(i \neq q), \pi_q(t) = 0, \forall(t < M-1)$
  $\pi_i(M-1) = \text{StaticShortestPaths}(c_{ij}(M-1), q), \forall i \in N.$

Step 1  (Main Loop).
  For $t = M-2$ down to 0 do:
    For $(i, j) \in A$ do:
      $\pi_i(t) = (\pi_i(t), c_{ij}(t) + \pi_j(t + \mathrm{d}_{ij}(t))$
```

2.4. *Complexity of algorithm DOT*

Proposition 2. *Algorithm DOT solves for the all-to-one shortest path problem, with a running time in $\theta(SSP + nM + mM)$, where $\theta(SSP)$ is the worst-case computational complexity of the 'best' algorithm to compute all-to-one static shortest paths.*

Proof. The correctness of the algorithm follows directly from Proposition 1. The running time complexity can be determined in a straightforward manner by counting the number of operations in the algorithm. The initialization step needs $\theta(nM)$ operations, the main loop requires $\theta(mM)$ operations and the worst-case time complexity of the static shortest path computation is in $\theta(SSP)$. Hence, the total running time complexity is $\theta(SSP + nM + mM)$. ■

Proposition 3. *The complexity of the all-to-one shortest paths problem for all departure times is $\Omega(nM + mM + SSP)$. Hence, algorithm DOT is optimal, in the sense that no algorithm with a better worst-case running time complexity can be found.*

Proof. The problem has a worst-case complexity of $\Omega(nM + mM + SSP)$ for the following reasons. In the worst case every solution algorithm must access all arc data (mM), initialize nM labels as shortest paths for all departure times are sought (nM), and compute all-to-one static shortest paths for departure time intervals greater than or equal to $M-1$ (SSP). In Proposition 2 we proved that the computational complexity of algorithm DOT is $\theta(nM + mM + SSP)$. Hence, algorithm DOT has an optimal running time complexity. ■

3. Parallel computing concepts and dynamic shortest implementations: A brief overview

In the discussion of the previous section, we implicitly assumed that algorithm DOT would be executed on a machine that can only do a single calculation at a given time. One way to speed up the computation time of algorithm DOT is to exploit parallel and distributed computing platforms, on which it is possible to perform multiple instructions simultaneously. In this section we overview some concepts related to parallel computing in general, and summarize some aspects common to the parallel implementations developed in this paper.

3.1. Shared-memory and distributed-memory multi-processors

The concept of parallel computing consists of simultaneously using multiple processors to carry out the computation of subtasks for a given computational task. Parallel computers are now widely available. Unlike sequential computers, they differ in a variety of ways that impact the design and implementation of algorithms. Compared to sequential processing, this adds new dimensions to the complexity of algorithm design and implementation. For a given computational problem, one has to find the best combination of type of parallel computing platform, parallelization strategy, and solution algorithm.

To define the meaning of computer type, a systematic classification of parallel computers is needed. Various classifications exist in the literature. Depending on the purpose behind their use, some are more suitable than others. Below we briefly describe a classification that provides a framework for the parallel algorithms of this paper. This classification is based on the type of medium through which processors of a parallel system communicate.

In existing parallel computers, processors communicate in essentially two different ways, though hybrid solutions also exist: (1) through a shared-memory, or (2) by passing messages through links connecting processors that do not have a common memory. This leads to two (extreme) classes of computer systems: shared-memory computers, and distributed-memory or message-passing machines. A brief description of each class of computers follows.

Shared memory computers
In this parallel computer design, there exists a global memory that can be accessed by all processors. Processors can communicate with one another by writing into locations in the global memory. A processor requiring this information reads it from that global-memory location. Although this solves the inter-processor communication problem, it also introduces other problems. For instance, one would need methods to resolve possible conflicts due to simultaneous modifications by different processors of the data stored in the same memory location. This problem is usually solved using *mutual exclusion locks* (see Lewis and Berg, 1996, for more details on this topic). There is however generally no need to explicitly resolve conflicts when processors are accessing memory locations for reading purposes only, as computer systems typically contain built-in switching circuits that automatically resolve such conflicts. Note that mechanisms which resolve data-access conflicts introduce computational delays.

Distributed memory (message-passing) systems
In these systems, processors have their own local memory, and typically communicate through an interconnection network, which possesses a topology describing how processors are interconnected. Common topologies are rings, trees, meshes, and hypercubes. A major factor affecting the speed of

parallel algorithms developed for these systems is the amount of time taken to communicate between the processors. An appropriate topology is chosen depending on the communication requirements of the parallel algorithm. In the next sections we will see that for certain decomposition strategies, implementations of dynamic shortest path algorithms on these systems can lead to high communication requirements. In such cases, shared-memory systems are usually a better alternative.

3.2. Multi-processor platforms: Examples and programming

We now give two examples of multi-processor platforms, each of which was used to test the computer implementations of the parallel algorithms developed in this paper. A SUN UltraSparc HPC 5000 Symmetric Multiprocessor workstation is an example of a shared-memory machine. SUN UltraSparc HPC 5000 workstations used to evaluate the shared-memory implementations of this paper have 512MB of RAM, 8 processors and 1GB of swap memory. At MIT we have nine such machines organized in a cluster, known as the Xolas cluster. An example of distributed-memory machines is a distributed network of Silicon Graphics Indy (SGI) workstations.

The development of codes on shared-memory machines is usually done using multi-threading techniques (see Lewis and Berg, 1996, for more details on this topic). Many processors can simultaneously execute threads, which have access to a single copy of the algorithm code and data. Each processor executes the same code for its own subset of data. The subsets of data may not be disjointed. Global memory locations used by processors for communication purposes are examples of data that may be accessed by all processors. Access to joint data for writing purposes, must be done in a way to avoid simultaneous access conflicts and preserve the order of data updating as specified in the algorithm. Shared-memory implementations of this paper were developed using the Solaris Multi-threading (MT) library (Lewis and Berg, 1996).

Parallel codes for distributed memory machines are a collection of sequential processes that communicate between each other. The physical communication medium is the interconnection network that links processors to each other. Processors communicate by passing messages through a communication software library. Parallel virtual machine (PVM) (Geist et al., 1995) is an example of such a library. PVM was used to develop the distributed memory implementations of this paper. Note that PVM implementations also run on shared-memory machines.

Tasks assigned to processors can be implemented as threads or as processes. The tasks can be organized under various paradigms such as the master-slave(s) paradigm, in which a master process (master thread) decomposes the problem into sub-problems and spawns out slave processes (slave threads) to solve each of the sub-problems. Slave processes (slave threads) report back to the master process (master thread) at the end of the computation.

3.3. Decomposition strategies for dynamic shortest paths computation

Parallel implementations of a given algorithm are generally obtained by partitioning the computational tasks into subtasks each of which is then assigned to a processor. This is known in the parallel computing literature as decomposition. For a given sequential algorithm, there usually exist various dimensions along which decomposition can be achieved. Note that for larger and more uniform subtasks, the performance of a parallel implementation improves.

Below we outline possible decomposition strategies for the following computational tasks: (1) the computation of dynamic shortest paths from all nodes to many destinations by repeatedly applying algorithm DOT, and (2) the computation of all-to-one dynamic shortest paths using algorithm DOT. Note that a decomposition scheme for the last computational task can also serve as an implicit decomposition scheme for the first computational task.

