
ELSEVIER

Parallel Computing 27 (2001) 1537–1568

PARALLEL COMPUTING

www.elsevier.com/locate/parco

# Implementing parallel shortest path for parallel transportation applications

Michelle R. Hribar [a], Valerie E. Taylor [b,*], David E. Boyce [c]

[a] *Pacific University, Mathematical Sciences Department, 2043 College Way, Forest Grove, OR 97116, USA*
[b] *Northwestern University, ECE Department, 2145 Sheridan Road, Evanston, IL 60208-3118, USA*
[c] *Department of Civil and Materials Engineering, University of Illinois at Chicago, 842 West Taylor Street, Chicago, IL 60607-7023, USA*

Received 26 October 2000

**Abstract**

Shortest path algorithms are required by several transportation applications; furthermore, the shortest path computation in these applications can account for a large percentage of the total execution time. Since these algorithms are very computationally intense, parallel processing can provide the compute power and memory required to solve large problems quickly. Therefore, good parallel shortest algorithms are critical for efficient parallel implementations of transportation applications. The experimental work related to parallel shortest path algorithms has focused on the development of parallel algorithms; however, very little work has been done with analyzing and understanding the performance impact of various implementation issues. In this paper, we conduct a thorough experimental analysis of parallel shortest path algorithms for sparse networks, concentrating on three implementation issues: (1) choice of shortest path algorithm, (2) termination detection and (3) network decomposition. The paper focuses on the choice of shortest path algorithm and network decomposition since the work on termination detection was published previously. We determine that all three issues affect the communication and convergence of the shortest path algorithm. Furthermore, we find that communicating the most information at a time results in the best convergence; this is contrary to most scientific applications where it is optimal to minimize communication. © 2001 Published by Elsevier Science B.V.

*Keywords:* Parallel transportation applications; Parallel shortest path; Network decomposition; Shortest path algorithms; Parallel performance; Traffic equilibrium

* Corresponding author.
*E-mail addresses:* hribarm@pacificu.edu (M.R. Hribar), taylor@ece.nwu.edu (V.E. Taylor), DBoyce @uic.edu (D.E. Boyce).

0167-8191/01/$ - see front matter © 2001 Published by Elsevier Science B.V.
PII: S0167-8191(01)00105-3

## 1. Introduction

Shortest path computation is an integral component of many applications, including transportation applications. For example, traffic equilibrium problems using the Frank–Wolfe algorithm require a shortest path solution for the assignment of traffic flow during each iteration of the algorithm. Consequently, this shortest path computation accounts for a large percentage of the overall execution time of the traffic equilibrium application. For example, using one processor of the Intel Paragon the shortest path solution step accounts for up to 95% of the total execution time of a traffic equilibrium problem for a 9701-node traffic network [18].

Traffic equilibrium applications are very large and computationally intense. Parallel processing provides the memory and computational power needed to solve the equilibrium problems in a reasonable amount of time. The primary goal of using parallel machines for traffic equilibrium applications is to reduce the execution time so that it is possible to conduct more in-depth analyses. Increases in uniprocessor speeds are significant, but insufficient for the computing requirements of traffic problems. For example, solving a dynamic equilibrium problem consisting of 18 time intervals (corresponding to 3 h of traffic flow) for a 9701-node traffic network requires 55 h (or 2.3 days) to complete on one 60 MHz, Convex C3880 processor [6]. Using this same processor for 10 h of traffic flow would require 183.3 h (or 7.6 days). Extrapolating from this experiment, we estimate that using a 800 MHz processor would reduce the execution time at most to approximately a half a day under ideal conditions. By a similar calculation, using 500 parallel processors could reduce the execution time to under 30 min. Hence, parallel processing is beneficial for this class of applications; furthermore, it is necessary to focus on optimizing the parallel shortest path algorithms in order to improve the overall execution time of the application.

Solving for the shortest paths in transportation applications is typically accomplished by repeatedly applying single-source labeling algorithms for each source, because of the sparsity of urban traffic networks [11–13,19,29]. Simply stated, labeling algorithms consist of iteratively updating node labels, which represent the shortest path distance at the end of the algorithm. There are many variants of the labeling algorithm, each updating the node labels in a different order. In this paper, we consider three simple labeling algorithms, focusing on the performance impact of three different implementation issues: (1) choice of algorithm, (2) termination detection and (3) network decomposition.

The focus of this paper is on the network decomposition and choice of shortest path algorithm since the details of the termination detection implementation were published in [20]; a brief summary of termination detection is presented for completeness. Each of the implementation issues affects the performance greatly. A poor termination detection algorithm can take twice as long to complete as a good implementation; a poor network decomposition can result in execution times that are orders of magnitude greater than a good decomposition; and the different variations of the labeling shortest path algorithm can vary in execution time by as much as 50%.

We present background information about traffic equilibrium, traffic networks, parallel systems and previous work in Section 2; we describe the shortest path algorithms in Section 3; the network decomposition in Section 4; the choice of algorithm in Section 5; and we summarize in Section 6.

## 2. Background

### 2.1. Traffic equilibrium

Traffic engineers use several different approaches for predicting traffic flow on a given traffic network. One approach is to determine what the steady-state flow on a given network may be. Transportation scientists assume that traffic flows tend toward an equilibrium state; predicting this state allows the engineers to analyze and plan traffic systems. There are several definitions of equilibrium, as well as static and dynamic models for each type of equilibrium. This study focuses on static user equilibrium. At user equilibrium, travel times of used paths between any two given nodes are equal and no unused path has a travel time that is less than that of a used path. Thus, a single user cannot unilaterally improve his/her travel time by changing paths [32].

#### 2.1.1. Network terminology

Networks are a set of $n$ nodes connected by $m$ directed arcs. In urban traffic networks, the nodes generally correspond to intersections and the arcs correspond to road segments. Urban traffic networks are sparse networks; that is, the nodes are connected to a constant number of other nodes. Therefore, the number of arcs is $O(n)$ instead of $(n^2 - n)$ as is the case for dense, fully connected networks. Urban traffic networks have groups of nodes and arcs called zones. These are the origins and destinations of all the trips in the network and are the sources (zones) and destinations of the shortest path solutions. Typically, the number of sources, $z$, is much less than the number of nodes. Each arc in the urban traffic network has a corresponding non-negative cost which is the travel time for that arc.

#### 2.1.2. Formal definition of user equilibrium

The problem formulation for the user equilibrium problem is stated in Fig. 1. While the formulation is not intuitive, the optimal solution fulfills the definition of user-equilibrium. This is shown by the Karash–Kuhn–Tucker conditions that are necessary for the existence of an optimal solution [32]:

$$T_r - T_r^* \geqslant 0 \quad \forall r \in R_{ij}, \tag{1}$$

$$h_r(T_r - T_r^*) = 0 \quad \forall r \in R_{ij}, \tag{2}$$

where $T_r^*$ is the minimum of $T_r$ over all $r \in R_{ij}$. Eq. (1) states that the travel times of each route is greater than or equal to the minimum travel time of that route. Eq. (2) implies that the flow $h_r \geqslant 0$ for routes whose time is equal to the minimum or op-

$f$ = $(f_1, ..., f_m)$
the flow on each of the $m$ arcs
$tc$ = $(tc_1(f_1), ..., tc_m(f_m))$
the travel cost functions
$R_{ij}$ = the set of routes from origin $i$ to destination $j$
$R$ = the set of all routes from all origins to all destinations, $\{R_{11}, R_{12}, ...R_{nn}\}$
$F_{ij}$ = the demand flow from origin $i$ to destination $j$
$h_r$ = the flow on route $r \in R$
$\delta_{lr}$ = $\begin{cases} 1 & \text{if arc } l \text{ belongs to route } r \in R \\ 0 & \text{otherwise} \end{cases}$
$T_r$ = travel time on route $r \in R$

$$\begin{aligned} \text{minimize} \quad & \sum_l \int_0^{f_l} tc_l(x)\, dx \\ \text{subject to} \quad & \sum_{r \in R_{ij}} h_r = F_{ij} \; \forall i, j \\ & h_r \geq 0 \quad \forall r \in R \\ & f_l = \sum_{r \in R} \delta_{lr}\, h_r \quad l = 1, ..., m \end{aligned}$$

Fig. 1. User-equilibrium problem statement.

timal time. For those routes whose time is greater than the optimal, the flow $h_r$ equals 0. This is equivalent to the definition of user-equilibrium.

### 2.1.3. Frank–Wolfe solution algorithm

One algorithm commonly used to solve for user-equilibrium is the Frank–Wolfe algorithm (convex combination method) [32] given in Fig. 2. Steps 1, 3 and 4 of the algorithm require O($m$) computation time (recall $m$ is the number of arcs), while Step 5 requires a constant amount of time. Step 2 requires solving a shortest path tree for the $z$ sources in the network. The amount of time required by the shortest path solution depends on the particular algorithm used, but it is significantly greater than O($m$). As stated previously, the shortest path solution accounts for a large percentage of the total execution time of the Frank–Wolfe algorithm. For this rea-

1. Compute the current travel cost vector $tc_l^k = tc_l^j(f_l^k)$ for $l = 1,...,m$ where $k$ is the number of the Frank-Wolfe iteration.
2. Find a new direction $y_l^k$ using all-or-nothing assignment with the arc costs $tc_l^k$.
3. Determine the optimal step size $\lambda^k$ by minimizing the objective function of user-equilibrium problem. This is done using a gradient technique or bisection method.
4. Compute a new solution $f_l^{k+1} = f_l^k + \lambda(y_l^k - f_l^k), l = 1, ...m$.
5. If the convergence criterion is satisfied, then $f_l^{k+1}$ is the solution. Otherwise, $k = k + 1$ and return to step 1.

Fig. 2. Frank–Wolfe algorithm.

son, parallelizing traffic equilibrium problems requires an efficient parallel shortest path algorithm.

### 2.2. Parallel systems

Loosely defined, parallel computing includes all computing that uses at least two computers to simultaneously execute a given application. Parallel computing today incorporates a wide variety of computing architectures and programming methodologies. These range from networks of workstations and clusters of PC's to massively parallel machines with thousands of processing nodes. These platforms are often characterized by how they treat memory. If the memory is seen as one large global address space accessible by all the processing nodes, then the machine is considered a shared memory machine. Otherwise, if the memory is seen as distributed among the processing nodes such that explicit communication is required for accesses to non-local memory, then the machine is considered to be message-passing. We concentrate on the latter category of parallel machines; however, the concepts are applicable to shared memory machines as well.

