Parallel Asynchronous Label Correcting Methods for Shortest Paths [1,2]

D. P. BERTSEKAS[3], F. GUERRIERO[4] AND R. MUSMANNO[5]

Communicated by G.P. Papavassilopoulos

---

[1] The authors acknowledge the director and the staff of CERFACS, Toulouse, France, for the use of the Alliant FX/80.

[2] This research was supported by National Science Foundation under Grants 9108058-CCR, 9221293-INT, and 9300494-DMI.

[3] Professor, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, U.S.A.

[4] Research Fellow, Dipartimento di Elettronica, Informatica e Sistemistica, Università della Calabria, Rende, Italy.

[5] Assistant Professor, Dipartimento di Elettronica, Informatica e Sistemistica, Università della Calabria, Rende, Italy.

Abstract. We develop parallel asynchronous implementations of some known and some new label correcting methods for finding a shortest path from a single origin to all the other nodes of a directed graph. We compare these implementations on a shared memory multiprocessor, the Alliant FX/80, using several types of randomly generated problems. Excellent (sometimes superlinear) speedup is achieved with some of the methods, and it is found that the asynchronous versions of these methods are substantially faster than their synchronous counterparts.

Key Words. Shortest path problem, parallel asynchronous algorithms, shared memory multiprocessor, label correcting method.

1. Introduction

In this paper we consider the problem of finding a path of minimum length from an origin node to each of the other nodes in a directed graph (N,A), where N is the set of nodes and A is the set of arcs. The numbers of nodes and arcs are denoted by n and m respectively. For each arc $(i,j) \in A$ we are given a scalar length a_{ij}. For convenience, we assume that there is at most one arc from a node i to a node j, so that we can unambiguously refer to arc (i,j). A path starting from a node i_1 and ending at a node i_k consists of a sequence of forward arcs of the form (i_1,i_2), (i_2,i_3), ..., (i_{k-1},i_k). The length of such a path is defined to be the sum of the lengths of its arcs:

$$\sum_{j=1}^{k-1} a_{i_j i_{j+1}} .$$

For each node j, we want to find to find a path of minimum length that starts at node 1 and ends at j. *Throughout this paper we assume that all arc lengths are nonnegative and that there exists at least one path from node 1 to each other node.*

The shortest path problem is very common in practice, either by itself or as a subroutine in algorithms for more complex problems. Its fast solution is thus of great practical interest. In this paper, we focus attention on the class of label correcting methods. A recent computational study by Gallo and Pallottino (Ref. 1) has shown that for single origin-all destinations shortest path problems, the most efficient label correcting methods are faster than the most efficient label setting methods in a serial computational environment, particularly for sparse problems, that is, for problems involving graphs with a relatively small number of arcs. This conclusion agrees with our own experience. The results of this paper strongly suggest that the advantage of label correcting methods for sparse all-destinations problems carries over to a shared memory parallel computation setting.

The methods of this paper can be adapted to solve single origin-few destinations problems. For such problems, however, label correcting methods have been outperformed by label setting (Dijkstra) methods and also by auction algorithms, as reported in Refs. 2-4. Parallel implementations of these methods for single-origin single-destination problems have been given in Refs. 4-5, and it is quite likely that for many problems of this type, the two-sided Dijkstra and the two-sided auction methods of Refs. 4-5, respectively, outperform the methods of the present paper in both a serial and a parallel computing environment.

The prototype label correcting algorithm, as given by Gallo and Pallottino (Ref. 6), maintains a vector $(d_1, d_2, \cdots, d_n)$ of labels and a candidate list V of nodes. Initially, we have

$$d_1 = 0, \qquad d_i = \infty \text{ for } i \neq 1,$$

and

$$V = \{1\}.$$

The algorithm terminates when V is empty, and upon termination, each label d_i is the shortest distance to node i. Assuming V is nonempty, a typical iteration is as follows:

Remove from V a node i that is in V.

For each arc $(i,j) \in A$, if $d_j > d_i + a_{ij}$, set

$$d_j := d_i + a_{ij}$$

and add j to V if j does not already belong to V.

There are several different methods for selecting at each iteration the node to be removed from the candidate list V. If the node exiting V is the node with the minimum label, Dijkstra's method is obtained. In this case, each node will enter and exit V exactly once. Label correcting methods avoid the overhead associated with finding the minimum label node at the expense of multiple entrances of nodes into V.

The simplest label correcting method, the Bellman-Ford method (Ref. 7), maintains V in a FIFO queue; nodes are removed from the top of the queue and are inserted at the bottom. More sophisticated label correcting methods maintain V in one or in two queues and use more complex removal and insertion strategies. The objective is to reduce the number of node reentries in V. These methods are significantly faster than the Bellman-Ford method, and will be discussed in the next two sections with an emphasis on a general principle enunciated in Ref. 8 for the case where the arc lengths are nonnegative. According to this principle, the number of node reentries is reduced if nodes with relatively small label are removed from V. Dijkstra's method, the threshold algorithm of Ref. 9, and the SLF (Small Label First) method of Ref. 8 conform to this principle. A new method, the LLL (Large Label Last) method, which also conforms to this principle, will be presented in the next section. Other methods can be obtained by combinations of threshold, SLF, LLL, and also the D'Esopo-Pape method of Ref. 10. For a recent textbook discussion and analysis of other shortest path methods, we refer the reader to Ref. 11.

Label correcting methods can be parallelized in straightforward fashion. Furthermore, they admit an asynchronous implementation, as first shown in Ref. 12 in the broader context of dynamic programming. In such an implementation, multiple nodes of the candidate list can be asynchronously and independently chosen for iteration by different processors, and the associated calculations may be done at the various processors with label information that is potentially out-of-date because of intermediate label updating operations by other processors; see also Ref. 13, p. 451. An extensive reference on parallel asynchronous algorithms, including shortest path methods, is Ref. 14, particularly Ch. 6. There is considerable computational evidence at present that asynchronous algorithms, when valid, can be substantially faster than their synchronous counterparts, primarily because they avoid the penalty associated with synchronizing the iterations at different processors. We note that with the exception of the auction algorithms of Ref. 5, all earlier implementations of parallel shortest path algorithms that we are aware of Refs. 4, 15-16, are synchronous.

A major aim of this paper is to develop parallel synchronous and asynchronous implementations of a variety of label correcting methods, and to evaluate their speedup over their serial versions in a shared memory machine, the Alliant FX/80 with 8 processors. This is done in Sections 3 and 4. Our major findings are that (a) with proper implementation, excellent (close to linear) speedup can be obtained with some but not all label correcting methods, (b) asynchronous implementations are considerable faster than their synchronous counterparts, and (c) the threshold method, which in combination with the SLF and the LLL methods is the fastest serial method in our experiments, does not lend itself to substantial speedup. As a result the pure SLF and SLF-LLL methods are the fastest in a parallel setting.