The above two computational tasks offer three dimensions to develop parallel implementations: (1) the set of destinations to which dynamic shortest paths are sought, (2) the set of network arcs (or links), and (3) the set of departure/arrival times at nodes. We now give a brief description of each decomposition strategy.

Destination-based decomposition

This design is trivial and has been adopted in earlier works such as Chabini et al. (1996), and Zialiaskopoulos et al. (1997). The set of destinations is divided into disjoint subsets. Subsets are then allotted to different processors, which compute all-to-one dynamic shortest paths for each of the destinations in their assigned subset.

Network-topology-based decomposition

The network is split into sub-networks. Each sub-network is allotted to a processor. Each processor determines the optimal labels of the nodes that belong to its sub-network. Parallel implementations of static shortest path algorithms using this decomposition technique have been developed in the literature (Hribar and Taylor, 1996; Habbal et al., 1994). To the best of our knowledge, no such implementations have been developed for dynamic shortest path algorithms.

Time-based decomposition

The decomposition of algorithm DOT by time is less explicit than the other two decomposition dimensions. A time-based parallelization of this algorithm is given in Section 6. Time-based parallel implementations can alternatively be viewed as decompositions by network topology of the time-space network representation of the time-dependent network.

Cooke and Halsey (1966)'s algorithm is decomposable by time and by network topology. Each step of the algorithm in Ziliaskopoulos and Mahmassani (1993) performs $O(M)$ operations, which are amenable to a decomposition by time. This latter parallelization was explored in Ziliaskopoulos et al. (1997), in the context of shared memory machines. The time loop in algorithm DOT performs $O(m)$ operations that are amenable to a decomposition by number of arcs. A sequential computational analysis of all-to-one dynamic sequential shortest path algorithms in Chabini and Dean (1999) shows that algorithm DOT is significantly faster than currently known alternative algorithms and is less sensitive to the choice of a destination node. Consequently, in dynamic networks with a number of time intervals less than or equal to the number of arcs (which is usually the case in practice), it follows that parallel implementations of algorithm DOT would lead to faster running times than parallel implementations of other alternative algorithms in the literature. Section 6 shows that a parallelization of algorithm DOT is also possible along the time dimension.

Based on the first two decomposition strategies outlined above, we have developed four parallel implementations based on algorithm DOT: two shared-memory and two distributed-memory implementations. In each of the two shared-memory implementations, only one copy of the network is stored in the global memory. In distributed-memory implementations, where each processor has its own local memory, the network storage scheme depends on the decomposition strategy adopted. For the destination-based decomposition, each processor needs a copy of the entire dynamic network. Hence the network is replicated: a complete network copy is stored in each processor's local memory. For the network-topology-based decomposition, only the sub-network assigned to a processor is stored in its local memory.

3.4. *Some notation used in the description of parallel implementations*

Following is additional notation used in the description of the parallel implementations. *MP* denotes a master process or thread. The number of processors is denoted by p. The set of indices to slave processes (or slave threads) is $P = \{1, \ldots, p\}$. k denotes the number of destinations to which dynamic shortest paths need to be determined. Without loss of generality, we assume that the set of destinations to which dynamic shortest paths needs to be computed is $K = \{1, \ldots, k\}$. This set is decomposed in p disjoint subsets K_i, such that that $\cup_{i=1}^{p} K_i = K$. Subset K_i is allotted to slave process (thread) $i \in P$. If r is the remainder of the integer division of k by p, K is decomposed into disjoint subsets K_i such that processors indexed 1 to r, contain $(k \div p + 1)$ destinations each, while the other slave processes (or slave threads) are assigned $k \div p$ destinations each.

4. Parallel implementations based on decomposition by destinations

4.1. *Distributed-memory parallel implementation*

4.1.1. *Description of the implementation and the algorithms*

As each processor stores its own copy of the dynamic network, in this implementation the master process has to communicate all network information to all slave processes. Since dynamic shortest path results are computed in different processor memories, if it is needed, the master process collects them from all the slave processes at the end of the computations. In Section 7, we will see that the collection of results leads to communication times that may significantly impact the performance of a distributed-memory implementation.

The collection of results by the master process is done in the following way. A slave process sends the results as soon as the computation for a given destination node, say node j, is complete. Consequently, while this slave process computes for the next destination, $j + 1$, the master process is busy receiving the results for destination j. This method is better than an approach in which a slave process would first compute shortest paths to all destinations assigned to it, before sending back at once all results; the latter approach may lead to idle times at the master process, and at the slave processes, and hence increase the overall computation time.

Below are statements of the algorithms on which the master process and the slave processes are based, and which together constitute the distributed-memory implementation of the dynamic shortest path computations based on a decomposition by destinations.

Master process algorithm: destination-based decomposition

```
1. Read the dynamic network $G(N, A, D, C)$
2. Decompose the destinations set $K$ into $p$ disjoint subsets $K_i$
```

```
3. Spawn $p$ child processes
4. Broadcast network $G$ to all $p$ processes
5. For all $i \in P$ and $K_i \neq \phi$, send subset $K_i$ to process $i$
6. While ($K \neq \phi$)
   • Receive dynamic shortest path labels for a destination $j \in K_i$ from a
     ready processor $i$
   • $K = K - \{j\}$
7. Broadcast the message "Quit" to all processes $i \in P$
8. Stop
```

Slave process algorithm: destination-based decomposition

In the statements of the slave process algorithm, $i \in P$ denotes the index of the slave process on which the algorithm runs.

```
1. Receive the network $G$ from the master process $MP$
2. Receive subset $K_i$ from the master process $MP$
3. For all $j \in K_i$
   – Run Algorithm DOT
   – Send the dynamic shortest path labels for destination $j$ to the master
     process $MP$
4. Wait for the message "Quit" from the master process $MP$
5. Stop
```

4.1.2. Run-time analysis

In the analysis below we assume that the time required transmitting one byte in a data-message between two processors, does not depend on the following: the size of the message, the pair of processors, and on the number of processors. Denote by c the average time required to transmit one byte of data between any pair of processors.

Proposition 4. *The average worst-case running time complexity of the distributed-memory implementation based on decomposition by destination is given by:*

$$O\left(mMc\log p + Max\left(\frac{k}{p}(nM + mM + SSP) + \frac{k}{p}nMc,\ knMC\right)\right)$$

Proof. The first term ($O(mMc \log p)$) in the worst-case run-time complexity denotes the amount of time required to communicate the network to all p slave processes by the master process. The amount of information to be communicated is in $O(mM)$, which corresponds to link travel-times for all possible M departure times. As we broadcast the information instead of sending it in sequence to each processor, the communication time required sending link travel times to all processes is in $O(mMc \log p)$.