### 2.3. Previous work in parallel shortest path

Most of the published work in parallel shortest path is theoretical in nature, proving the worst case execution times. There has been some work on experimental studies of parallel algorithms, including parallel label-setting and label-correcting. This work has focused on a comparison of execution times, where our work focuses on developing new insight into why the algorithms perform as they do.

There have been a few experimental studies of distributed label-setting algorithms. Adamson and Tick [1] and Traff [35] use a distributed shared memory machine to solve label-setting algorithms and partition the network among processors such that each processor works on its own subnetwork. They observed that label-setting algorithms have little parallelism since only one node with the minimum distance label is extracted from the queue at a time. Traff improves performance by allowing the processors to remove more than one node at a time even if the node does not have the smallest label, thus no longer performing a strict label-setting algorithm. Romeijn and Smith [31] attempt to decrease the communication requirements for a label-setting algorithm by solving for an approximate shortest path tree on a distributed network. However, not solving for the exact shortest paths may affect the convergence of the transportation application.

Similarly, there has been some experimental work done in the area of parallel label-correcting algorithms. Bertsekas et al. [5] provided an experimental comparison of several label-correcting algorithms on a shared memory machine with eight processors. Some of the algorithms achieved superlinear speedup, while others achieved no speedup at all. Only a comparison of algorithms' resultant execution times and speedup were given; no detailed analysis was provided to identify the bottlenecks that affect performance. Papaefthymiou and Rodrigue [28] implemented Bellman–Ford–Moore algorithms on the CM-5. They compared the difference

between a coarse grained and a fine grained implementation. They use simple partitioning schemes and acknowledge more work needs to be done to develop more sophisticated partitioning techniques. Habbal et al. [14] presented an all-pairs shortest path algorithm for a network divided into subnetworks. The shortest paths are solved by solving for the all-pairs shortest path within the local subnetwork and then over the entire network.

In summary, the previous work has focused on developing and comparing different parallel label-correcting and label-setting algorithms. Our work provides an in-depth analysis of the implementation of these algorithms in order to determine the most effective way to perform the termination detection, network decomposition and to choose between algorithms.

### 2.4. Data sets

The traffic networks used in this paper consist of five urban networks: Waltham, and New York City [5], Chicago North-Northwest Suburbs [17], Chicago Area Transportation Study (CATS) 1996 TIP network [9] and Chicago ADVANCE Test Area [4]. We also generated 12 grid networks and assigned random travel times to the arcs. These grids are intended to be similar to traffic networks in their sparsity. We added diagonal arcs radiating from the center of the grid networks to represent more accurately the traffic networks. For this reason, some of the rectangular grids with the same number of nodes do not have the same number of arcs. The grids range in size from $33 \times 33$ grid (1089 nodes) to $257 \times 257$ grid (66,049 nodes). Six of these networks are square and six are rectangular. The number of nodes and arcs for each of the networks is given in Table 1.

Table 1
Test networks

| Network name | Number of nodes | Number of arcs |
|---|---|---|
| New York City | 4795 | 16,458 |
| N/NW Suburbs | 9628 | 30,506 |
| ADVANCE | 9701 | 22,918 |
| CATS | 15,949 | 36,042 |
| Waltham | 26,347 | 64,708 |
| $33 \times 33$ Grid | 1089 | 4288 |
| $33 \times 65$ Grid | 2145 | 8512 |
| $65 \times 33$ Grid | 2145 | 8448 |
| $65 \times 65$ Grid | 4225 | 16,768 |
| $65 \times 129$ Grid | 8385 | 33,408 |
| $129 \times 65$ Grid | 8385 | 33,280 |
| $129 \times 129$ Grid | 16,641 | 66,304 |
| $161 \times 161$ Grid | 25,921 | 103,360 |
| $129 \times 257$ Grid | 33,153 | 132,352 |
| $257 \times 129$ Grid | 33,153 | 132,096 |
| $193 \times 193$ Grid | 37,249 | 148,608 |
| $257 \times 257$ Grid | 66,049 | 263,680 |

The five traffic networks are a representative sample of how the algorithms perform on real urban networks. The 12 grid networks provide insight as to how network size affects the algorithms performance on sparse networks. These grid networks allow us to easily vary the size of the network from a small number of nodes (1089) to a very large number of nodes (66,049).

## 3. Shortest path labeling algorithms

### 3.1. Labeling algorithms

The shortest path problem is a well researched area and many approaches have been developed to solve this problem. We focus on iterative labeling algorithms since they are generally used to solve for shortest paths on transportation networks and other sparse networks [12,13]. These algorithms generate shortest path trees rooted at the different sources in the network. Each node $i$ has a label $dl[i]$ that represents the smallest known distance from the source to node $i$. Each node's label is iteratively updated until the end of the algorithm when it is equal to the shortest distance from the source to node $i$. Fig. 3 gives a general labeling algorithm. The *list* in the algorithm contains nodes whose distance labels are not guaranteed to be the shortest path distance. During each iteration a node is removed from the list and its adjacent nodes' labels are updated. How these nodes are removed from the *list* determines whether the algorithm is label-setting or label-correcting.

```
General Labeling Algorithm
  begin
    dl[s] = 0;
    pred[s] = 0;
    for i = 1..n, i ≠ s do
      dl[i] = ∞;
    end for
    while list ≠ ∅
      remove i from list;
      for each edge (i, j) ∈ A[i] do
        if dl[j] > dl[i] + c_ij then
          dl[j] = dl[i] + c_ij;
          pred[j] = i;
          if j ∉ list then
            list = list ∪ {j};
          end if
        end if
      end for
    end while
  end
```

Fig. 3. General labeling algorithm.

Label-setting algorithms remove from the *list* the node with the smallest distance label during each iteration of the algorithm. If the arc costs are non-negative, the node that is removed will never enter the list again. The time complexity of the algorithm depends on how the *list* is stored and/or how the minimum label is found. The basic algorithm is attributed to Dijkstra [8], but in this algorithm no sorted data structure is used to store the list, so the time complexity is $O(n^2)$. In this paper, we use balanced binary trees [7] to store the list. This reduces the execution time to $O(m \log n + n \log n)$.

Label-correcting algorithms, on the other hand, do not necessarily remove the node with the minimum label from the *list*. Therefore, a removed node may re-enter the *list* at a later time. Since the labels are not "set" as in the label-setting algorithms, the distance labels are not guaranteed to be correct after $n$ steps. Distance labels are all correct when for every arc $(i, j)$, $dl[j] \leqslant dl[i] + c_{ij}$. The order in which the nodes are chosen from the *list* affects the number of node removals needed to complete the algorithm. In terms of worst-case performance, the complexity bounds depend on the method for removing and adding nodes to the *list*. If the nodes are selected in a first-in first-out (FIFO) order from the *list*, the algorithm is equivalent to that of Bellman–Ford–Moore [3,10,25]. These researchers are attributed with the first polynomially bounded label-correcting algorithm, which has a worst-case behavior of $O(nm)$. In this paper, we refer to this algorithm as the one-queue algorithm. A variant of this algorithm by Pallottino uses two queues to store the *list* [26,27]. If a node has been seen before, it is put in the first queue; otherwise, it is put in the second. Nodes are removed first from the first queue, then the second queue. The idea of the algorithm is that nodes that have been seen and updated already should be updated again before nodes that have yet to be seen. The worst-case bound of the algorithm is $O(n^2m)$.

In this paper, we focus on three basic shortest path algorithms: label-setting implemented with balanced binary trees (hereafter referred to as LS), and two commonly used label-correcting algorithms – one-queue (Bellman–Ford–Moore, hereafter referred to as LC-1) and two-queue (Pallottino, hereafter referred to as LC-2). We focus on these particular algorithms to show the difference between label-setting and label-correcting, and the difference between two widely used label-correcting algorithms. We study only one label-setting algorithm since the difference between algorithms in this class is the data structure used for the sorted list of labels. We use one of the most efficient data structures – balanced binary trees. In contrast, there is more variance between label-correcting algorithms, in particular the criteria for selecting nodes from the list, the placement of nodes in the list and the number of lists. Therefore, we study two widely used label-correcting algorithms. Table 2 summarizes the theoretical differences between the algorithms.

Table 2
Summary of theoretical bounds of the three shortest path algorithms

| Algorithm | Node removal | Node update | Total time |
|---|---|---|---|
| LS | $O(\log n)$ | $O(\log n)$ | $O((n+m)\log n)$ |
| LC-1 | $O(1)$ | $O(1)$ | $O(nm)$ |
| LC-2 | $O(1)$ | $O(1)$ | $O(n^2m)$ |

Despite their large theoretical bounds, label-correcting algorithms have been shown to have better experimental performance than label-setting for sparse networks [11–13,19,29]. The label-correcting algorithms rarely attain their worst case behavior, but predicting their expected behavior is difficult. The label-setting algorithms' behavior, on the other hand, is very predictable. This fact will be very important for the performance of the parallel implementation as will be shown in the discussion of the choice of algorithm in Section 5.

### 3.2. Parallel shortest path

There are two approaches for solving multiple-source shortest paths using labeling algorithms on distributed-memory machines: (1) network replication [24] and (2) distributed network [1,5,14,28,31,35]. The first approach, network replication, assigns a copy of the entire traffic network to each processor, which then solves the shortest path tree for a subset of sources. This approach requires no interprocessor communication for the shortest path calculation, but subsequent steps in the application that use the shortest path information will require all-to-all communication to update the replicated network. The drawback to this is its limited scalability. First, the algorithm does not take advantage of the large aggregate memory of a distributed-memory machine; instead, the size of the traffic network is limited by the amount of local memory available on a single processor. Second, the number of processors is limited to the number of sources. Additionally, the time needed for the global communication of the entire traffic network increases as the number of processors increases.

The second approach, solving a single-source shortest path for each source on a distributed network, partitions the traffic network into $P$ subnetworks, one for each of $P$ processors. Each processor is responsible for computing the shortest path on its local subnetwork. The advantages of the distributed network approach are that it takes advantage of the aggregate memory of the parallel system and the amount of parallelism can extend beyond the number of sources. Further, no all-to-all communication is required for any of the steps in the traffic application, including the shortest path computation. The disadvantage is the communication required for communicating node labels and for termination detection.

We conducted an experiment comparing the two approaches appropriate for sparse networks: network replication and distributed network. The goal of the experiment was to quantify the difference between the all-to-all communication required for the update step in the traffic equilibrium when using the network replication and the local communication required for the shortest path step using the distributed network. The two approaches were executed on the Intel Paragon at the California Institute of Technology. The results indicated that the distributed approach requires about half the execution time of the replicated approach when we have 16 or more processors. Hence, this paper focuses on the distributed network approach.