2. Label Correcting Methods Based On the Small Label Principle

In this section we describe three methods motivated by a general principle given in Ref. 8 regarding the node selection policy of a label correcting method. According to this principle, for problems with nonnegative arc lengths, the number of iterations of the method is strongly correlated with the average rank of the node removed from V, where nodes are ranked in terms of the size of their label (nodes with small labels have small rank). Thus one should make an effort to select nodes with relatively small label. This was verified by extensive testing reported in Ref. 8 with two methods based on this principle, the threshold and SLF methods, and their combinations. We describe these two methods and we then propose a third new method, that can also be combined with the first two.

In the threshold algorithm of Ref. 9, the candidate list V is partitioned in two disjoint queues Q_1 and Q_2, on the basis of a threshold parameter s. At each iteration, the node removed from V is the top node of Q_1, while a node entering V is added at the bottom of Q_2 or Q_1 depending on whether its label is greater than s, or smaller or equal to s, respectively. In this way, the queue Q_1 contains only nodes whose labels are not larger than s. When Q_1 is exhausted, the entire list V is repartitioned in two queues according to an appropriately adjusted threshold parameter.

To understand the main idea of the threshold algorithm, suppose that at time t, the threshold is set to a new value s, and at some subsequent time t'>t the queue Q_1 gets exhausted. Then at time t' all the nodes of the candidate list have label greater than s. In view of the nonnegativity of the arc lengths, this implies that all nodes with label less or equal to s will not reenter the candidate list after time t'. In particular, all nodes that exited the candidate list between times t and t' become permanently labeled at time t' and never reenter the candidate list. We may thus interpret the threshold algorithm as a block version of Dijkstra's method, whereby a whole subset of nodes becomes permanently labeled when the queue Q_1 gets exhausted.

However, when one tries to parallelize the threshold algorithm, it is difficult to maintain the permanent labeling property described above. The reason is that this property depends on using a uniform threshold value for the entire candidate list. In particular, this property will not hold if the candidate list is divided into multiple (partial) candidate lists, each operated by a separate processor with its own independent threshold value. The alternative to maintaining multiple parallel lists with independent threshold values is either to maintain a single list, which is accessed by all processors, or to maintain a common threshold value across the independent lists of the different processors. Both of these alternatives requires considerable synchronization between processors, and this is the reason why we were unable to parallelize the threshold method as efficiently as other methods.

We also note that the performance of the threshold method is very sensitive to the procedure used for adjusting the threshold parameter s. In particular, if s is chosen too small, the method becomes equivalent to an unsophisticated version of Dijkstra's algorithm, while if s is chosen too large, the method is quite similar to the Bellman-Ford algorithm. The original proposal of the threshold algorithm (Ref. 9) gives a heuristic method for choosing the threshold that works remarkably well for many problems, as also verified in Refs. 1, 8. However, it appears that choosing appropriate threshold values becomes more complicated in a parallel setting.

In the Small Label First algorithm (SLF) the candidate list V is maintained as a double ended queue Q. At each iteration, the node removed is the top node of Q. The rule for inserting new nodes is given below:

Let i be the top node of Q, and let j be a node that enters Q.

If $d_j \leq d_i$, then enter j at the top of Q ;

else, enter j at the bottom of Q.

The SLF method provides a rule for inserting nodes in the queue, but always removes nodes from the top of Q. We now propose a more sophisticated node removal strategy, which aims to remove from Q nodes with small labels. In particular, we suggest that, at each iteration, when the node at the top of Q has a larger label than the average node label in Q (defined as the sum of the labels of the nodes in Q divided by the cardinality |Q| of Q), this node is not removed from Q, but rather it is repositioned to the bottom of Q. We refer to this as the Large Label Last strategy (LLL for short). The LLL strategy for removing nodes from V can be summerized as follows:

Let i be the top node of Q, and let s = $\sum_{j \in Q} d_j / |Q|$.

If $d_i > s$, then move i at the bottom of Q .

Repeat until a node i such that $d_i \leq s$ is found and is removed from Q .

It is simple to combine the SLF queue insertion and the LLL node selection strategies, thereby obtaining a method referred to as SLF-LLL. We have found that the combined SLF-LLL method consistently requires a smaller number of iterations than either SLF or LLL, although the gain in number of iterations is sometimes more than offset by the extra overhead per iteration.

The SLF and LLL strategies can also be combined with the threshold algorithm. In particular, the LLL strategy is used when selecting a node to exit the queue Q_1 (the top node of Q_1 is repositioned to the bottom of Q_1 if its label is found smaller than the average label in Q_1). Furthermore, whenever a node enters the queue Q_1, it is added to the bottom or the top of Q_1 depending on whether its label is greater than the label of the top node of Q_1 or not. The same policy is used when transferring to Q_1 the nodes of Q_2 whose labels do not exceed the current threshold parameter. Thus the nodes of Q_2 are transferred to Q_1 one-by-one, and they are added to the top or the bottom of Q_1 according to the SLF strategy. Finally, the SLF strategy is also followed when a node enters the queue Q_2.

It is also possible to combine the SLF and LLL strategies with the D'Esopo-Pape method (Ref. 10), as has already been proposed (for the case of the SLF strategy) in Ref. 17. In the D'Esopo-Pape method the candidate list V is maintained as a double ended queue Q. At each iteration, the node removed is the top node of Q, but a new node is inserted at the bottom of Q if it has never entered Q before, and is inserted at the top of Q otherwise. The rationale for this queue insertion strategy is somewhat unclear, but the literature contains numerous reports of excellent performance of the D'Esopo-Pape method. However, the results of Ref. 8 show that the D’Esopo-Pape method is not consistently faster than the Bellman-Ford algorithm and indeed in some cases it is dramatically slower. Following the suggestion of Ref. 17, we have also experimented with serial implementations of various combinations of the SFL and the SLF-LLL strategies with the D’Esopo-Pape method. We have verified that the use of the SLF strategy for nodes that enter Q for the first time reduces the number of iterations and that the use of the LLL strategy, in addition to SLF, reduces the number of iterations even further. However, we found that in a serial environment, the combinations of SLF and LLL with the threshold algorithm are much faster than the corresponding combinations with the D'Esopo-Pape method. We have not experimented with combinations of the D'Esopo-Pape method with SLF and LLL in a parallel setting. We note, however, that parallel asynchronous implementations of such combinations based on the ideas of this paper are straightforward. It is plausible that these implementations will prove effective for problems where the D'Esopo-Pape method is much faster than the Bellman-Ford algorithm.