The second term denotes the time that a slave process takes to compute dynamic minimum time labels, and to communicate them to the master process. The computation of dynamic fastest paths and the communication of results are interleaved. The total time taken to compute the dynamic shortest paths for k destinations by p processors using algorithm DOT is $(k/p)(nM + mM + SSP)$. The time taken by a slave process to send out nM fast path labels corresponding to k/p destinations is $(k/p)(nMC)$. The total time spent by a slave process to compute the results and to send them out to the master process is $(k/p)(nM + mM + SSP + nMC)$. The master process takes $knMc$ units of time to collect nM labels for k destinations. The time taken to compute and communicate the results to the master process is the maximum of the worst-case run-time taken by the master and the slave processes to carry out these two separate tasks. ■

4.2. Shared-memory destination-based parallel implementation

4.2.1. Description of implementation and algorithms

In the shared-memory implementation, each slave sub-task is implemented as a different thread. Only one copy of the network is maintained in the global memory. As all the threads have access to all network data and results, the master thread does not need to collect the results from the slave processes. Below are statements of the algorithms on which are based the master and slave threads.

Master thread algorithm: destination-based decomposition

```
1. Read the network $G(N, A, D, C)$
2. Decompose the destinations set $K$ into $p$ disjoint subsets $K_i$
3. Create $p$ child slave threads
4. Wait for the slave threads to join
5. Stop
```

Slave thread algorithm: destination-based decomposition

In the statements of a slave thread algorithm, $i \in p$ denotes the ID of a slave thread

```
1. For all $j \in K_i$ do:
   – Run a thread implementation of algorithm $DOT$ for destination $j$ using
   network $G$
2. Exit
```

It can be seen from the above algorithm statements that there is no communication required in the shared-memory implementation. The ‘equivalent’ of the communication delay required in the distributed-memory implementation, is the time needed to resolve possible contentions of threads to access a same memory location, while reading link travel times for given departure times. This can increase the memory-access time, and hence the overall running time of the parallel implementation. This computation-time overhead is assessed in Section 7.

5. Parallel implementations of algorithm DOT by decomposition of network topology

In the discussion of this section, we focus on the parallelization of the truly dynamic part of algorithm DOT, which corresponds to $t < M - 1$. The computations needed at time $t = M - 1$, which consist of computing a static shortest path tree, are comparatively less expensive than the computational needs of the dynamic part. A variety of parallel implementations of static shortest path algorithms exist in the literature. They can be adopted for the computational needs of algorithm DOT at time index $t = M - 1$. We hence omit including details on the parallel implementation of the static part of algorithm DOT.

Shortest path algorithms generally visit network links in a particular order, such as a forward (or a backward star) order. In the main step of algorithm DOT, links can be processed, however, in an arbitrary order at each time interval. This renders the decomposition based on network topology particularly suitable to apply to algorithm DOT.

5.1. Additional notation

We provide additional notation used in the description of the master process (thread) and the slave process (thread) algorithms of this section. The set of nodes N is split into p disjoint subset of nodes N_i, such that $\cup_{i=1}^{p} N_i = N$. Each subset of nodes will be assigned to a different processor. Let $P(i)$ denote the index of the subset of nodes to which node i belongs. If node $i \in N_j$, we have $P(i) = j$. Let S_{ij} be the set of links from a node in sub-network i to node in sub-network j: $S_{ij} = \{(x, y) \in A | x \in N_i, y \in N_j\}$. Let SD_{ij} denote the set of travel time and travel cost functions of the arcs belonging to S_{ij}: $SD_{ij} = \{(d_{xy}(t), c_{xy}(t)) | (x, y) \in S_{ij}, 0 \leq t \leq M - 1\}$. Set S_{ij} contains the arcs linking nodes in subset N_i . Let S_{ij}^{from} denote the set of start nodes of all arcs belonging to set S_{ij}, i.e. $S_{ij}^{from} = \{x | (x, y) \in S_{ij}\}$. Similarly, let S_{ij}^{to} denote the set of end-nodes of all arcs belonging to set S_{ij}, i.e. $S_{ij}^{to} = \{y | (x, y) \in S_{ij}\}$. Note that $S_{ij}^{from} \subset N_i$, and that $S_{ij}^{to} \subset N_j$.

5.2. Distributed-memory implementation

In this implementation each processor $i = 1, \ldots, p$, stores the following data: a copy of its own sub-network which is composed of nodes in subset N_i, and of the links in sets $S_{ij}(j = 1, \ldots, p)$, as well as the sets of nodes S_{ij}^{from} (and S_{ij}^{to}) $(j = 1, \ldots, p)$, to which processor i sends (respectively from which processor i receives) values of optimal labels that are needed in subsequent computations. The master process communicates information about these sets to all p slave processes. In Section 7 we will see that the communication of this data can require significant communication times that substantially impact the overall running time of this distributed-memory implementation. The results of dynamic shortest path computations reside in the processors' local memories. If needed, the master process collects them from the slave processes at the end of the computation. This of course adds an additional communication time overhead, which however should be lower than the communication time overhead discussed above.

In the remainder of this subsection, we provide the statements of the algorithms on which the master process and the slave processes are based. Together these algorithms constitute the distributed-memory implementation of algorithm DOT, based on a decomposition by network topology.

Master process algorithm: decomposition by network topology

```
1. Read the network $G(N, A, D, C)$
2. Compute $\pi_i(M-1)$ = StaticShortestPaths($c_{ij}(M-1)$, q) $\forall i \in N$
```

```
3. Divide the set of nodes $N$ into $p$ disjoint subsets $N_i$
4. $\forall(i, j) \in$ A do: $(l, m) = (P(i), P(j))$; $S_{lm} = S_{lm} \cup \{(i, j)\}$
5. Spawn $p$ slave processes
6. Send to processes $i = 1, \ldots p$:
   • Set $N_i$
   • Sets $S_{ij}$, $S_{ji}$ and $SD_{ij}$, $\forall j \in P$
   • Destination node $q$, and $\pi_s(M-1), \forall s \in N_i$
7. $\forall i \in P$ do:
         Receive from process i: $\pi_j(t)\ \forall(j, t) \in N_j \times [0, M-2]$
8. Broadcast the message "Quit" to all slave processes
9. Stop
```

Slave process algorithm: decomposition by network topology

In the algorithm statements, index i denotes the ID of the slave process, ($i \in P = \{1, \ldots, p\}$).

Initialization

```
1. Receive from the master process ($MP$):
   • Set $N_i$
   • Sets $S_{ij}$, $S_{ji}$ and $SD_{ij}$, $\forall j \in P$
   • Destination $q$
   • $\pi_s(M-1)\ \forall s \in N_i$
2. Set $\pi_s(t) = \infty, \forall s \in N_i - \{q\}, \pi_q(t) = 0, \forall 0 \leqslant t < M-1$
```

Main Steps

```
3. For $t = M-2$ downto 0
   • $\forall j = 1, \ldots,$ p:   $\forall(a, b) \in S_{ij}$ do:
       $\pi_a(t) = \min(\pi_a(t), c_{ab}(t) + \pi_b(Min(t + d_{ab}(t), M-1)))$
   • $\forall j \in P, j \neq i$ do:
       Send to process $j$: $\pi_a(t), \forall a \in S_{ij}^{to}$
   • $\forall j \in P, j \neq i$
       Receive from process $j$: $\pi_a(t), \forall a \in S_{ji}^{to}$
4. Send $\pi_a(t), \forall a \in N_i, \forall t, 0 \leqslant t \leqslant M-2$
5. Wait for message "Quit" from $MP$
6. Stop
```

5.3. Shared-memory implementation

In algorithm DOT, the labels of nodes for time index t should be set before computing labels for time index $(t-1)$. All threads need to be synchronized before the computation for a next time index can proceed. (This condition can, however, be relaxed such that the synchronization is done only after a number of time steps that is equal to the minimum among all link travel times.) In the distributed-memory implementation, this synchronization is implicitly ensured, as a slave processor may not proceed to time $t-1$ until it receives necessary results for time t from other processors. The synchronization of threads is implemented using a synchronization barrier function (see Lewis and Berg 1996). Let us denote by SYNCHRONIZATION_BARRIER(x) the function that synchronizes an 'x' number of threads.