The distributed parallel shortest path algorithm uses the single program, multiple data (SPMD) model for which each processor solves the portion of the shortest path

```
Initialize node labels to ∞;
If source is in local subnetwork then
     insert it in list;
end if
Repeat
    while list ≠ ∅
         Compute shortest path for subnetwork;
    end while
    Send updated boundary node labels to neighbors;
    Receive updated boundary node labels from neighbors and put in list;
    Detect termination condition: all processors' lists = ∅ and no unreceived messages;
Until termination detected
```

Fig. 4. Parallel shortest path algorithm.

tree on its subnetwork for all sources as given in Fig. 4. Each processor repeatedly solves for shortest paths for its assigned subnetwork using a serial shortest path algorithm. Distributing the traffic network among $P$ processors requires that the network be partitioned into $P$ subnetworks, each of which is assigned to a processor. Multiple processors share nodes; these nodes are called *boundary nodes*. The remaining nodes are called *interior nodes*. Since each processor has information about its local subnetwork only, communication must occur between processors in order to compute the shortest path trees. Only distance label information about the boundary nodes needs to be communicated during the parallel algorithm since any path to each interior node that contains nodes on other processors must include a boundary node. Similar to the serial algorithm, termination occurs when all processors have no remaining work to do, that is when all the processors' lists are empty.

The shortest path algorithm is complete when all the node labels are correct. In the serial shortest path labeling algorithm, each node is placed in a list after its label has just changed. Termination is ensured when the list is empty [2,25,26]. In the parallel algorithm, when a node label changes, it is placed into the processor's list. If it is a boundary node, it is also placed into a message. This leads to the following claim identifying the condition for all the node labels are correct in the parallel algorithm:

**Claim 1.** *All processors are finished, i.e., all the node labels are correct, when the following conditions are satisfied*:

(1) *All processors' lists are empty*.

(2) *There are no unprocessed messages in communication network*.

**Proof.** By contradiction.

Suppose node $j$'s label is incorrect and (1) and (2) hold. From the proof of the serial shortest path algorithm [2,25,26] this means that at least one ancestor of node $j$ is in the list. In parallel, this means that the ancestor must be in a processors local list or in a message. This contradicts the initial supposition. □

The correctness of the distributed parallel algorithm follows from Claim 1.

### 3.3. Implementation issues

There are three main implementation details that affect the performance of the shortest path algorithms: implementation of the termination detection step, the network decomposition and the algorithm for the local shortest path computation. The network decomposition and algorithm are discussed detail in Sections 4 and 5; the details of the termination detection step are given in [20]. For completeness, we provide a brief summary of this work in the following paragraphs.

In the shortest path algorithm, the processors must perform a synchronization step to determine if the termination criteria given in Claim 1 are satisfied. The challenge of the implementation of termination detection is to determine when detection should occur and how frequently. If the processors do not reach the synchronization step at the same time, idle time will be incurred. For example, one implementation which did not carefully synchronize had an execution time that was three times longer than the most efficient implementation.

We analyzed four different implementations of the termination detection step that explored different detection and communication frequencies. The analysis indicates that low detection frequency is best for small numbers of processors (less than 64); as the number of processors increase high detection frequency becomes best. High communication frequency is always best.

### 3.4. Experimental setup

In this paper, we present the results of our studies of the two latter implementation issues, network decomposition and choice of shortest path algorithm. For both of these studies, we used the Intel Paragon at the California Institute of Technology. Because of space limits, we do not present all the results in both sections; instead, we present illustrative examples using a particular network and number of processors. We solve for 32 sources for all the networks for ease of analysis. In transportation algorithms, however, the number of sources would be much greater – on the order of hundreds. Furthermore, the shortest path solution is computed during each iteration of the algorithm, where there are typically 10–20 iterations total. Therefore, the relative difference between shortest path algorithms given in subsequent sections would be much greater when included in the solution of the transportation applications. Furthermore, for extensions to the standard equilibrium problem, such as solving for dynamic instances of equilibrium, the number of iterations greatly increases. This makes it even more essential to optimize the shortest path computation as much as possible. Consequently, we focus on the difference in execution times of the different implementations of the algorithms.

## 4. Network decomposition

Network decompositions for shortest path problems affect both the communication and computation time of the algorithm. This is different from most parallel

Table 3
$129 \times 257$ Grid LC-2 updates, 32 sources, 16 processors

| Decomposition | Parallel LC-2 updates |
|---|---|
| 1 | 7,799,371 |
| 2 | 7,154,577 |
| 3 | 10,856,987 |
| 4 | 8,067,494 |
| 5 | 10,651,603 |
| 6 | 2,776,602 |
| 7 | 10,701,668 |
| 8 | 3,780,505 |
| 9 | 3,627,503 |
| 10 | 8,206,039 |
| 11 | 8,730,665 |

applications where only the communication time is affected. In shortest path problems the network decomposition affects the order in which the node labels are updated and hence, affects the convergence of the algorithm. For example, Table 3 gives the total number of updates (the amount of computation) required for 11 different decompositions of the $129 \times 257$ grid network on 16 processors. The details of these different decompositions will be presented shortly. As shown in the table, the number of total updates varies greatly for the different decompositions. For example, decomposition 6 performs the minimum number of updates, 2,776,602, which is about one-fourth the maximum number of updates, 10,856,987 (decomposition 3).

### 4.1. Decomposition characteristics

To understand how different decompositions produce different results, we first need to determine how the network decompositions differ. In this paper, we consider five different characteristics of the subnetworks as the defining characteristics. These characteristics are: number of boundary nodes, number of interfaces, number of boundary nodes per interface, subnetwork connectivity and subnetwork diameter. These different characteristics are illustrated in Fig. 5.

- *Number of boundary nodes*: Boundary nodes form the cuts across the network which separate the network into subnetworks. As described previously, the labels of these nodes are those that are communicated between processors. The number of these nodes affects the communication time as well as the computation time. The larger the number of boundary nodes, the fewer the number of interior local nodes and thus, the fewer the number of node labels that can be updated locally without communication.
- *Number of interfaces*: An interface in a subnetwork is the group of nodes that is shared with another single processor. For example, if a network is decomposed into strips, each processor has at most two interfaces, as illustrated in the example decomposition shown in Fig. 5. The number of interfaces determines the

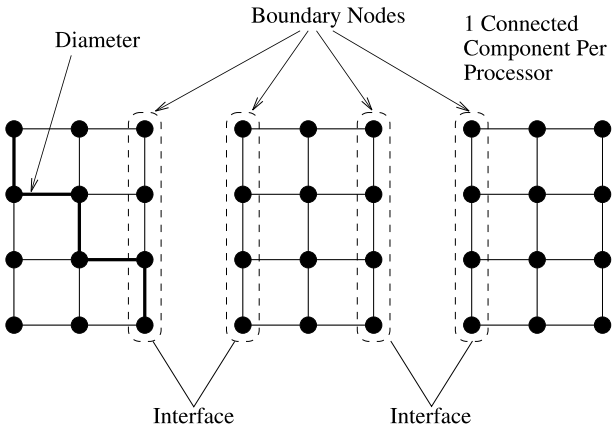


Fig. 5. Decomposition characteristics.

maximum number of messages that are sent during an iteration of the shortest path algorithm.

- *Number of boundary nodes per interface*: The number of boundary nodes per interface is the number of nodes that a processor shares with a given neighbor. This ratio determines the amount of information that a processor receives in a given message and upon which it will base its subsequent computation.
- *Subnetwork connectivity*: The subnetwork connectivity refers to the number of connected components per subnetwork. If the subnetworks are a group of disconnected sets of nodes, the algorithm's computation time is orders of magnitude more than with subnetworks that are fully or mostly connected (i.e. have very few connected components) [18]. This phenomenon within the subnetwork occurs because very little local computation can be performed between computation steps since very few nodes can be reached within the local subnetwork. This results in the computation being very dependent on communication and the order in which the messages are received.

  Disconnected subnetworks have large numbers of boundary nodes and interfaces, and a small ratio of boundary nodes to interfaces. Therefore, because of the great advantage of connected networks, we consider decomposition methods that produce subnetworks that are fully or mostly connected like the one shown in Fig. 5.
- *Subnetwork diameter*: The diameter of a network is the largest number of arcs that separate any two nodes within the network as shown in Fig. 5. For the decomposed network, we define the subnetwork diameter to be the largest number of arcs that separate a boundary node from any other node in the subnetwork as shown in Fig. 5. We use this revised definition of diameter because most of the label updates on a processor result from a change to the boundary node labels. The diameter affects the number of total updates that may be performed during the solution of the shortest path algorithm; a larger diameter can result in more updates.

### 4.2. Decomposition tools

To generate the different decompositions used in this analysis, we use three different decomposition software environments along with three general decomposition strategies. The different decomposition strategies are summarized in Table 4. The first decomposition software, simulated annealing [34], is a heuristic for finding the minimum of an objective function. For this decomposition strategy, the objective function is the weighted sum of the square of the number of boundary nodes and the square of the number of arcs in each partition. This minimizes the variance of the number of arcs and boundary nodes among the subnetworks, thus resulting in subnetworks with balanced number of arcs and boundary nodes. Three decompositions were generated with this approach: SA1, SA2 and SA3. The different simulated annealing decompositions were generated using different values for the weights used in the objective function.

The second decomposition tool, Chaco [15], provides a number of different decomposition strategies. We used the multilevel Kernighan–Lin, inertial bisection, spectral bisection and linear decomposition programs (abbreviated Mult, Inert, Spect, and Lin, respectively). The multilevel Kernighan–Lin [16] creates increasingly smaller networks which approximate the original network. The smallest network is partitioned and projected back up through the larger approximated networks. Kernighan and Lin [23], a greedy graph bisection algorithm, is performed during the projections to refine the decompositions. The inertial strategy [33] finds the longest dimension of the network and cuts the network orthogonal to this dimension. The spectral bisection [30] uses eigenvectors of a matrix created from the network to perform the decomposition. The linear decomposition is a simple decomposition that assigns the arcs to the processors in the order in which they are read from the input file. For example, processor 0 gets the first $m/p$ arcs, processor 1 gets the next $m/p$ arcs, and so on.