The results of Ref. 8 and Ref. 17, and the results of the present paper demonstrate that for problems with nonnegative arc lengths the SLF and LLL strategies consistently improve the performance of the Bellman-Ford, the threshold, and the D'Esopo-Pape method. It is seen therefore that SLF and LLL are complementary to the other basic label correcting methods and improve their performance when combined with them. We will see in the next two sections that the same is true in a parallel setting.

Regarding the theoretical worst-case performance of the SLF and the combined SLF-LLL algorithms, it is not known at present whether these algorithms have polynomial complexity. However, extensive computational experience has yielded no indication of nonpolynomial behavior. In any case, it is possible to construct provably polynomial versions of these algorithms as follows.

Suppose that there is a set of increasing iteration indices $t_1, t_2, \ldots, t_{n+1}$ such that $t_1=1$, and for $i=1,\ldots,n$, all nodes that are in V at the start of iteration t_i are removed from V at least once prior to iteration t_{i+1}. Because all arc lengths are nonnegative, this guarantees that the minimum label node of V at the start of iteration t_i will never reenter V after iteration t_{i+1}. Thus the candidate list must have no more than n-i nodes at the start of iteration t_{i+1}, and must become empty prior to iteration t_{n+1}. Thus, if the running time of the algorithm between iterations t_i and t_{i+1} is bounded by

R, the total running time of the algorithm will be bounded by nR, and if R is polynomially bounded, the running time of the algorithm will also be polynomially bounded.

Assume now, in particular, that between iterations t_i and t_{i+1}, each node is inserted at the top of Q for a number of times that is bounded by a constant and that (in the case of SLF-LLL) the total number of repositionings is bounded by a constant multiple of m. Then it can be seen that the running time of the algorithm between iterations t_i and t_{i+1} is O(m), and therefore the complexity of the algorithm is O(nm). To modify SLF or SLF-LLL so that this result applies, it is sufficient that we fix an integer k>1, and that we separate the iterations of the algorithm in successive blocks of kn iterations each. We then impose an additional restriction that, within each block of kn iterations, each node can be inserted at most k-1 times at the top of Q (that is, after the (k-1)th insertion of a node to the top of Q within a given block of kn iterations, all subsequent insertions of that node within that block of kn iterations must be at the bottom of Q). In the case of SLF-LLL, we also impose the additional restriction that the total number of repositionings within each block of kn iterations should be at most km (that is, once the maximum number of km repositionings is reached, the top node of Q is removed from Q regardless of the value of its label). The worst-case running time of the modified algorithms are then O(nm). In practice, it should be highly unlikely that the restrictions introduced into the algorithms to guarantee O(nm) complexity will be exercised if k is larger than say 3 or 4.

3. Parallel Label Correcting Methods

The general principle for parallelizing the generic label correcting method is straightforward. The basic idea is that several nodes can be simultaneously removed from the candidate list and the labels of the adjacent nodes can be updated in parallel. In a shared memory machine, the label of a node is maintained in a unique memory location, which can be accessed by all processors. During the concurrent label updating it is possible that multiple processors will attempt to modify simultaneously the label of the same node. For this reason, the label updating operation must be executed with the use of a lock, which guarantees that only one processor at a time can modify a given label.

Two important characteristics of a parallel shared memory implementation of a label correcting method are whether:

(a) The candidate list is organized in a single queue shared by all processors, or in multiple queues, that is, a separate queue for each processor.

(b) The label updating is synchronous or asynchronous.

The issue of one versus multiple queues primarily deals with the tradeoff between good load balancing among multiple processor queues and increased contention for access to a single queue. We will see, however, that multiple queues also enhance the effectiveness of the SLF and LLL strategies because with multiple queues, more nodes with small labels tend to rise to the top of the queues.

Our implementation of the various queue strategies is as follows:

Parallel One-Queue Algorithm. We have a single queue Q shared among all processors (in the case of the threshold algorithms this queue is partitioned as discussed earlier). Each processor removes the node at the top of Q, updates the labels of its adjacent nodes, and adds these nodes (if necessary) into Q, according to the insertion strategy used. The procedure is repeated until Q is found empty. In the latter case the processor switches to an idle state and reawakens when Q becomes nonempty. The execution is stopped when the idle condition is reached by all processors. This algorithm suffers for substantial contention between the processors to access the top node of Q and also to insert nodes into Q.

Parallel Multiple-Queues Algorithm. In this algorithm, each processor uses a separate queue. It extracts nodes from the top of its queue, updates the labels for adjacent nodes, and uses a heuristic procedure for choosing the queue to insert a node that enters V. In particular, the queue chosen is the one with minimum current value for the sum of the out-degrees of the nodes assigned to the queue (the out-degree of a node i is the number of outgoing arcs from i). This heuristic is easy to implement and ensures good load balancing among the processors. In our implementations, a node can reside in at most one queue. In particular, a processor can check whether a node is present in the candidate list (that is, in some queue) by checking the value of a boolean variable, which is updated each time a node enters or exits the candidate list. In the case of the threshold algorithms, the threshold setting policy of the corresponding serial method was used independently for each of the queues.

For all algorithms tested, we have found that the multiple-queues versions were more efficient than their single queue counterparts. The reason is that in the case of multiple queues, there is much less contention for queue access than in the case of a single queue, because with multiple queues, the likelihood of multiple processors attempting simultaneously to insert a node in the same queue is much smaller. For this reason, we concentrate in what follows in the multiple-queues implementation.

The issue of synchronous versus asynchronous implementation is an issue of tradeoff between orderliness of computation and penalty for synchronization. In a synchronous implementation, the computation proceeds in rounds of parallel iterations. During each round, each processor removes a different node from the candidate list (if the number of processors is greater than the number of nodes, some processors remain idle). The processors then update in parallel the labels of the corresponding adjacent nodes. Finally, a new round begins once all the label updating from the current round is finished.

In an asynchronous algorithm, there is no notion of rounds, and a new node may be removed from the candidate list by some processor while other processors are still updating the labels of various nodes. A single origin-single destination label correcting method resembling the ones considered here is given in p. 451 of Ref. 14. More formally, for t = 0, 1, …, let $d_j(t)$ denote the value of the label of node j at time t; this is the value of d_j that is kept in shared memory. In our mathematical model of the asynchronous label correcting algorithm, the label $d_j(t)$ is updated at a subset of times $T^j \subset \{0, 1, \ldots\}$ by some processor that need not be specified further.