Master thread algorithm: Decomposition by network topology

The statements of the master thread algorithm are as follows.

```
1. Read the network $G = (N, A, D, C)$
2. Compute $\pi_i(M-1)$ = StaticShortestPaths($c_{ij}(M-1)$, q) $\forall$ i $\in$ N
3. Divide the set of nodes $N$ into $p$ disjoint subsets $N_i$
4. For all $(i, j) \in A$ do: $(l, m) = (P(i), P(j))$; $S_{lm} = S_{lm} \cup \{(i, j)\}$
5. Create $p$ slave threads
6. Wait for all the threads to join back, and then Stop
```

Slave thread algorithm: Decomposition by network topology

A slave thread $i \in \{1, \ldots, p\}$, performs the following sequence of steps:

```
1. Set $\pi_s(t) = \infty, \forall s \in N_i - \{q\}, \pi_q(t) = 0, \forall 0 \leq t < M-1$
2. For $t = M-2$ downto 0 do:
   1.1 For all $(a, b) \in S_{ii}$: $\pi_a(t) = Min(\pi_a(t), c_{ab}(t) + \pi_b(Min(t + d_{ab}(t), M-1))$
   1.2 For all $j \in P, j \neq i$
          For all $(a, b) \in S_{ij}$ do:
          $\pi_a(t) = Min(\pi_a(t), c_{ab}(t) + \pi_b(Min(t + d_{ab}(t), M-1))$
   1.3 SYNCHRONIZATION_BARRIER(p)
3. Exit
```

There are two potential sources of parallelization time overheads in a shared-memory implementation: (1) the waiting time at the synchronization barrier for every time interval t; (2) the time lost due to the contention of threads to access given link travel time data or a given minimum travel time label computed at earlier steps. If the computational loads of the threads are balanced, the lost time at the synchronization barrier is minimal.

5.4. Decomposition of network topology

In the preceding description of the two implementations of algorithm DOT based on the decomposition by network topology, we did not specify the criteria and methods used to partition the network into sub-networks. An adequate partitioning should split the network into p balanced sub-networks. That is, for each sub-network, the sum of the number of links within the sub-network and the number of its boundary outgoing links should be 'equal'. The problem of partitioning a network satisfying this condition is known to be an NP-Hard problem (see for instance Karypis and Kumar, 1998). In the case of distributed-memory implementations, one should additionally balance the sum of the number of incoming and outgoing links of node-subsets N_i.

There exist a variety of network-partitioning algorithms and software libraries in the public domain. In the implementations of this paper, we have used a graph partitioning software library called METIS (Karypis and Kumar, 1998). It is available in the public domain and was developed at the University of Minnesota. Following are statistics obtained using METIS to decompose a randomly generated network composed by 3000 nodes and 9000 arcs into two sub-network parts. The first part contains 3504 links and the second part contains 3518 links. There were 1020 links going from sub-network 1 to sub-network 2, and 958 going from sub-network 2 to sub-network 1. At each time interval in algorithm DOT, a slave process needs to send (receive) information about the boundary links to (from) the other slave processes. In the network example of this paragraph, at every time interval, slave process 1 will need to send 958 node labels to process 2, and receive 1020 node labels from process 2, and vice versa for process 2. Moreover, the master process will send to the two slave processes their respective link travel time data, and the slave processes will send the computed labels back to the master process. The network example of this paragraph suggests that in a distributed-memory implementation, a substantial amount of communication time may be required among the slave processes and between the master process and the slave processes. Consequently the distributed-memory implementation can potentially become slower than even the sequential implementation as a result of these coordination requirements. The multi-threads shared-memory implementation may also ‘suffer’ from these coordination requirements, however at a lesser degree with comparison to the distributed-memory implementation.

6. Other parallel implementations of algorithm DOT

6.1. A time-based parallelization

In this section we present a time-based parallelization of algorithm DOT. It is valid for the minimum-time path problem in FIFO networks. We first study the problem of computing maximum departure times at nodes, for which it is possible to arrive at destination node q at or before a given time t_0.

Let us consider the latter problem for one arc, say (i, j). Consider function $b_{ij}(s)$ defined as follows: $b_{ij}(s) = \mathrm{Max}\{t,\ \text{such that } a_{ij}(t) <= s\}$. As $a_{ij}(t)$ is non-decreasing (because the network is assumed FIFO), function $b_{ij}(s)$ is well defined on interval $[d_{ij}(0), +\infty)$, and is also non-decreasing.

Consider a path, say $i_1 - i_2 - \ \dots - i_k$. The latest departure time at the beginning of the path corresponding to an arrival by times s at its end, is given by the following composite function: $b_{i_1 i_2}(\dots(b_{i_{k-1} i_k}(s)))$.

Denote by f_j the latest departure time at node j, such that there exists a path that allows one to arrive at destination node q by time t_0. f_i are solutions to the following equations:

$$f_i = \begin{cases} t_0, & \text{if } i = q \\ \max_{j \in A(i)} b_{ij}(f_j), & \text{otherwise} \end{cases} \tag{2}$$

For $t_0 <= M - 1$, equations (2) can be solved by a dynamic adaptation of a static shortest path algorithm such as Dijkstra’s shortest path algorithm. Details of such adaptations are omitted in this paper, as they are similar to well-known dynamic adaptations of static shortest path algorithms to solve one-to-all fastest path problems in FIFO networks for a given departure time. The latter two problems are in fact symmetric problems of one another in FIFO networks.

We now describe the time-based parallelization of algorithm DOT to solve the all-to-one minimum time problem in FIFO networks. We discuss the parallelization for the case of two processors only. The generalization of the result to multiple processors is straightforward. Consider an arrival time t_0 (for instance $t_0 = (M-1)/2$). Solving equation (2), one obtains for each node $i \in N$ the latest departure time f_i for which an arrival at the destination node q is possible by time t_0.

Now consider the following sets of (node, time) pairs: $N_1 = \{(i, t) | i \in N \text{ and } t \leqslant f_i\}$ and $N_2 = \{(i, t) | i \in N \text{ and } t > f_i\}$. We assign to one processor the computation of minimum time labels for (node, time) pairs in N_1. To the second processor we assign similar computations for (node, time) pairs in set N_2. The computations in both processors involve only travel times of links that are internal to their corresponding set of (node, time) pairs. The remaining arcs, which form a cut from N_1 to N_2 in the time space network, need not be considered, as they will never be part of a minimum time path for (node, time) pairs in set N_2 (and also in set N_1). Therefore, the partition of the all-to-one minimum time path computations for the whole time-space network reduces to two disjoint sub-networks. Thus, the two processors need not communicate and the total number of link travel time data sent by the master process is at most equal to the travel time data of all links. Each processor independently applies algorithm DOT for its assigned sub-network. Note that the parallelization described in this subsection applies to any all-to-one minimum time path algorithm in FIFO networks.