The third decomposition tool, Metis [21,22], provides a number of options for generating the decomposition. For this analysis, we used the multilevel recursive bisection option with heavy edge matching. This method is similar to the multilevel method of Chaco. The heavy edge matching is used in the coarsening phase when the

Table 4
Decomposition descriptions

| Name | Description |
|---|---|
| Simulated annealing | Minimizes objective function |
| Chaco: multilevel-KL | Uses Kernighan–Lin on multiple levels |
| Chaco: inertial bisection | Cuts orthogonal to longest dimension |
| Chaco: spectral bisection | Uses eigenvalues for decomposition |
| Chaco: linear | Assigns arcs in order from input file |
| Metis: recursive bisection | Recursively finds minimum node cut bisection |
| Squares | Divides network into squares |
| Vertical strips | Divides network into vertical strips |
| Horizontal strips | Divides network into horizontal strips |

smaller networks are approximated to attempt to minimize the number of arcs cut in the original network. The other three decomposition strategies – squares, horizontal strips (HStrip) and vertical strips (VStrip) are easily implemented.

### 4.3. Statistical analysis of network decompositions

#### 4.3.1. Experimental results

We used the different decomposition strategies to generate decompositions for each of the grid networks. In this section, for ease of presentation, we give the results for the different decompositions for only two of the grid networks. These two networks represent trends common to all the grid networks. Tables 5 and 6 give the different characteristics for the networks decomposed for 16 processors where each

Table 5
$33 \times 33$ Grid decompositions, 16 processors

| Decomp. number | Decomp. name | Ave. # bound. nodes | Ave. # int. | Ave. # conn. comp. | Ave. diam. | Ave. # bound. nodes per int. |
|---|---|---|---|---|---|---|
| 1 | SA1 | 24.062 | 4.125 | 1.000 | 18.562 | 6.099 |
| 2 | SA2 | 23.812 | 4.125 | 1.000 | 18.000 | 6.047 |
| 3 | SA3 | 23.812 | 4.125 | 1.000 | 18.250 | 6.083 |
| **4** | **Square** | **24.250** | **4.250** | **1.000** | **16.000** | **5.953** |
| 5 | VStrip | 61.875 | 1.875 | 1.000 | 33.875 | 33.000 |
| 6 | HStrip | 61.875 | 1.875 | 1.000 | 34.000 | 33.000 |
| 7 | Multilevel | 26.688 | 4.250 | 1.000 | 20.812 | 6.517 |
| 8 | Inertial | 38.875 | 4.500 | 1.562 | 20.875 | 8.879 |
| 9 | Spectral | 26.250 | 3.875 | 1.000 | 19.812 | 7.235 |
| 10 | Linear | 40.125 | 4.625 | 2.000 | 20.875 | 9.074 |
| 11 | Metis | 26.688 | 4.125 | 1.000 | 20.000 | 6.773 |

Table 6
$129 \times 257$ Grid decompositions, 16 processors

| Decomp. number | Decomp. name | Ave. # bound. nodes | Ave. # int. | Ave. # conn. comp. | Ave. diam. | Ave. # bound. nodes per int. |
|---|---|---|---|---|---|---|
| 1 | SA1 | 228.062 | 9.000 | 2.500 | 80.688 | 25.874 |
| 2 | SA2 | 175.438 | 6.375 | 1.625 | 89.750 | 28.128 |
| 3 | SA3 | 346.812 | 12.750 | 5.125 | 79.938 | 27.162 |
| 4 | Square | 144.250 | 4.250 | 1.000 | 95.938 | 35.322 |
| 5 | VStrip | 481.875 | 1.875 | 1.000 | 257.875 | 257.000 |
| **6** | **HStrip** | **241.875** | **1.875** | **1.000** | **144.000** | **129.000** |
| 7 | Multilevel | 160.562 | 3.875 | 1.000 | 131.125 | 44.190 |
| 8 | Inertial | 259.500 | 1.875 | 1.000 | 144.062 | 138.594 |
| 9 | Spectral | 146.500 | 3.625 | 1.000 | 127.250 | 43.284 |
| 10 | Linear | 260.375 | 1.875 | 1.000 | 144.250 | 138.969 |
| 11 | Metis | 150.062 | 4.000 | 1.000 | 109.312 | 40.358 |

of the characteristics given is the average value per processor for each decomposition.

The best decomposition for each network is given in bold in the tables; the relative performance of the other decompositions are given in Figs. 6 and 7. The results given

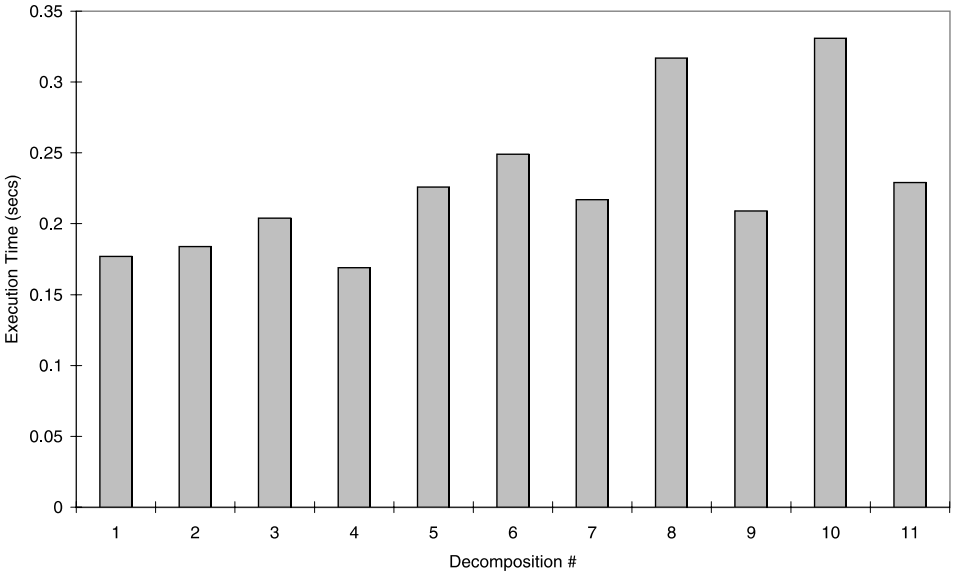


Fig. 6. Comparison of decompositions, $33 \times 33$ grid 16 processors, 32 sources.

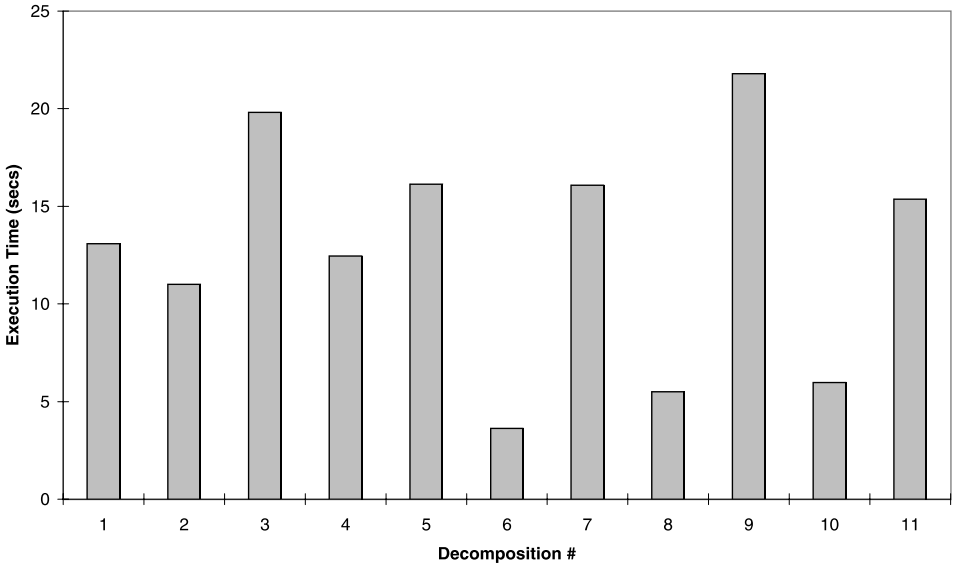


Fig. 7. Comparison of decompositions, $129 \times 257$ grid 16 processors, 32 sources.

are for the solution of 32 sources on 16 processors. To come to some conclusions about the importance of each of the characteristics on parallel execution times, we used regression analysis described below. The tables and the figures given here illustrate the best decompositions for different grid configurations (square versus rectangular) and for large versus small grids.

For the square grid networks, the square (4) and strip (5 & 6) decompositions are best, while for the rectangular grids, the strip decomposition which cuts along the longest dimension is best. These graphs show that for the $33 \times 33$ square grid network, the two strip partitions (5 & 6) have good performance. For the $129 \times 257$ rectangular grid network, the square decomposition (4) has good performance and the horizontal strips partition (6) which cuts across the longest dimension has the best performance. The vertical strip partition (5), however, has the worst performance. The inertial bisection partition (8) which cuts orthogonal to the longest dimension similar to the horizontal strip partition (6) has good performance for the $129 \times 257$ rectangular network as well.

As for the other decompositions, their relative performance is different from network to network. For example, for some of the networks, such as the $33 \times 33$ grid, the simulated annealing decompositions have better performance that the Chaco decompositions. The opposite is true for some of the Chaco decompositions for the $129 \times 257$ grid. Predicting the performance of these decompositions is difficult and no blanket statements can be made based purely on the decomposition program. Instead, we analyze the decomposition characteristics and how they affect the performance.

### 4.4. Regression analysis

We performed multiple, linear regression analyses on the characteristics of the different decompositions for each of the grid and traffic networks. The predictor variables were each of the decomposition characteristics identified above: average number of boundary nodes per processor, average number of interfaces per processor, average number of boundary nodes per interface, average number of connected components per processor and average diameter of the processor's subnetwork. The response variables were the execution times for the parallel algorithm.

In order to compare the values of the coefficients across the different decompositions, we first standardized each of the characteristics and the execution times by subtracting the mean and dividing by the standard deviation. Because of this standardization, the additive constants of the equations are always zero. Next, we divided each of the coefficients for the characteristics by the sum of the magnitudes of all the coefficients. Thus, the coefficient represents the percentage of the total magnitude effect on the response variable.

As mentioned previously, the connectivity of the decomposition greatly affects the performance. Therefore, we consider decompositions that have subnetworks that are mostly or fully connected. For this reason, we do not include this parameter in the regression analysis. In addition, the five characteristics are correlated with each

other. Therefore, performing a regression on all the characteristics at once results in coefficients that are not statistically significant as determined by *t*-tests. This means that with 90% or greater probability we cannot ensure that the corresponding predictor variable has either a negative or positive impact on the response variable. In other words, the predictor variable most likely does not have an impact on the response variable when included in that particular equation. For this reason, we consider two regressions: (1) number of boundary nodes, number of interfaces and diameter and (2) number of boundary nodes, number of boundary nodes per interface and diameter. These two regressions give the most statistically significant coefficients. In all the tables, the coefficients which are **not** statistically significant are shown in italics.