The updating formula is:

$$d_j(t+1) = \begin{cases} d_i(\tau_i^j(t)) + a_{ij} & \text{, if } \quad d_i(\tau_i^j(t)) + a_{ij} < d_j(t) \quad \text{and} \quad t \in T^j \\ d_j(t) & \text{, otherwise} \quad . \end{cases} \tag{1}$$

Here $\tau_i^j(t)$ is the time at which the label d_i was read from shared memory by the processor updating d_j at time t. The asynchronism results from the possibility that we may have $\tau_i^j(t) < t$ and $d_i(\tau_i^j(t)) \neq d_i(t)$ because the label d_i stored in shared memory may have been changed between the times $\tau_i^j(t)$ and t by another processor. Note, however, that before the label of d_j can be changed according to (1), the value $d_i(\tau_i^j(t)) + a_{ij}$ must be found smaller than the current value $d_j(t)$. One way to accomplish this is to lock the memory location storing d_j after d_j is read, to ensure that no other processor can change d_j while the test

$$d_i(\tau_i^j(t)) + a_{ij} < d_j(t) \tag{2}$$

is conducted. The drawback of this method is that the memory location of d_j may be locked unnecessarily, while other processors are waiting to read the value of d_j.

An alternative method that we found much more efficient is to first read $d_j(t')$ at some time t' and (without locking its value) compare it to $d_i(\tau_i^j(t')) + a_{ij}$. If d_j is found smaller, its memory location is locked and its current value $d_j(t)$ (which may have been changed by another processor as the test (2) was being conducted) is read again. Depending on whether the test (2) is passed, the new value $d_j(t+1)$ is recorded according to (1) and the corresponding memory location is unlocked. This memory management method reduced significantly the number of locking operations and contributed substantially in the speedup of the algorithms.

The convergence of the preceding algorithm to the correct shortest distances d_i^*, that is,

$$d_i(t) = d_i^*, \qquad \forall\, t \geq \bar{t}, \quad i = 1, 2, \ldots, n, \tag{3}$$

where $\bar{t}$ is some (finite) time, can be shown under very weak assumptions. In particular, what is needed is that T^j is an infinite set for each $j \neq 1$, that if (i,j) is an arc, the node i is used in (1) for an infinite subset of T^j, and that $\tau_i^j(t) \to \infty$ as $t \to \infty$. These are the minimal conditions for asynchronous convergence, as discussed in Ref. 14, Ch. 6. Note that the computation can be terminated once a time $\bar{t}$ such that (3) holds is found. In our shared memory context, the time $\bar{t}$ where termination occurs as in (3) is recognized as the time where the queue Q is empty and all processors are idle. The proof of convergence closely resembles related proofs in Ref. 14, Section 6.4, in Ref. 11, and inRef. 13, Section 5.2.4, and will not be given here.

It has often been found empirically that asynchronous algorithms, when valid, outperform their synchronous counterparts because they are not delayed by synchronization requirements. Examples are given in Refs. 6, 18, which give parallel asynchronous implementations of auction algorithms that bear similarity with the implementations given here. However, to our knowledge, the present paper is the first to address the implementation of asynchronous label correcting methods and to assess their performance.

The synchronous algorithms also use multiple queues, since we found the single queue versions to be relatively inefficient. The insertion of nodes in the queues is done similar to the corresponding asynchronous algorithms. Our implementation is depicted in Fig. 1, and involves two synchronization points, the first at the end of the label updating procedure and the second at the conclusion of the iteration. Each processor temporarily stores the values of the updated labels in a private memory area; in this way, the new labels of nodes can be computed by a processor without locking their shared memory locations, which would delay the reading of these labels by other processors. Thus, at the end of the label updating task, the same node could be stored into multiple private memory locations with different label values. Following the label updating task, the updated labels are transferred to their main (shared) memory locations, and the corresponding nodes are added to V, if they are not already present in V. We have also tried the alternative scheme where the node labels are directly updated at their shared memory locations, but this approach turned out to be less efficient. In our implementation of the asynchronous algorithms, a processor upon completing an iteration, does not wait for the completion of the iteration of the other processors at any time but starts instead a new iteration (if V is not empty), thereby avoiding the corresponding synchronization penalty.

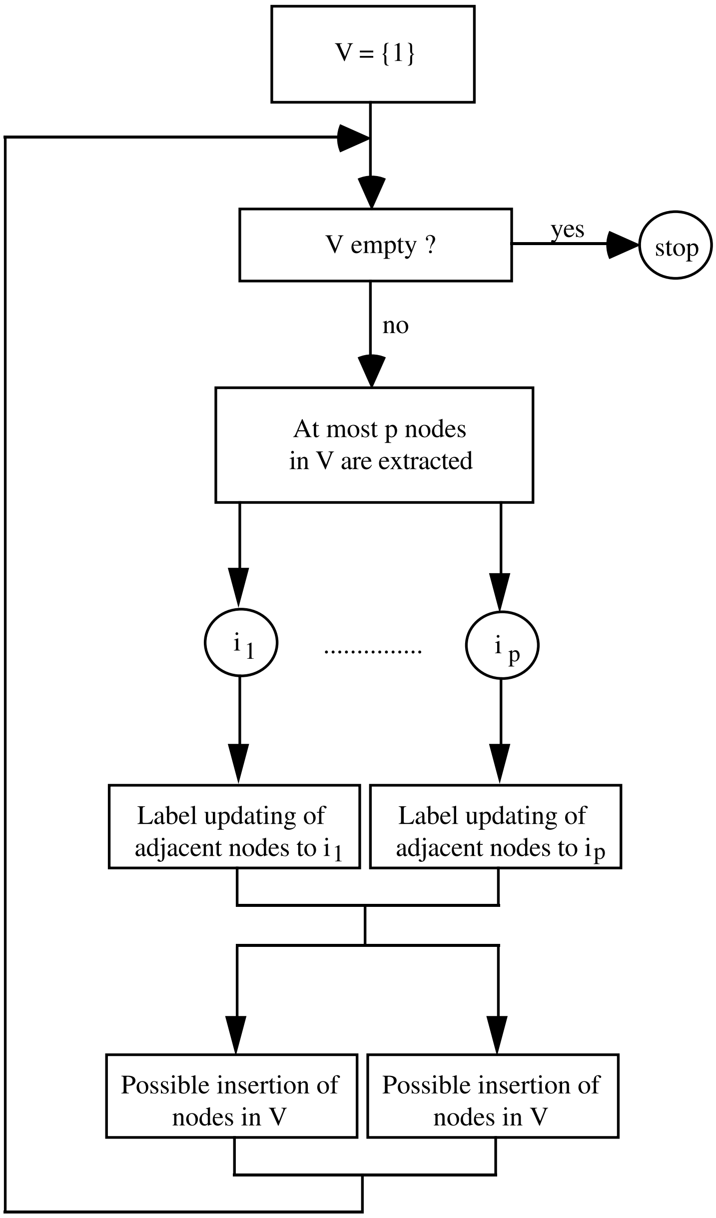

Fig. 1. Parallel synchronous label correcting algorithm

4. Numerical Experiments

The SLF and SLF-LLL algorithms were implemented and tested using an Alliant FX/80. This computer is based on a vector-parallel architecture with 8 processors, each with 23 Mflops of peak performance, sharing a common memory of 32 MBytes. The compiler used was FX/Fortran 4.2. The vectorization capability of the processors was not used in our experiments.