6.2. *Implementation of algorithm DOT on an ideal parallel machine*

We define an ideal parallel computer as a shared-memory parallel computer containing as many processors as required by the parallel algorithm, and with a constant memory access time regardless of the number of processors in use.

In describing parallel algorithms designed for this ideal parallel computer, we use parallel statements of the general form:

For *x in S* in parallel do: *statement*(*x*)

This statement means that we assign a processor to each element *x* of set *S,* and then carry out in parallel the instructions in *statement(x)* for every such element, using *x* as the data.

The parallel implementation of algorithm DOT on the ideal parallel computer is referred to as *DOT-IP.* In *DOT-IP,* we use a similar technique to the network decomposition technique described in Section 5. At each time step, we use *m* processors to run the main loop of algorithm DOT. We use *nM* processors to initialize the labels for all nodes at all time intervals. We assume a parallel static shortest path algorithm called *ParallelStaticShortestPaths*(N, A, l_{ij}, q), which returns the all-to-one static shortest path distances in the minimum run time possible. l_{ij} denotes link costs and q denotes the destination node. The statements of algorithm *DOT-IP* are:

Step 0 (Initialization).

```
$\forall(t < M - 1)$ in parallel do: $\pi_q(t) = 0$
$\forall(i \neq q, t < M - 1)$ in parallel do: $\pi_i(t) = \infty$
$\pi_i(M - 1) = \texttt{ParallelStaticShortestPaths}(N, A, c_{ij}(M - 1), q), \forall i \in N$
```

Step 1 (Main Loop).

```
For t = M − 2 downto 0 do:
  For all (i, j) ∈ A in parallel do: $\Phi_{ij} = (\pi_i(t), c_{ij}(t) + \pi_j(t + d_{ij}(t))$
  For all i ∈ N in parallel do: $\pi_i(t) = \mathrm{Min}\{\Phi_{ij}, j \in A(I)\}$
```

The worst-case run-time complexity of algorithm *DOT-IP* is $O(PSSP + M\ log(R))$, where $O(PSSP)$ is the best possible worst-case run-time complexity of a parallel static shortest path algorithm on an ideal parallel computer, and R is the maximum out-degree of a node. Assuming that $O(M\ log(R))$ dominates $O(PSSP)$, algorithm *DOT-IP* is approximately $O(m^* \log R)$ times faster than algorithm DOT. Thus, the maximum speed-up of algorithm DOT on an ideal parallel machine is approximately $O(m^* \log R)$.

Note that a better implementation of algorithm DOT on an ideal parallel computer, can be developed in the case of minimum time paths in FIFO networks. One can partition the computations into M independent static shortest path problems, each of which corresponds to computing the latest departure times at all nodes for an arrival at destination node q by a time between 0 and $M - 1$. These M problems are simultaneously solved in $O(PSSP)$ on an ideal parallel machine.

7. Computational results

7.1. Introduction

The shared-memory implementations were coded using the SUN Solaris Multi-Threads (MT) library and the C++ programming language. The distributed-memory implementations were coded using the C++ programming language and the PVM interprocess communications library.

We conducted a computational study of the parallel implementations developed to assess and analyze their computational performance. Given space limitations, we report only on a subset of computational results.

The platforms used to evaluate the four parallel implementations are (1) a SUN Ultrasparc HPC 5000 workstation denoted here as the Xolas machine and, (2) a distributed network of six SGI workstations. A Xolas machine is a shared-memory computer containing eight processors. Both the PVM and the multi-threads (MT) implementations can run on this platform. Results on a Xolas machine obtained using the PVM and the MT implementations are respectively referred to as PVM-Xolas and MT-Xolas. We refer to the results obtained on the distributed network of SGI workstations using the PVM implementations as PVM-SGI. The MT implementations may not benefit from the distributed network of SGI workstations, as these are not shared-memory machines.

Most of the numerical results obtained in this section were performed using a random network with 1000 nodes, 3000 links and 100 time intervals. The running time of sequential algorithm DOT for one destination was in the order of 1 second (0.98 seconds on (one processor of) the Xolas machine, and 1.08 seconds on an SGI workstation).

7.2. Performance measures

The parallel performance measures reported are the curves of speed-up and of relative burden, as a function of the number of processors available on each parallel machine used in the tests. Let $T(p)$

denote the running time obtained using p processors. The speed-up is defined here as $T(1)/T(p)$. Next, we first give the motivation behind the relative burden measure before we give its definition.

When the number of processors on parallel machines used in laboratory experiments is limited, the evaluations of the performance measures are available for only a relatively small number of processors. In our study, the number of processors is in the order of eight. One may, however, want to be able to draw conclusions on the performance of parallel implementations for a larger number of processors.

The speed-up measure does not generally allow for performance predictions for a larger number of processors, based on results obtained using a small number of processors. This is essentially due to numerical problems inherently related to the definition of the speed-up measure. The relative burden is a parallel performance measure that was designed to avoid this numerical problem. It measures the deviation from the ideal improvement in time from the execution of the algorithm on one processor, to its execution on p processors, normalized by the running time on one processor. The expression of the burden is: $(T(1)/T(p)-1/p)$.

As their definitions suggest, the burden and the speed-up are interrelated measures. If we denote by $S(p)$ and $B(p)$ the speed-up and the relative burden, we have: $B(p) = (1/S(p) - 1/p)$ and $S(p) = p/(1 + B(p)p)$. In most parallel algorithms, the value of the burden is usually small, especially for smaller values of the number of processors. Consequently, for smaller values of the number of processors p, the term $B(p)p$ in the denominator of the expression of the speed-up as a function of the burden is typically dominated by 1. Hence, the speed-up curve of any parallel algorithm is typically linear (or almost linear) for smaller values of the number of processors.

Relative burden curves can be useful in obtaining further insight into the performance of parallel implementations for a larger number of processors, and in situations where speed-up curves alone do not distinguish among parallel implementations for smaller values of the number of processors. For instance, the relative burden can be used to obtain better estimate of (maximum) speed-ups for larger values of the number of processors. This is useful if one wants, for instance, to obtain estimates of speed-ups, prior to investing in a parallel machine with a larger number of processors.

A description and an analysis of the speed-up and the relative burden performance measures can be found in Chabini and Gendron (1995). A mathematical analysis of the speed-up as a function of the burden demonstrates that if the relative burden curve is linear, the maximum speed-up is achieved at a number of processors equal to the square root of the inverse of the slope of the linear curve. If the relative burden tends towards a constant, the maximum speed-up is reached asymptotically. For large values of the number of processors, the speed-up converges to the inverse of the relative burden. In the rest of this paper, we also use the shorter term 'burden' to refer to the term 'relative burden'.

7.3. Numerical results for destination-based parallel implementations

Figure 1 shows the speed-up curves of the PVM-Xolas , PVM-SGI and MT-Xolas implementations based on a decomposition by destinations to compute all-to-many dynamic minimum time paths for 100 different destinations using algorithm DOT. The transportation network used contains 1000 nodes, 3000 links, and 100 time intervals.