The coefficients from the first regression analysis (number of boundary nodes, number of interfaces and diameter) are given in Table 7. The positive coefficients for the number of interfaces and diameter for all the grid networks indicate that these two characteristics should be minimized. The coefficients for the boundary nodes are positive for small networks, but negative for the large networks. This result indicates that as the network size increases, it becomes more advantageous to increase the number of boundary nodes. However, no conclusion can be drawn about the number of boundary nodes from this analysis.

The second regression analysis examines both the number of boundary nodes and number of boundary nodes per interface characteristics at the same time. Table 8 gives the coefficients for the second regression: number of boundary nodes, number of boundary nodes per interface and diameter for the label-correcting two-queue algorithm for each of the grid networks. The number of boundary nodes per interface has the largest percentage of magnitude of the three characteristics. Furthermore, this coefficient is negative, which indicates that the best decomposition has large number of boundary nodes per interface. The number of boundary nodes coefficients are positive for all networks, which is contrary to the results of the first

Table 7
Regression 1 coefficients

| Grid | Num. bound. nodes | Num. int | Diam. | $r^2$ |
|---|---|---|---|---|
| $33 \times 33$ | 0.26 | 0.48 | 0.26 | 92.7 |
| $33 \times 65$ | 0.34 | 0.29 | 0.37 | 96.6 |
| $65 \times 33$ | 0.23 | 0.41 | 0.37 | 98.7 |
| $65 \times 65$ | 0.33 | 0.46 | 0.21 | 96.2 |
| $65 \times 129$ | *−0.09* | 0.34 | 0.57 | 90.7 |
| $129 \times 65$ | 0.57 | 0.37 | 0.06 | 91.8 |
| $129 \times 129$ | *−0.25* | 0.33 | *0.42* | 43.6 |
| $161 \times 161$ | −0.27 | 0.36 | 0.38 | 84.3 |
| $129 \times 257$ | −0.27 | 0.34 | 0.39 | 83.0 |
| $257 \times 129$ | *−0.22* | 0.29 | 0.49 | 75.6 |
| $193 \times 193$ | *−0.27* | *0.44* | *0.29* | 72.3 |
| $257 \times 257$ | −0.25 | 0.50 | *0.26* | 81.7 |

Table 8
Regression 2 coefficients

| Grid | Num. bound. nodes | Bound. nodes per int. | Diam. | $r^2$ |
|---|---|---|---|---|
| $33 \times 33$ | 0.40 | −0.48 | 0.12 | 96.1 |
| $33 \times 65$ | 0.27 | −0.42 | 0.31 | 99.0 |
| $65 \times 33$ | 0.35 | −0.42 | 0.23 | 97.1 |
| $65 \times 65$ | 0.43 | −0.46 | 0.11 | 96.0 |
| $65 \times 129$ | 0.35 | −0.47 | 0.18 | 96.2 |
| $129 \times 65$ | 0.38 | −0.42 | 0.20 | 96.3 |
| $129 \times 129$ | 0.26 | −0.51 | 0.22 | 56.5 |
| $161 \times 161$ | 0.20 | −0.52 | 0.28 | 90.6 |
| $129 \times 257$ | 0.13 | −0.50 | 0.36 | 87.6 |
| $257 \times 129$ | 0.20 | −0.45 | 0.35 | 80.5 |
| $193 \times 193$ | 0.15 | −0.53 | 0.31 | 81.9 |
| $257 \times 257$ | 0.18 | −0.49 | 0.33 | 86.4 |

regression analysis. The second regression analysis indicates that the number of boundary nodes per interface should be made large, but not the overall number of boundary nodes, as was suggested by the first analysis. Hence, the decomposition method should attempt to reduce the number of interfaces.

Next, the number of boundary nodes has a greater magnitude than the diameter for the small sized networks, but the opposite is true for the larger networks. This suggests that as the size of the networks grows, the diameter of the network is more important to the performance of the decomposition than the number of boundary nodes. This corresponds to the observation that for the large rectangular networks, the strip decomposition which cuts across the longest dimension has a small diameter and has the best performance. The opposite is true for the strip decomposition which cuts across the shortest dimension. In short, it is beneficial to reduce both the number of boundary nodes and diameter, but increase the number of boundary nodes per interface.

### 4.5. Summary of analyses

In summary, the best decompositions for grid networks have connected subnetworks with a small diameter, a small number of interfaces and overall boundary nodes, but a large number of boundary nodes per interface. The small number of interfaces reduces the communication cost by reducing the number of messages per iteration. The large number of boundary nodes per interface provides the processor with more information for the solution step.

The statistical analysis results are consistent with the observed performance of the different decompositions. For example, Table 6 gives the decomposition characteristics and Fig. 7 gives the decompositions' performance for the $129 \times 257$ grid network. The best decomposition, the horizontal strips (6), has a relatively average number of boundary nodes, small number of interfaces, small number of connected components, average diameter and large number of boundary nodes per interface.

Other decompositions have better values for individual characteristics, but much worse values for the others. For example, the square decomposition (4) has a small diameter but a small number of boundary nodes per interface. The vertical strips decomposition (5) has the largest number of boundary nodes per interface, but also the largest diameter. Therefore, it is important to find decompositions with ideal values for all the characteristics, not just one.

Furthermore, the results are consistent with decompositions for the urban traffic networks used in transportation applications. These networks do not have geometric information readily available; therefore, we cannot use the inertial bisection, squares or strips partitioning for these networks. As a result, the remaining decomposition schemes do not produce decompositions which are very different. For example, Table 9 gives the decomposition characteristics for the Chicago ADVANCE Test Area [4] (see Fig. 8). All of the decompositions have similar characteristics, except for the linear decomposition. Correspondingly, the execution times for the decompositions are all similar except for the linear decomposition which is significantly greater. This decomposition has a large number of boundary nodes, interfaces and connected components as well as a large diameter. Therefore, the results from the grid network analysis readily identify the linear decomposition as being poor for the traffic networks. Furthermore, the results for the decompositions are consistent for larger numbers of processors. Table 10 gives the decomposition characteristics for the ADVANCE network decomposed for 32 processors. Once again, all the decompositions except for linear have similar characteristics. Correspondingly, all decompositions have similar performance except for linear which has significantly worse performance.

## 5. Algorithm selection

As mentioned previously, we study the implementation of three parallel shortest path algorithms: a label-setting algorithm and two label-correcting algorithms. In this section, we examine the differences in performance between the algorithms, why

Table 9
ADVANCE network decompositions, 16 processors

| Decomp. number | Decomp. name | Ave. # bound. nodes | Ave. # int. | Ave. # conn. comp. | Ave. diam. | Ave. # bound. nodes per int. |
|---|---|---|---|---|---|---|
| 1 | SA1 | 72.062 | 13.875 | 1.312 | 28.062 | 5.472 |
| 2 | SA2 | 58.000 | 13.500 | 1.062 | 27.938 | 4.731 |
| 3 | SA3 | 66.250 | 15.000 | 1.375 | 27.562 | 4.417 |
| 4 | SA4 | 68.438 | 14.250 | 1.500 | 26.938 | 4.897 |
| 5 | SA5 | 58.438 | 13.625 | 1.125 | 27.562 | 4.583 |
| 6 | Multilevel | 106.000 | 11.875 | 2.188 | 32.688 | 9.704 |
| 7 | Spectral | 86.875 | 14.125 | 1.750 | 29.938 | 6.281 |
| 8 | Linear | 568.000 | 15.000 | 47.188 | 25.688 | 37.867 |
| 9 | Metis | 70.938 | 12.625 | 1.125 | 29.375 | 6.257 |

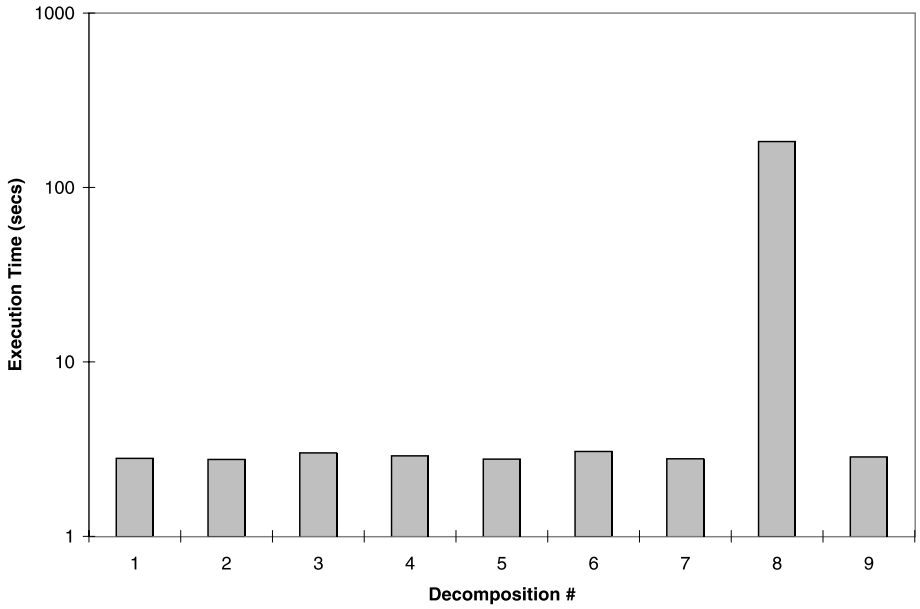


Fig. 8. Comparison of decompositions, ADVANCE network 16 processors, 32 sources.

Table 10
ADVANCE network decompositions, 32 processors

| Decomp. number | Decomp. name | Ave. # bound. nodes | Ave. # int. | Ave. # conn. comp. | Ave. diam. | Ave. # bound. nodes per int. |
|---|---|---|---|---|---|---|
| 1 | SA1 | 51.062 | 18.125 | 1.188 | 22.625 | 4.226 |
| 2 | SA2 | 48.688 | 24.000 | 1.188 | 21.812 | 2.544 |
| 3 | SA3 | 66.250 | 15.000 | 1.375 | 21.250 | 4.417 |
| 4 | SA4 | 49.312 | 20.438 | 1.375 | 23.188 | 2.774 |
| 5 | SA5 | 46.812 | 19.625 | 1.375 | 20.750 | 2.960 |
| 6 | Multilevel | 58.125 | 16.312 | 1.312 | 25.188 | 4.911 |
| 7 | Spectral | 62.312 | 18.375 | 1.000 | 22.500 | 4.674 |
| 8 | Linear | 222.000 | 25.562 | 13.312 | 23.312 | 8.685 |
| 9 | Metis | 61.625 | 18.125 | 1.125 | 25.188 | 4.510 |

these differences occur and make recommendations for the choice of the best algorithm.