In order to evaluate numerically the efficiency of the methods, we have tested the following six codes, which evolved from the codes of Refs. 1,8:

- B-F: Bellman-Ford method;
- SLF: Small Label First method;
- SLF-LLL: Small Label First method, using in addition the Large Label Last strategy for node removal;
- THRESH: Threshold method; the method for setting the threshold parameter is the same as the one that was recommended in Ref. 9 and was also used in Ref. 8;
- SLF-THRESH: Threshold method in combination with the SLF method for the node insertion strategy;
- SLF-LLL-THRESH: The preceding method, using in addition the Large Label Last strategy for node removal;

We used four different types of randomly generated test problems for which all arc lengths were chosen according to a uniform distribution from the range [1,1000].

Grid/Random Problems (G1, G2, G3, G4). These are problems generated by a modified version of the GRIDGEN generator of Ref. 11. The number of arcs is 1,000,000 for all problems, and the nodes are arranged in a square planar grid with the origin node 1 set in the southwest corner. Each pair of adjacent grid nodes is connected in both directions. We can also have additional arcs with random starting and ending nodes. The number of nodes was selected so that the total number of additional arcs is approximately 2, 3, 4, and 5 times the number of grid arcs.

Euclidean Grid/Random Problems (E1, E2, E3, E4). These problems are generated similar to the preceding class. The only difference is that the length of each nongrid arc from the grid node (i,j) to the grid node (h,k) is set to $r \cdot e_{ij,hk}$, where $e_{ij,hk}$ is the Euclidean distance of the nodes (the square root of $(i-h)^2+(j-k)^2$), and r is an integer chosen according to a uniform distribution from the range [1,1000].

Netgen Problems (N1, N2, N3, N4). These are problems generated with the public domain program NETGEN (Ref. 19). The number of arcs is 1,000,000, whereas the number of nodes was chosen as 31,622, 15,811, 11,952, and 10,000.

Fully Dense Problems (C1, C2, C3, C4). In these problems all the possible n(n-1) arcs are present.

Road Networks (R1, R2, R3, R4). These are the Manhattan, Waltham, Boston, and Middlesex Country road networks from the TIGER/Line^{TM} Census Files, which were also tested in Ref. 17. We thank Dr. T. Dung for providing these networks to us. In all our tests, node 0 was taken as the origin.

Table 1. List of test problems

Test	Nodes	Arcs
G1, E1	70756	1000000
G2, E2	50176	1000000
G3, E3	40804	1000000
G4, E4	35344	1000000
N1	31622	1000000
N2	15811	1000000
N3	11952	1000000
N4	10000	1000000
C1	250	62250
C2	500	249500
C3	750	561750
C4	1000	999000
R1	4795	16458
R2	26347	64708
R3	102557	250616
R4	108324	271340

5. Experimental Results and Discussion

We now discuss our experimental results. For each category of test problems we give the sequential (one-processor) and the parallel (8-processor) solution times for each algorithm. We also give the speedup for 4 and 8 processors. We measured speedup for a given problem and for a given algorithm as the ratio of the one-processor time over the multiple-processor time required by the algorithm. A more detailed accounting of our experimental results is given in Ref. 20, and includes the number of iterations and the times required by the synchronous and the asynchronous version of each algorithm on each of the test problems. For the parallel algorithms we report results only with the more efficient multiple-queues versions.

Grid/Random Problems. Fig. 2 gives the sequential execution times and shows that the threshold methods are much faster than the others. For these problems, the threshold methods require a very small number of iterations, almost equal to the number of nodes, which is the lower bound attained by Dijkstra's algorithm. The combinations with the SLF and LLL strategies consistently require a smaller number of iterations than the pure threshold method. However, since the threshold method works very well for these problems, there is little or no further reduction in the serial execution time as a result of the combination and, in some cases, there is a slight time increase due to the extra overhead of the SLF and LLL strategies. However, the SLF and LLL strategies are also very helpful in reducing the number of iterations without a threshold, as can be inferred by comparing the results of the SLF, SLF-LLL, and B-F methods.

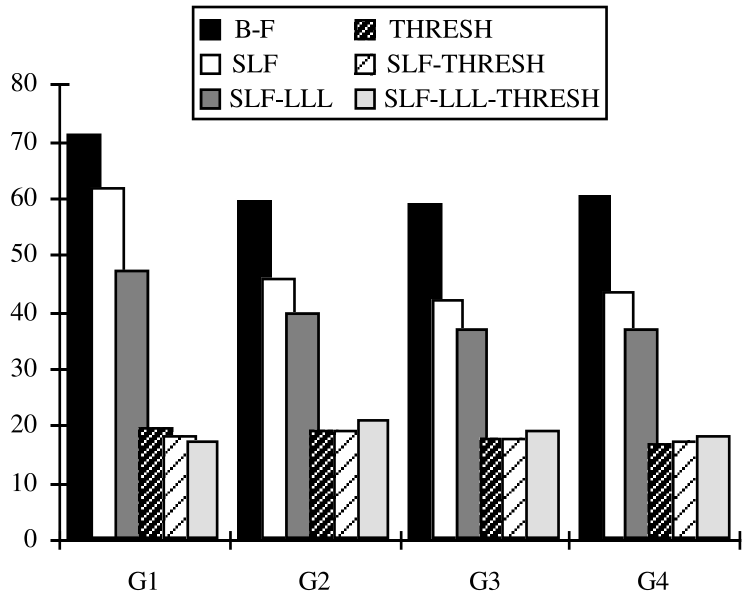

Fig. 2. Time in secs required to solve grid/random problems with the sequential codes

The improvements due to parallelism are summarized in Table 2, where the speedup values using 4 processors and 8 processors are reported for the asynchronous parallel algorithms. Fig. 3 gives the corresponding times using 8 processors.