PVM implementations involve message exchanges between the master process and the slave processes, while the MT-Xolas does not. This explains why the MT-Xolas implementation showed better speed-ups than the PVM implementations. Furthermore, the PVM-Xolas implementation shows better speed-ups than the PVM-SGI implementation. This is due to faster communication speeds on a

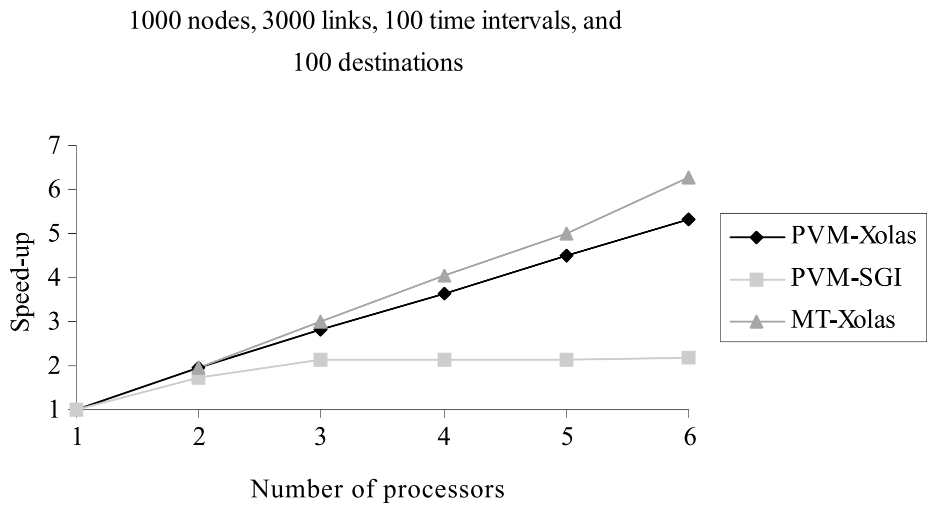

Fig. 1. Speed-up curves of the parallel implementations based on decomposition by destinations.
Note: The network used in these tests contains 1000 nodes, 3000 links, 100 time intervals, and 100 destination nodes.

Xolas workstation, as compared to the distributed network of SGI workstations where communications take place over a non-dedicated 10 Mbit/second Ethernet network.

Some speed-up values for the MT-Xolas implementation are greater than the number of processors. This may be explained by the reduced amount of memory swapping that takes place when the number of threads increases.

For the range of five processors used in the tests, the speed-up curve of the PVM-SGI implementation tends to a value of 2, while the speed-up curve of the PVM-Xolas is linear. Assume that one is interested in predicting the speed-ups that would be obtained if one had more processors in the Xolas machine and in the network of SGI workstations. The burden curves should answer this question.

Figure 2 shows the burden curves of the PVM-Xolas and PVM-SGI implementations. If the PVM-SGI burden is approximated by a linear curve, its slope would be approximately equal to 0.04. This suggests a maximum speed-up at around five processors, which is consistent with the experimental results reported on the PVM-SGI speed-up curve. On the other hand, the PVM-Xolas burden curve suggests that the maximum speed-up on a Xolas machine would be asymptotically approached with more processors, and that its values would be approximately 50 (1/0.02). Note that if one had 50 processors on a Xolas machine, a speed-up of only 25 would be attained. In such case, only half the time, a process would be busy doing computations specified in the sequential algorithm.

The results in Figs 1 and 2 were obtained assuming that the shortest-path results are sent back to the master process. We now analyze the effect on parallel performance of this communication task. The analysis is useful for network problems where the results may not need to be communicated back to the master process.

Figure 3 shows the speed-up curves of the PVM-Xolas and PVM-SGI implementations, when slave processes do not send back the computational results to the master process. The speed-up curve of the MT-Xolas implementation is not affected by this experiment, as it does not involve exchange of results, but was included in Fig. 3 for comparison only. One can note the improvements in the speed-ups for

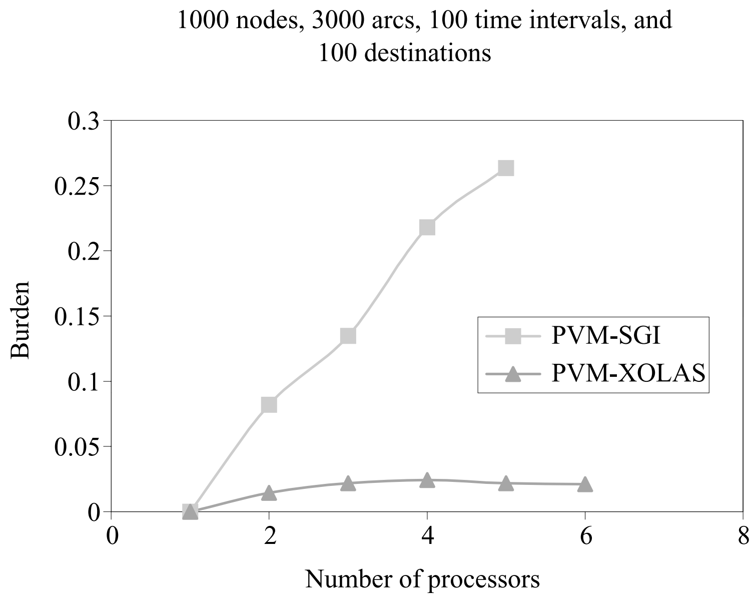

Fig. 2. Burden curves of the PVM parallel implementations based on decomposition by destinations.
Note: The networks used in these tests contain 1000 nodes, 3000 links, 100 time intervals, and 100 destination nodes.

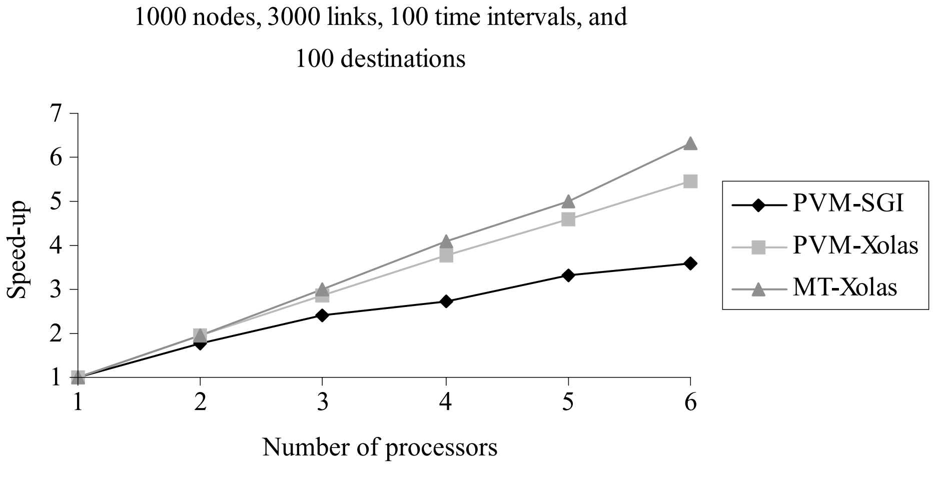

Fig. 3. Speed-up curves of the parallel implementations based on decomposition by destinations, without collection of shortest path results.