For the parallel label-setting algorithm, each processor uses the serial label-setting algorithm implemented with balanced binary trees to solve for local shortest paths for its subnetwork. Therefore, this algorithm is no longer a strict label-setting algorithm; processors are simultaneously removing nodes from a queue without checking if the node label is a global minimum across all the processors. For the two

label-correcting algorithms, we implement the one-queue (Bellman–Ford–Moore) and two-queue (Pallottino) for the local shortest path solution. As mentioned previously, we use a termination detection method and network decomposition that we determined to be best for the networks we test.

For the initial analysis, we use 16 processors, but we provide results for other numbers of processors in the scalability analysis in the following section. For the analysis of the algorithms, we added counters to the code to record the number of updates, messages and iterations.

### 5.1. Execution time comparison

The execution time for each of algorithms are shown in Fig. 9 for the grid networks and in Fig. 10 for the traffic networks. Note that a log-scale is used for the $y$-axis. For the grid networks, the parallel label-correcting algorithms are 30–40% better for the smaller networks. The label-setting algorithm is 30–47% better for the larger networks. For networks larger than the $129 \times 129$ grid network (16,641 nodes), the parallel label-setting algorithm is clearly better than both parallel label-correcting algorithms. Similarly, as shown in Fig. 10, the parallel label-setting algorithm has the best performance for all of the urban traffic networks. It is around 8% better for the N/NW and ADVANCE networks, 35% better for the New York City and Waltham networks, and 66% better for the CATS network. Also, for all the traffic networks except for Waltham, the label-correcting one-queue has better performance than the label-correcting two-queue (see Table 11).

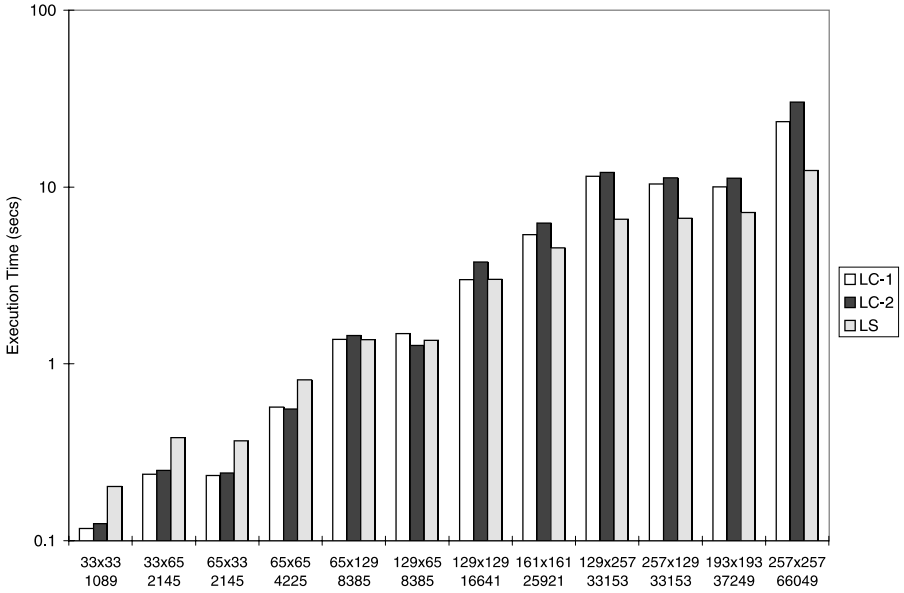


Fig. 9. Comparison of parallel algorithms, grid networks, 32 sources, 16 processors.

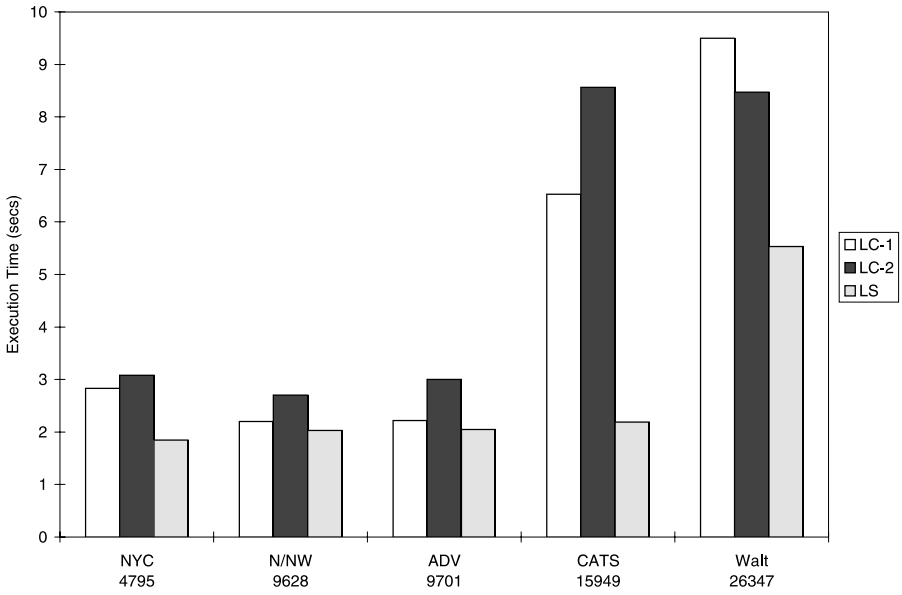


Fig. 10. Comparison of parallel algorithms, traffic networks, 32 sources, 16 processors.

Table 11
Algorithm comparison

| Grid | Execution times | | |
|---|---|---|---|
| | LC-1 | LC-2 | LS |
| $33 \times 33$ | **0.117** | 0.125 | 0.203 |
| $33 \times 65$ | **0.238** | 0.250 | 0.383 |
| $65 \times 33$ | **0.234** | 0.242 | 0.367 |
| $65 \times 65$ | 0.570 | **0.555** | 0.813 |
| $65 \times 129$ | **1.377** | 1.445 | 1.369 |
| $129 \times 65$ | 1.484 | **1.273** | 1.359 |
| $129 \times 129$ | **3.000** | 3.771 | 3.008 |
| $161 \times 161$ | 5.368 | 6.258 | **4.531** |
| $129 \times 257$ | 11.531 | 12.125 | **6.578** |
| $257 \times 129$ | 10.406 | 11.250 | **6.656** |
| $193 \times 193$ | 10.021 | 11.230 | **7.184** |
| $257 \times 257$ | 23.430 | 30.229 | **12.401** |

While the differences between the algorithms may seem small in terms of execution time, keep in mind that these results are for a small number of sources. In a transportation algorithm, the number of sources would be several hundred and the shortest path solution would be repeated for each iteration of the application. Therefore, the differences in execution time shown here would result in large differences when considered in the transportation application.

We identify three factors which affect the performance of the parallel shortest path algorithms to further explain these differences.

- **Define** *total updates*: The total number of label updates performed by the shortest path algorithm.
- **Define** *work per update*: The number of steps required by the shortest path algorithm to update a node's label. The number of steps depends on the data structure used to store the list of updated nodes. For the label-setting algorithm, it is $O(\log n/p)$, and for the two label-correcting algorithms, it is $O(1)$.
- **Define** *iteration*: One instance of the **while** loop in the parallel algorithm given in Table 4. An iteration consists of the solution of the shortest path tree for one source followed by the communication of the boundary node labels. The number of iterations that each processor performs is very difficult to predict. Intuitively, this number should be the same for all three algorithms since all the algorithms solve for the shortest path until their list is empty before proceeding to the communication step. Therefore, all the algorithms could have the same intermediate results at the end of each iteration; however, this is not the case since small differences in timing among the algorithms affect the order of communication. This results in different convergence among the algorithms.

### 5.2. Parallel performance factors

In our study, we investigated each of the three performance factors. In this paper, for brevity, we present data only from the grid networks to explain the factors; the data from the traffic networks is consistent with that of the grid networks.

#### 5.2.1. Total updates

In parallel, the number of total updates performed is affected by the order in which the nodes are updated. This order for the parallel algorithms is determined by (1) the type of algorithm used for the local solution step and (2) the order in which the boundary node labels are received. The type of algorithm determines the number of updates that will be performed for the given subnetwork and boundary node information. The order that the boundary nodes labels are received determines the order that the nodes are updated for the label-correcting algorithms since they use queues to store the list. The label-setting algorithm, on the other hand, is not sensitive to the boundary node order since the nodes are removed from the list in increasing order of the label values, not in the order in which the node labels were received.

The total number of updates for each of the grid networks is given in Table 12. The results indicate that the label-setting algorithm has the smallest number of updates, in most cases less than half that of the label-correcting algorithms. For example, for the $129 \times 65$ grid network, the label-setting algorithm performs 541,452 updates, while the label-correcting algorithms perform 1,242,797 and 1,106,410 updates for the one-queue and two-queue algorithms, respectively. The fewer updates for label-setting results in less execution time for the large networks, but not for the small networks. For example, for the $129 \times 65$ grid network, even though the

Table 12
Total updates, grid networks

| Grid | LC-1 Updates | LC-2 Updates | LS Updates |
|---|---|---|---|
| $33 \times 33$ | 102,909 | 101,927 | 65,860 |
| $33 \times 65$ | 246,821 | 238,540 | 145,158 |
| $65 \times 33$ | 238,727 | 233,112 | 137,022 |
| $65 \times 65$ | 557,850 | 514,795 | 286,734 |
| $65 \times 129$ | 1,281,879 | 1,241,823 | 547,731 |
| $129 \times 65$ | 1,242,797 | 1,106,410 | 541,452 |
| $129 \times 129$ | 2,613,761 | 2,672,568 | 1,110,917 |
| $161 \times 161$ | 4,925,999 | 4,731,674 | 1,729,889 |
| $129 \times 257$ | 8,873,576 | 8,067,494 | 2,249,286 |
| $257 \times 129$ | 8,385,158 | 7,522,878 | 2,377,962 |
| $193 \times 193$ | 8,043,723 | 8,063,876 | 2,582,184 |
| $257 \times 257$ | 18,731,866 | 18,487,611 | 4,423,215 |

number of label-setting updates is roughly half that of the label-correcting, the label-correcting two-queue algorithm has better performance. Hence, the amount of work per update affects the performance as well.