Table 2. Speedup values for the asynchronous parallel codes

(4 processors / 8 processors)

Problem	B-F	SLF	SLF-LLL	THRESH	SLF THRESH	SLF-LLL THRESH
G1	2.67 / 4.28	2.81 / 5.21	2.51 / 4.48	1.17 / 1.43	1.16 / 1.58	1.11 / 1.59
G2	2.92 / 5.09	3.01 / 4.77	2.49 / 4.61	1.22 / 1.61	1.28 / 1.96	1.27 / 1.95
G3	2.98 / 4.71	2.97 / 5.48	2.52 / 4.75	0.96 / 1.46	1.29 / 1.72	1.32 / 1.81
G4	2.03 / 5.25	3.21 / 6.25	2.75 / 5.47	0.92 / 1.36	1.19 / 1.83	1.28 / 1.81

It can be seen from Table 2 that the performance of the parallel asynchronous threshold methods is poor; a maximum speedup value of only 1.96 is obtained. This is due in part to the difficulty in parallelizing the threshold methods, which involve operations, such as the threshold setting and the transfer of nodes between the two queues, that are inherently sequential. Furthermore, with the use of multiple queues the permanent labeling property of the threshold method is lost, as discussed in Section 2. In addition, in the threshold methods, it is difficult to choose an appropriate threshold, especially in the parallel case, when a threshold must be set for each queue. The SLF and LLL strategies are very helpful in reducing the number of iterations and are well suited for parallelization. An interesting result, especially with SLF, is that the use of multiple queues reduces substantially the number of iterations over the sequential version. This phenomenon was also noted for the other test problems. One possible explanation is that by using multiple queues, the sorting process that places nodes with small labels near the top of the queues is enhanced. The reduction in number of iterations accounts for the particularly good speedup achieved with SLF (up to 6.25 with 8 processors), and also with SLF-LLL.

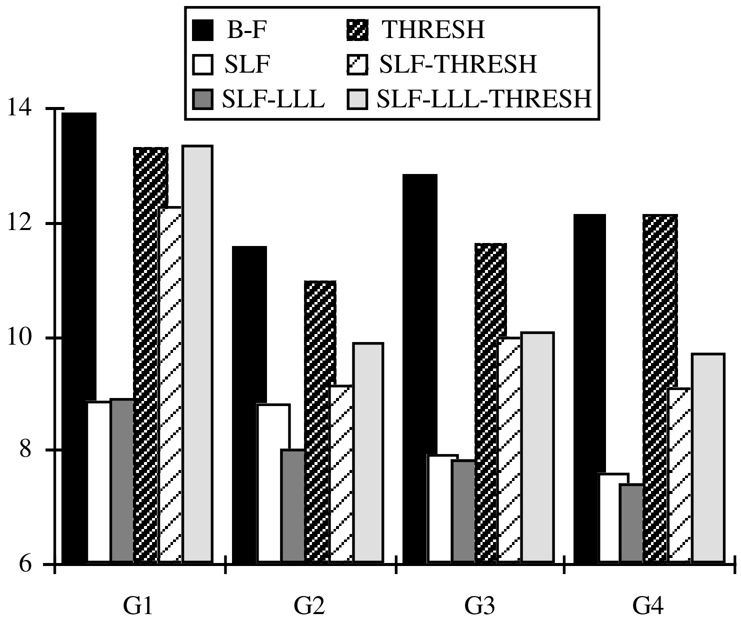

Fig. 3. Time in secs required to solve grid/random problems with the parallel asynchronous codes using 8 processors.

Euclidean Grid/Random Problems. These problems are more difficult than the preceding ones because of the considerable difference between the lengths of the grid arcs and the nongrid arcs. Here THRESH requires a substantially smaller number of iterations that B-F, but the number of iterations of THRESH is quite large (two or three times larger than the number of nodes). The SLF and LLL strategies substantially reduce the number of iterations as can be inferred from Fig. 4. Also in the parallel case we observe a large speedup with SLF and SLF-LLL. In particular, with SLF we achieve maximum speedup of around 6.82, whereas with the SLF-LLL version we achieve a maximum speedup of 5.46. Again, our explanation is that the use of multiple queues enhances the process of examining nodes with small labels first, and results in a reduced number of iterations.

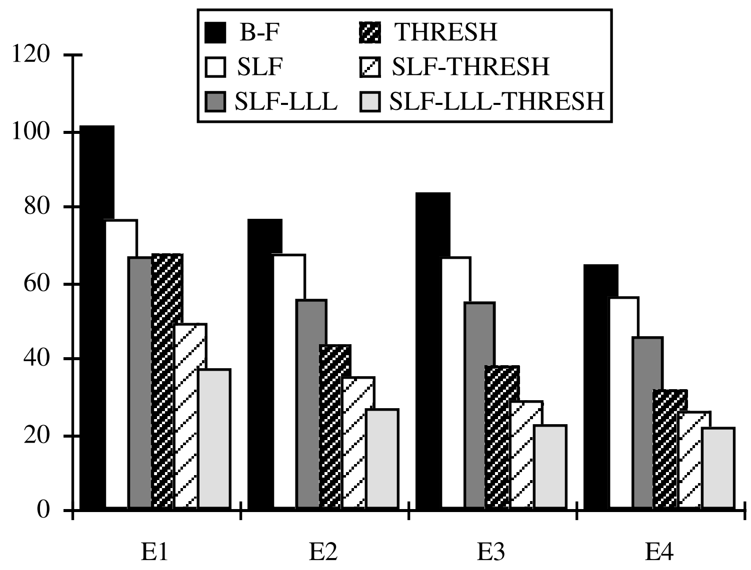

Fig. 4. Time in secs required to solve Euclidean grid/random problems with the sequential codes

Table 3. Speedup values for the asynchronous parallel codes (4 processors / 8 processors)

Problem	B-F	SLF	SLF-LLL	THRESH	SLF THRESH	SLF-LLL THRESH
E1	2.65 / 4.69	2.83 / 5.49	2.28 / 4.54	1.40 / 2.32	1.40 / 2.70	1.13 / 2.12
E2	2.95 / 4.49	3.08 / 6.03	2.50 / 5.10	1.33 / 2.37	1.38 / 2.37	1.03 / 1.54
E3	3.13 / 5.50	3.36 / 6.82	2.78 / 5.46	1.17 / 1.97	1.15 / 2.30	0.89 / 1.82
E4	3.18 / 4.90	3.15 / 6.28	2.51 / 4.41	1.21 / 2.15	0.89 / 2.28	0.95 / 1.97

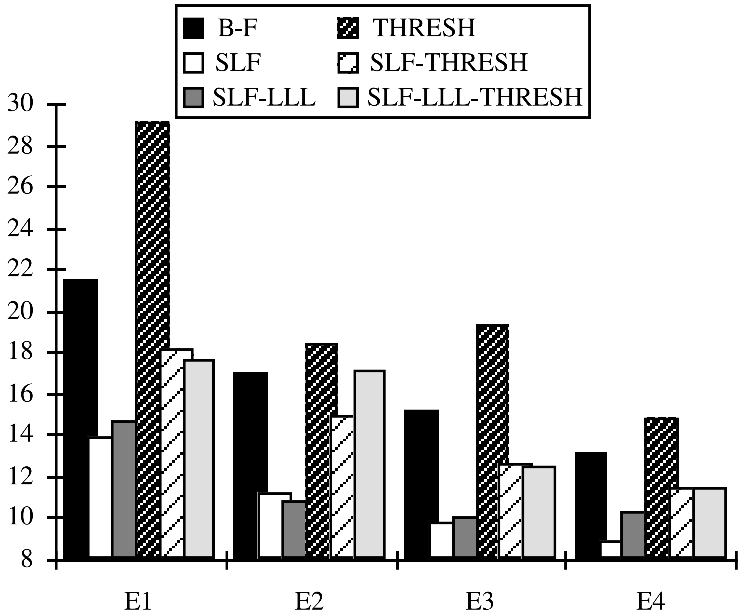

Fig. 5. Time in secs required to solve Euclidean grid/random problems with the parallel asynchronous codes using 8 processors.