Note: The network used in these tests contains 1000 nodes, 3000 links, 100 time intervals, and 100 destination nodes.

the PVM-SGI implementation, and hence its parallel running time, while there was not noticeable improvement in the PVM-Xolas implementation. In Proposition 4, it is shown that the second term in the parallel run-time analysis is the maximum among a computational term and of a communication delay term. The communication speed on the SGI network is lower that the communication speed on the Xolas machine. In the PVM-SGI implementation, the second term in the run-time analysis appears to be due to the communication part, while in the PVM-Xolas implementation it appears to be due to the computation part. The term that dominates is of course dependent on the values of the parameters n, m, M, and p .

7.4. Numerical results for parallel implementations based on the decomposition by network topology

Figure 4 shows the speed-up curves for the parallel implementations of algorithm DOT using the network-topology decomposition. The PVM implementations for two processors and more have lower running times than the run-time obtained on one processor (the latter run-time is approximately the same as the running time of the sequential implementation of algorithm DOT). The communication requirements among slave processes, and between the master process and the slave processes are too high to obtain a speed-up greater than one.

The speed-up curve of the MT-Xolas implementation shows satisfactory speed-ups, which are however lower than the ideal speed-ups. The idle time due to the synchronization barrier function, as well as the potential time overhead due to contentions of threads requiring access to the same memory location, is a possible explanation of the differences between observed and ideal speed-ups.

The burden curve of the MT-Xolas implementation shown in Fig. 5 suggests that the maximum speed-up would be asymptotically approached with more processors, and that the value of the asymptotic speed-up would be 20 (the inverse of 0.05, which is the maximum value of the burden).

The numerical results indicate that only the MT-Xolas implementation, based on a network-topology decomposition, led to speed-ups of the sequential running time of algorithm DOT. The latter running time depends on the following network parameters: number of nodes, number of links, and number of time intervals. In the rest of this subsection we analyze the effect of these parameters on the computational performance of the MT-Xolas implementation.

Figure 6 shows speed-up curves of the MT-Xolas parallel implementation of algorithm DOT for three values of the number of time intervals. The time taken by the synchronization barrier function

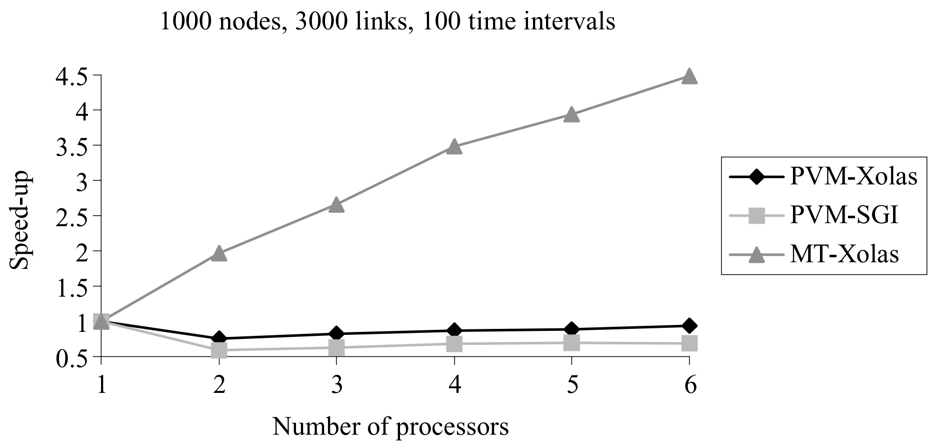

Fig. 4. Speed-up curves of the parallel implementations based on decomposition by network topology. Note: The network used in these tests contains 1000 nodes, 3000 links, and 100 time intervals.

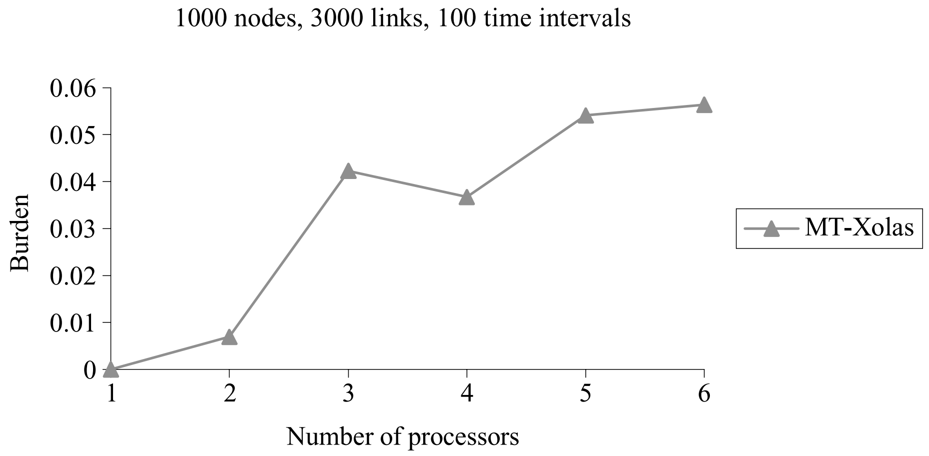

Fig. 5. The burden curve of the MT-Xolas parallel implementation of algorithm DOT based on decomposition by network topology.
Note: The network used in these tests contains 1000 nodes, 3000 links, and 100 time intervals.

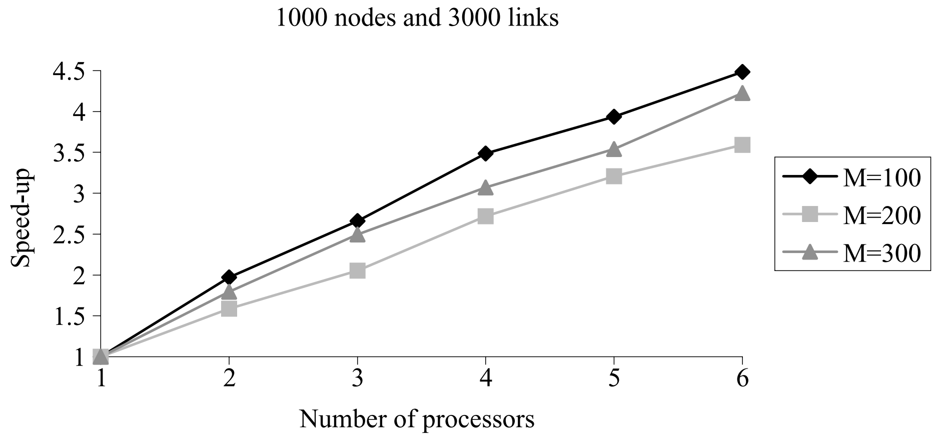

Fig. 6. Speed-up curve of the MT-Xolas implementation of algorithm DOT for three values of the number of time intervals (M = 100, 200 and 300).
Note: The network used in these tests contains 1000 nodes and 3000 links.

should increase with the number of time intervals. The speed-up is then a decreasing function of the number of time intervals. This analysis is consistent with the results obtained in Fig. 6.