*5.2.2. Amount of work per update*

In all cases, the parallel label-setting algorithm has the least number of updates; however, this algorithm does not always have the best performance. Each update in the label-setting algorithm requires $O(\log n)$ to insert and delete from a balanced binary tree. For the label-correcting algorithm, the work per update is a constant value for the insertions and deletions from the queues. Therefore, the label-correcting algorithms can perform more updates than the label-setting algorithm and still have less execution time. The experimental results indicate that when the number of label-setting updates is less than half of the label-correcting updates, the label-setting algorithms had better performance than the label-correcting algorithms.

*5.2.3. Number of iterations*

The label-setting algorithm has good performance when implemented in parallel, yet in serial, the label-correcting algorithms outperform the label-setting. The key reason to the good performance of the parallel label-setting algorithm is it has better convergence. The convergence of the algorithm dictates the number of iterations performed. Recall that an iteration consists of (1) local shortest path solution, (2) communication of boundary nodes and (3) termination detection. Algorithms with more iterations result in more updates and more communication.

Table 13 gives the execution times, the average number of iterations and the average number of messages received per iteration per processor for the grid networks. In most cases, the label-setting algorithm has the fewest average number of iterations and the largest average number of received messages per iteration (given in bold). Fewer iterations occur because receiving more messages per iteration provides the label-setting algorithm with more node label information for the computation step.

Table 13
Number of iterations and receives, grid networks

| Grid | Execution times | | | Num. of iter. | | | Recvs. per iter. | | |
|---|---|---|---|---|---|---|---|---|---|
| | LC-1 | LC-2 | LS | LC-1 | LC-2 | LS | LC-1 | LC-2 | LS |
| $33 \times 33$ | **0.117** | 0.125 | 0.203 | 95.625 | 99.812 | **87.375** | 1.708 | **1.729** | 1.727 |
| $33 \times 65$ | **0.238** | 0.250 | 0.383 | 114.250 | 114.312 | **105.375** | 1.723 | 1.739 | **1.781** |
| $65 \times 33$ | **0.234** | 0.242 | 0.367 | 108.375 | 113.375 | **101.875** | 1.760 | 1.765 | **1.820** |
| $65 \times 65$ | 0.570 | **0.555** | 0.813 | 120.938 | 133.438 | **115.125** | 1.841 | 1.730 | **1.849** |
| $65 \times 129$ | **1.377** | 1.445 | 1.369 | 129.375 | 147.688 | **115.688** | 1.817 | 1.726 | **1.942** |
| $129 \times 65$ | 1.484 | **1.273** | 1.359 | 124.000 | 130.938 | **112.125** | 1.737 | 1.693 | **1.824** |
| $129 \times 129$ | **3.000** | 3.771 | 3.008 | 128.875 | 158.625 | **127.125** | 1.869 | 1.732 | **1.937** |
| $161 \times 161$ | 5.368 | 6.258 | **4.531** | 143.625 | 174.625 | **132.438** | 1.805 | 1.675 | **1.922** |
| $129 \times 257$ | 11.531 | 12.125 | **6.578** | 156.250 | 189.125 | **134.625** | 1.848 | 1.677 | **1.946** |
| $257 \times 129$ | 10.406 | 11.250 | **6.656** | 155.750 | 171.688 | **139.312** | 1.748 | 1.665 | **1.878** |
| $193 \times 193$ | 10.021 | 11.230 | **7.184** | 141.812 | 174.750 | **136.000** | 1.810 | 1.680 | **1.923** |
| $257 \times 257$ | 23.430 | 30.229 | **12.401** | 175.625 | 217.375 | **148.188** | 1.704 | 1.566 | **1.876** |

The label-setting algorithm's execution time has small variance across the processors since each node label is removed from the list at most once. In contrast, the variance in execution time for the label-correcting algorithm across the processors is much greater because nodes are removed from the list different numbers of times. Therefore, the work performed by the label-setting algorithm is more likely to be balanced among the processors than the label-correcting. This load balance results in the processors sending and receiving messages at the same time, thus receiving more messages during a given iteration. When the work is imbalanced, as in the case of the label-correcting algorithms, some processors reach the communication step and finish it before others; therefore, they will have to wait until subsequent iterations to receive messages from the busier processors. Having more messages results in more information for the local solve step and possibly less updates.

While the average number of received messages is only slightly higher for the label-setting, receiving just one more message per iteration, particularly in the beginning when all the node labels need to be updated, can result in much fewer iterations. The fewer number of iterations for the label-setting algorithm does not always result in the best performance, however. Even though fewer iterations are performed, the amount of time per iteration is much greater for the label-setting algorithms, especially for the small networks, resulting in larger execution time. Therefore, it is only for the large networks that the decreased number of iterations results in better performance for the label-setting algorithms.

### 5.3. *Effects of system scaling*

The results from the previous sections provide insight for the different algorithms solved on a fixed number of processors, 16. In this section, we investigate how the relative performance of the algorithms changes as the number of processors is increased from 16 up to 128. We present data for one grid network, the

$257 \times 257$ square grid; these results are consistent with the other grid and traffic networks.

The relative performance of these algorithms changes as the number of processors increases since the size of the subnetworks changes. For the $257 \times 257$ grid, the best algorithm for 16 processors is the label-setting algorithm followed by the label-correcting one-queue and then the label-correcting two-queue. Fig. 11 shows that for small numbers of processors, when the subnetwork is large, the label-setting algorithm is best. As the number of processors increases and the subnetwork sizes decrease, the three algorithms have very similar performance solving for 32 sources. This is consistent with the results given in Fig. 9 where for small subnetworks the three algorithms have comparable performance but for the large subnetworks, the label-setting algorithm has clearly better performance. The relative performance of the algorithms is the same when the number of sources is increased to 64 as shown for the $257 \times 257$ grid in Fig. 12.

The relative speedups for the data given in Figs. 11 and 12 are given in Figs. 13 and 14. This relative speedup was calculated based on the time for the smallest number of processors: 16. These graphs show that the label-correcting algorithms have better speedup than in the label-setting algorithm. This result is somewhat misleading since the reason that the speedups for the label-correcting algorithms are better than for the label-setting is that the execution time is much greater for the label-correcting algorithms at 16 processors. As shown by the execution time graphs (Figs. 11 and 12), the execution times of all three algorithms is about the same at larger number of processors, but quite different at smaller number of processors. Since the primary goal of using parallel machines is to reduce the execution time, the

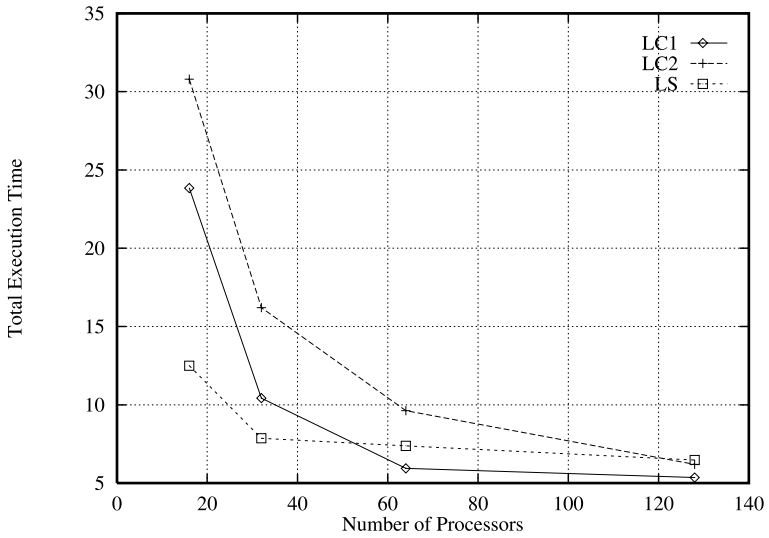


Fig. 11. $257 \times 257$ Grid scaled comparison of algorithms, 32 sources.

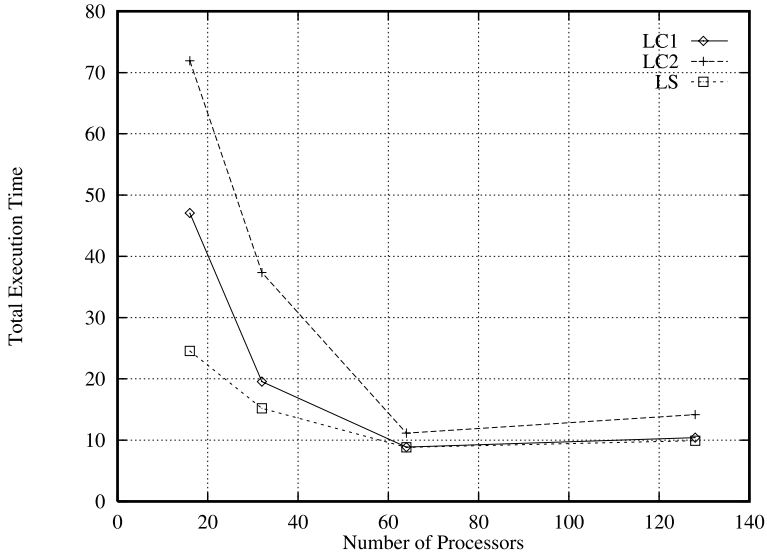


Fig. 12. 257 × 257 Grid scaled comparison of algorithms, 64 sources.

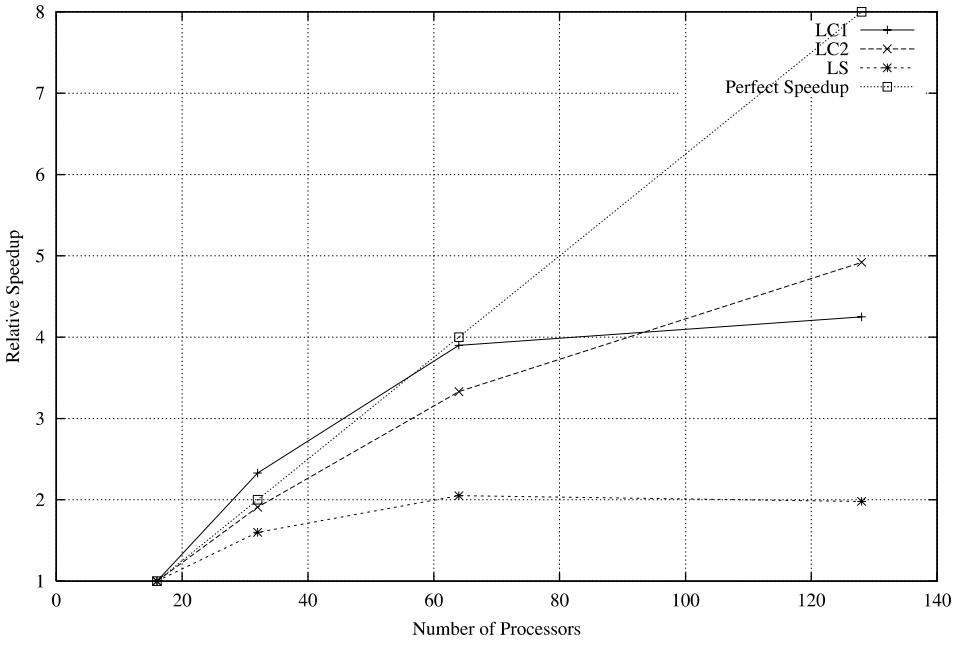


Fig. 13. 257 × 257 Grid speedup comparison of algorithms, 32 sources.