Netgen problems. These problems are substantially more dense than the preceding ones, and in the sequential case, the threshold algorithms are much faster than the others. The improvement in execution time relative to B-F is due to the substantial reduction of the number of iterations.

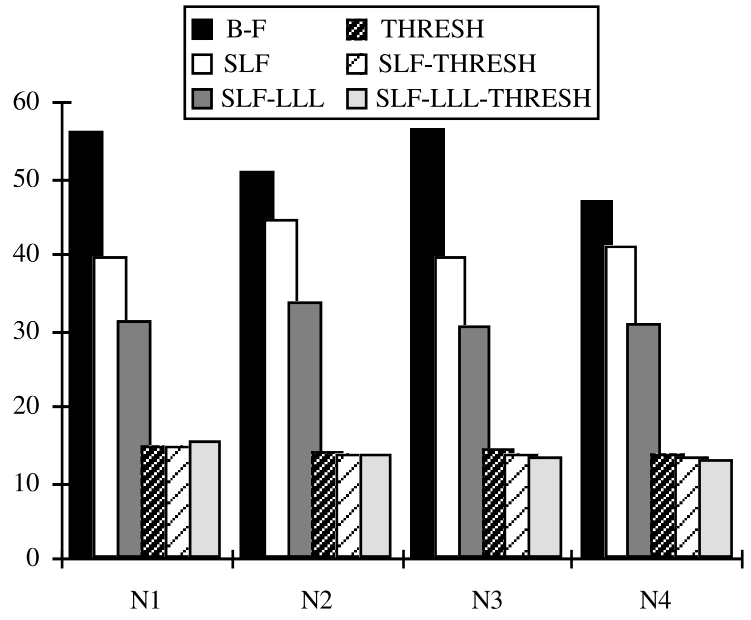

Fig. 6. Time in secs required to solve Netgen problems
with the sequential codes

In the parallel asynchronous case, using multiple queues in combination with the SLF strategy works very well and results in fewer iterations. The reduction in the number of iterations is so large for one of the problems that the speedup is greater than 8 with 8 processors. As a result, the SLF method outperforms all other parallel methods.

Table 4. Speedup values for the asynchronous parallel codes
(4 processors / 8 processors)

Problem	B-F	SLF	SLF-LLL	THRESH	SLF THRESH	SLF-LLL THRESH
N1	3.37 / 6.07	3.10 / 6.12	2.46 / 4.73	1.03 / 1.33	1.11 / 1.68	1.25 / 1.71
N2	3.37 / 6.37	4.46 / 8.56	2.60 / 6.31	0.81 / 1.44	1.06 / 1.85	1.10 / 2.01
N3	3.67 / 6.85	3.93 / 7.12	3.08 / 4.44	0.96 / 1.50	1.19 / 2.16	0.89 / 2.08
N4	3.47 / 6.76	4.51 / 8.45	2.65 / 5.35	0.80 / 1.48	1.06 / 2.04	1.01 / 2.05

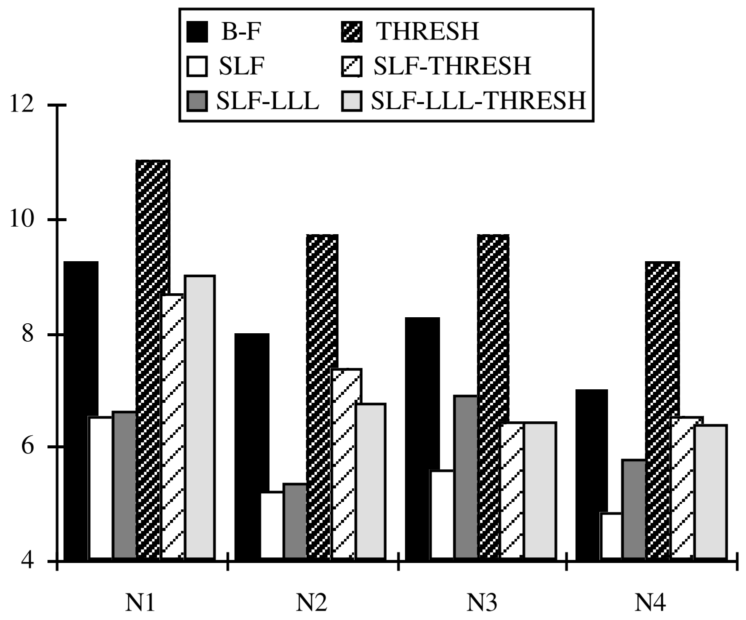

Fig. 7. Time in secs required to solve Netgen problems with the parallel asynchronous codes using 8 processors.

Fully dense problems. For fully dense problems the results are quite similar to those for the preceding problems, as can be seen from Figs. 8-9. The value of speedup is larger for these problems and the parallel performance of the Bellman-Ford method is relatively better than for the preceding problems.