The average number of potential contentions to memory locations should increase with the number of arcs leaving out of a given subset of nodes assigned to a given processor. This would happen under the following two scenarios: (1) the number of arcs is kept constant while the number of nodes decreases, and (2) the number of nodes is kept constant while the number of arcs increases. This would explain the trends shown in Figs 7 and 8, where speed-ups appear to be an increasing function of the

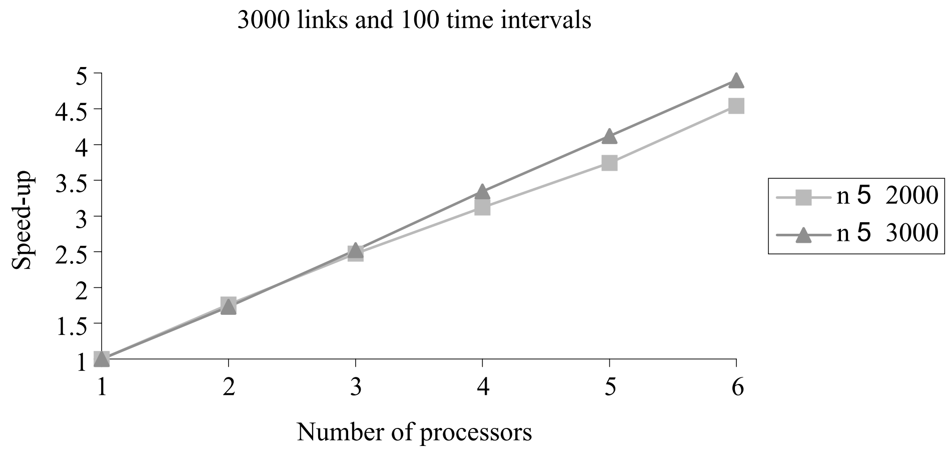

Fig. 7. Speed-up curve of the MT-Xolas implementation of algorithm DOT for two values of the number of nodes (n = 2000, n = 3000).
Note: The network used in these tests contains 3000 links and 100 time intervals.

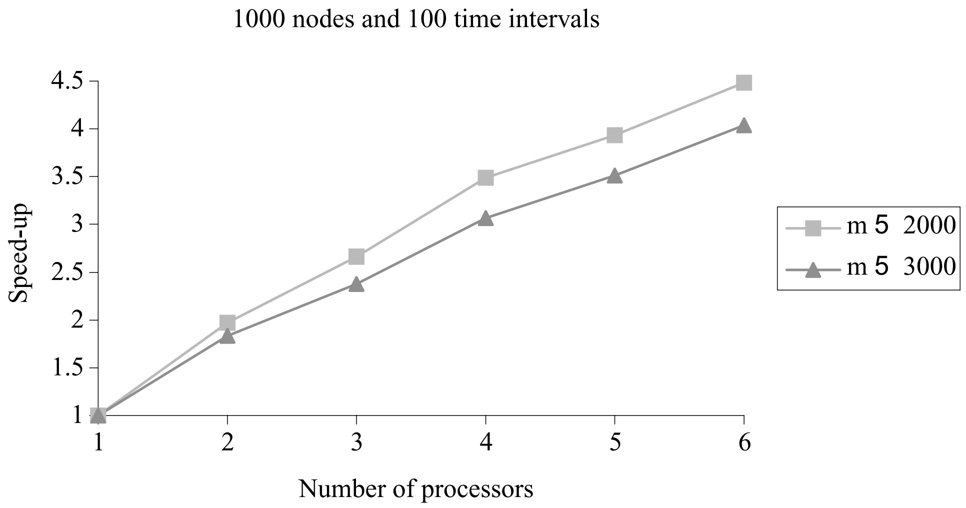

Fig. 8. Speed-up curve of the MT-Xolas implementation of algorithm DOT for two different values of the number of arcs (m = 2000, m = 3000).
Note: The network used in these tests contains 3000 links and 100 time intervals.

number of nodes, for a constant number of arcs, and a decreasing function of the number of arcs, for a constant number of nodes.

Acknowledgments

This research was supported by the US National Science Foundation (NSF) and by the US Department of Transportation (DOT). The NSF support was under the CAREER Award Grant number CMS-

9733948. The DOT support was under contracts DTRS99-G-0001 and DTRS95-G-0001 to the New England (Region One) University Transportation Centers Program at MIT.

References

Bertsekas, D., Tsitsikilis J., 1989. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Englewood Cliffs.

Chabini, I., 1997. A New Shortest Paths Algorithm for Discrete Dynamic Networks. *Proceeding of 8th IFAC Symposium on Transport Systems*, 551–556.

Chabini, I., 1998. Discrete Dynamic Shortest Path Problems in Transportation Applications: Complexity and Algorithms with Optimal Run Time. *Transportation Research Record* 1645, 170–175.

Chabini, I., Dean, B., 1999. Shortest Path Problems in Discrete-Time Dynamic Networks: Complexity, Algorithms, and Implementations. Internal Report, MIT, Cambridge, USA.

Chabini, I., Gendron, B., 1995. Parallel Performance Measures Revisited. Proceedings of *High Performance Computing Symposium 95*, Montreal, Canada, July 10–12.

Chabini, I., Florian, M., Le Saux, E., 1997. High Performance Computation of Shortest Routes for Intelligent Transportation Systems Applications. *Proceedings of the Second World Congress of Intelligent Transport Systems '95*, Yokohama, 2021–2026.

Chabini, I., He, Y., 1999. An Analytical Approach to Dynamic Traffic Assignment: Models, Algorithms and Computer Implementations. Internal Report, MIT, Cambridge, USA.

Cooke, K., Halsey, E., 1966. The Shortest Route Through a Network with Time Dependent Internodal Transit Times. *Journal of Math. Anal. Appl.* 14, 492–498.

Ganugapati, S., 1998. Dynamic Shortest Paths Algorithms: Parallel Implementations and Application to the Solution of Dynamic Traffic Assignment Models. M.S. Thesis, Department of Civil and Environmental Engineering, MIT.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchak, R., Sunderam, V., 1995. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, MA.

Habbal, M., Koutsopolous, H., Lerman, S., 1994. A Decomposition Algorithm for the All-Pairs Shortest Path Problem on Massively Parallel Computer Architectures. *Transportation Science*, 28(4), 292–308.

Hribar, M., Taylor, V., 1996. Reducing the Idle Time of Parallel Transportation Applications. *Submitted to the International Parallel Processing Symposium*.

Karypis, G., Kumar, V., 1998. Multilevel Algorithms for Multi-Constraint Graph Partitioning. Technical Report #98-019, Department of Computer Science, University of Minnesota.

Lewis, B., Berg, D., 1996. *Threads Primer: A Guide to Multithreaded Programming*. Prentice Hall, Upper Saddle River, NJ.

Ziliaskopoulos, A., Mahmassani, H., 1993. A Time-Dependent Shortest Path Algorithm for Real-Time Intelligent Vehicle/ Highway System. *Transportation Research Record* 1408, 94–104.

Ziliaskopoulous, A., Kotzinos, D., Mahmassani, H., 1997. Design and Implementation of Parallel time dependent least time path algorithms for Intelligent Transportation Applications. *Transportation Research Part C* 5(2), 95–107.