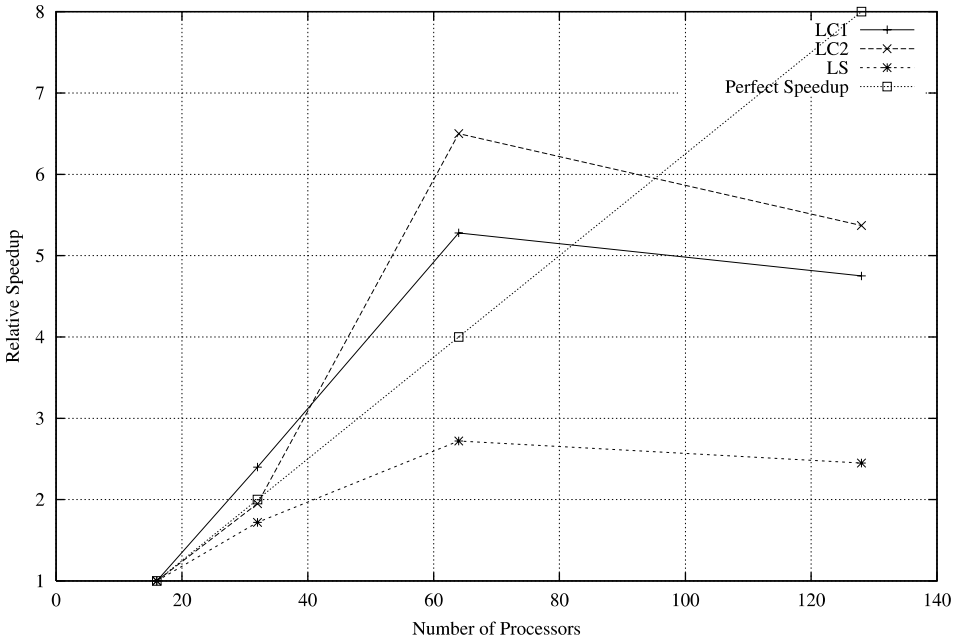


Fig. 14. $257 \times 257$ Grid speedup comparison of algorithms, 64 sources.

trends indicated by the execution time graphs should be used as a guide for algorithm selection.

## 6. Summary

Shortest path algorithms are a key component of many applications, including traffic equilibrium applications. When these applications are parallelized, an efficient parallel shortest path algorithm implementation is required in order to achieve good performance. We identify three implementation issues, each of which greatly impact the performance of the shortest path algorithm. They are termination detection implementation, network decomposition and choice of algorithm.

For the termination detection, we find a low termination detection frequency is good for small numbers of processors, but a high detection frequency is best for larger numbers of processors. This is to prevent large communication bottlenecks. In all cases, communicating often is the best since it provides the processors with the most information.

For the network decomposition, we find that the best decomposition minimizes the number of connected components, diameter and number of interfaces, but maximizes the number of boundary nodes per interface. This ideal decomposition provides the processors with a large amount of information per message, but limits communication with the small number of connected components and interfaces.

Finally, label-correcting algorithms have better serial performance than the label-setting algorithms for grid networks. In parallel, the label-setting algorithms have good performance, especially for large networks. The parallel label-setting algorithm has significantly fewer updates than the label-correcting algorithms. This is because the label-setting algorithms require fewer iterations; therefore, they incur fewer updates and less communication time. The label-setting algorithm converges faster than the label-correcting since the label-setting algorithm receives more messages per iteration on average than the label-correcting algorithms. This increased information leads to fewer updates.

## Acknowledgements

This research was part of Michelle Hribar's Ph.D. dissertation completed at Northwestern University. She was supported by a National Science Foundation Graduate Fellowship and the Department of Energy. Valerie Taylor is supported by NSF Young Investigator award under Grant CCR-9215482. David Boyce is supported by the National Science Foundation through Grant DMS-9313113 to the National Institute of Statistical Sciences.

The authors would like to acknowledge Stanislaw Berka for his help with the traffic networks and Jane Hagstrom for her suggestions and encouragement. We thank Robert Dial for his comments on the basic approach to the parallel shortest path algorithm. Finally, we acknowledge Bruce Holmer for the use of his simulated annealing software.

This research was performed in part using the CACR parallel computer system operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by NSF.

## References

[1] P. Adamson, E. Tick, Greedy partitioned algorithms for the shortest path problem, International Journal of Parallel Programming 20 (1991) 271–298.

[2] R. Ahuja, T. Magnanti, J. Orlin, Network Flows: Theory, Algorithms and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1993.

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

[4] S. Berka, D.E. Boyce, J. Raj, B. Ran, A. Tarko, Y. Zhang, A large-scale route choice model with realistic link delay functions for generating highway travel times, Technical Report to Illinois Department of Transportation, Urban Transportation Center, University of Illinois, Chicago, IL, 1994.

[5] D. Bertsekas, F. Guerriero, R. Musmanno, Parallel asynchronous label-correcting methods for shortest paths, Journal of Optimization Theory and Applications 88 (1996) 297–320.

[6] D.E. Boyce, D.H. Lee, B.N. Janson, S. Berka, Dynamic user-optimal route choice model of a large-scale traffic network, Presented at the 14th Pacific Regional Science Conference, Taipei, Taiwan, ROC, 1995.

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.
[8] E.W. Dijkstra, A note on two problems in connection with graphs, Numerical Mathematics 1 (1959) 269–271.
[9] D. Englund, Personal Correspondence, 1996.
[10] L.R. Ford Jr., D.R. Fulkerson, Flows in Networks, Princeton University Press, Princeton, NJ, 1962.
[11] G. Gallo, S. Pallottino, Shortest path methods in transportation methods, in: M. Florian (Ed.), Transportation Planning Models, Elsevier, Amsterdam, 1984.
[12] G. Gallo, S. Pallottino, Shortest path methods: a unifying approach, Mathematical Programming Study 26 (1986) 38–64.
[13] B. Golden, Shortest-path algorithms: a comparison, Operations Research 24 (6) (1976) 1164–1168.
[14] M. Habbal, H. Koutsopoulos, S. Lerman, A decomposition algorithm for the all-pairs shortest path problem on massively parallel computer architectures, Transportation Science, December 1994.
[15] B. Hendrickson, R. Leland, The Chaco user's guide, Sandia National Laboratories, Albuquerque, 1993.
[16] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, Technical Report SAND92-1460, Sandia National Laboratories, 1993.
[17] J.E. Hicks, D.E. Boyce, A. Sen, Static network equilibrium models and analyses for the design of dynamic route guidance systems, Final Report to Illinois Department of Transportation, Urban Transportation Center, University of Illinois, Chicago, IL, 1992.
[18] M. Hribar, Parallel traffic equilibrium applications, Ph.D. Thesis, Northwestern University, Evanston, IL, 1997.
[19] M. Hribar, V. Taylor, D. Boyce, Choosing a shortest path algorithm, Technical Report CSE-95-004, Computer Science and Engineering, EECS Department, Northwestern University, 1995.
[20] M. Hribar, V. Taylor, D. Boyce, Termination detection for parallel shortest path algorithms, Journal of Parallel and Distributed Computing 55 (1998) 153–165.
[21] G. Karypis, V. Kumar, Metis: unstructured graph partitioning and sparse matrix ordering system, Department of Computer Science, University of Minnesota, August 26, 1995.
[22] G. Karypis, V. Kumar, Multilevel $k$-way partitioning scheme for irregular graphs, Department of Computer Science, University of Minnesota, August 28, 1995.
[23] B. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, Bell System Technical Journal 29 (1970) 291–307.
[24] V. Kumar, V. Singh, Scalability of parallel algorithms for the all-pairs shortest-path problem, Journal of Parallel and Distributed Computing 13 (1991) 124–138.
[25] E.F. Moore, The shortest path through a maze, in: Proceedings of the International Symposium on Theory of Switching, Harvard University Press, Cambridge, MA, 1957, pp. 285–292.
[26] S. Pallottino, Adaptation de l'algorithme de d'esopo-pape pour la determination de tous les chemins les plus courts: ameliorations et simplifications, Technical Report 136, Centre de Recherche sur les Transports, Universite de Montreal, 1979.
[27] S. Pallottino, Shortest-path methods: complexity, interrelations and new propositions, Networks 14 (1984) 257–267.
[28] M. Papaefthymiou, J. Rodrigue, Implementing parallel shortest-paths algorithms, 1994.
[29] U. Pape, Implementation and efficiency of Moore-algorithms for the shortest path problem, Mathematical Programming 7 (1974) 212–222.
[30] A. Pothen, H. Simon, K. Liou, Partitioning sparse matrices with eigenvectors of graphs, SIAM Journal of Matrix Analysis and Algorithms 11 (3) (1990) 430–452.
[31] H.E. Romeijn, R.L. Smith, Parallel algorithms for solving aggregated shortest path problems, The University of Michigan, September 1993.
[32] Y. Sheffi, Urban Transportation Networks: Equilibrium Analysis with Mathematical Programming Methods, Prentice-Hall, Englewood Cliffs, NJ, 1985.
[33] H.D. Simon, Partitioning of unstructured problems for parallel processing, in: Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications, 1991.

[34] V.E. Taylor, B.K. Holmer, E.J. Schwabe, M.R. Hribar, Balancing load versus decreasing communication: exploring the tradeoffs, in: Hawaii International Conference on System Sciences, 1996.
[35] J.L. Traff, An experimental comparison of two distributed single-source shortest path algorithms, Parallel Computing 21 (1995) 1505–1532.