Table 5. Speed-up values for the asynchronous parallel codes (4 processors / 8 processors)

Problem	B-F	SLF	SLF-LLL	THRESH	SLF THRESH	SLF-LLL THRESH
C1	3.96 / 7.26	3.38 / 7.52	2.97 / 5.72	1.90 / 3.18	1.85 / 2.84	1.57 / 2.72
C2	4.03 / 7.50	4.22 / 8.08	3.33 / 5.98	1.88 / 3.59	2.03 / 3.92	1.67 / 2.97
C3	4.10 / 7.73	4.20 / 8.23	3.09 / 5.80	2.32 / 2.96	2.08 / 3.87	1.70 / 3.32
C4	4.21 / 8.02	4.15 / 8.13	3.06 / 6.16	2.09 / 3.05	2.25 / 4.07	1.68 / 3.32

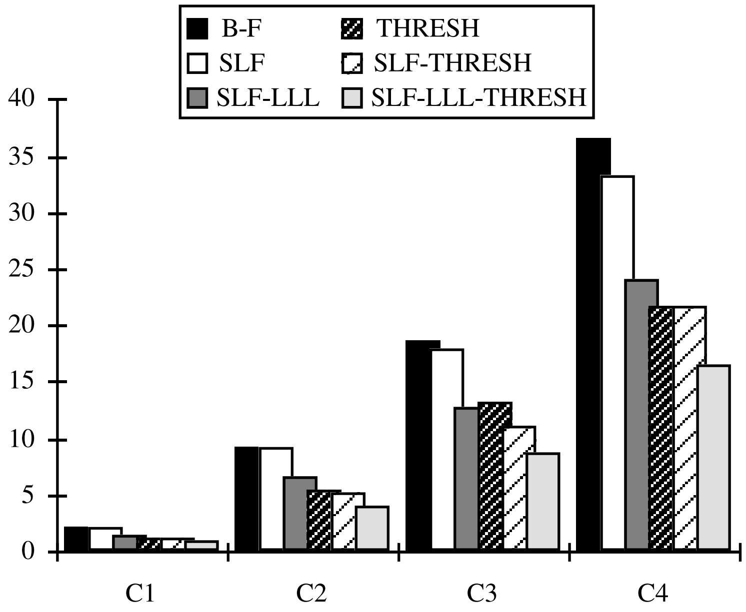

Fig. 8. Time in secs required to solve fully dense problems with the sequential codes

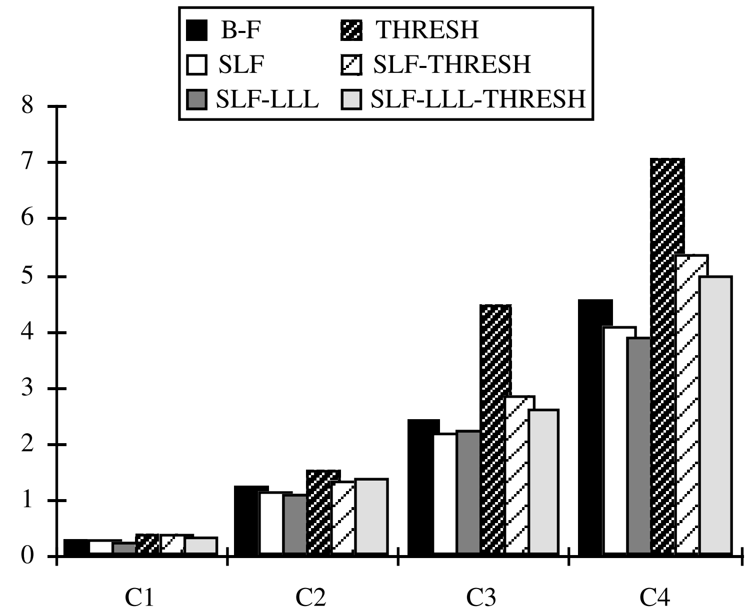

Fig. 9. Time in secs required to solve fully dense problems with the parallel asynchronous codes using 8 processors.

Road networks. For these problems, the SLF and LLL strategies are remarkably effective. In a serial setting they improve a great deal the performance of the Bellman-Ford and the threshold algorithms, as can be seen from Fig. 10. In a parallel setting they exibit excellent (often superlinear) speedup, due to a greatly reduced number of iterations, as can be seen from Table 6 and Fig. 11. The reduction in the number of iterations for the SLF and LLL strategies must be attributed to the use of multiple queues and the associated enhanced sorting that places nodes with small labels near the top of the queues.

Table 6. Speed-up values for the asynchronous parallel codes

(4 processors / 8 processors)

Problem	B-F	SLF	SLF-LLL	THRESH	SLF THRESH	SLF-LLL THRESH
R1	1.81 / 2.49	2.53 / 5.39	2.49 / 5.05	1.16 / 1.70	1.04 / 1.24	0.93 / 1.33
R2	1.94 / 3.35	3.88 / 5.08	4.17 / 7.12	0.69 / 1.20	0.69 / 0.90	0.72 / 1.05
R3	2.21 / 3.60	5.78 / 10.45	17.53 / 21.13	0.59 / 0.49	0.40 / 0.59	0.63 / 0.81
R4	1.84 / 2.43	7.37 / 12.66	11.33 / 18.38	0.64 / 1.24	0.77 / 1.23	0.64 / 0.76

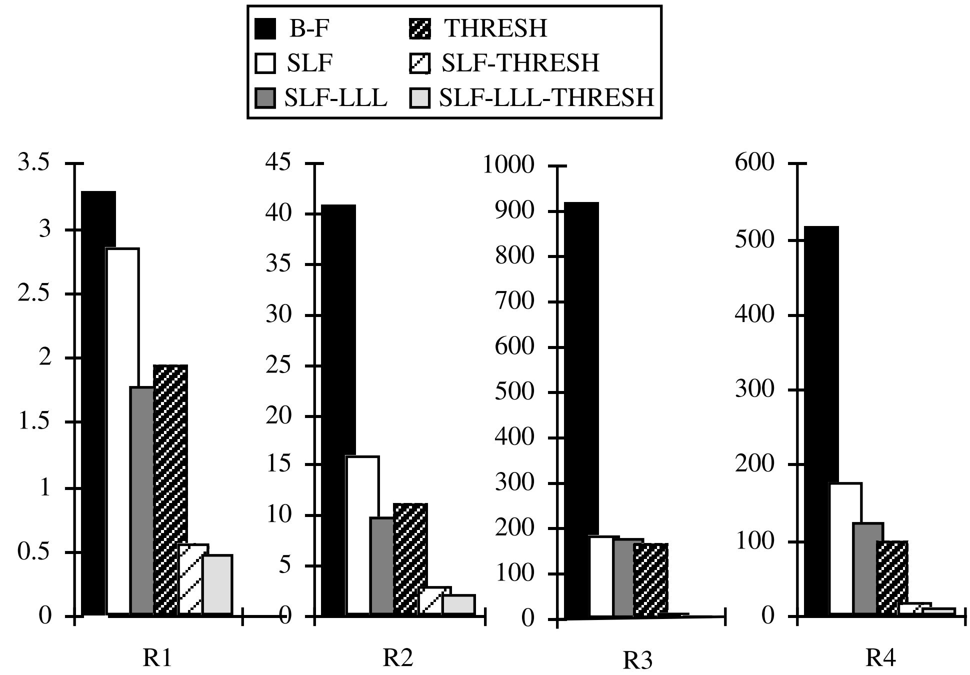

Fig. 10. Time in secs required to solve road network problems

with the sequential codes

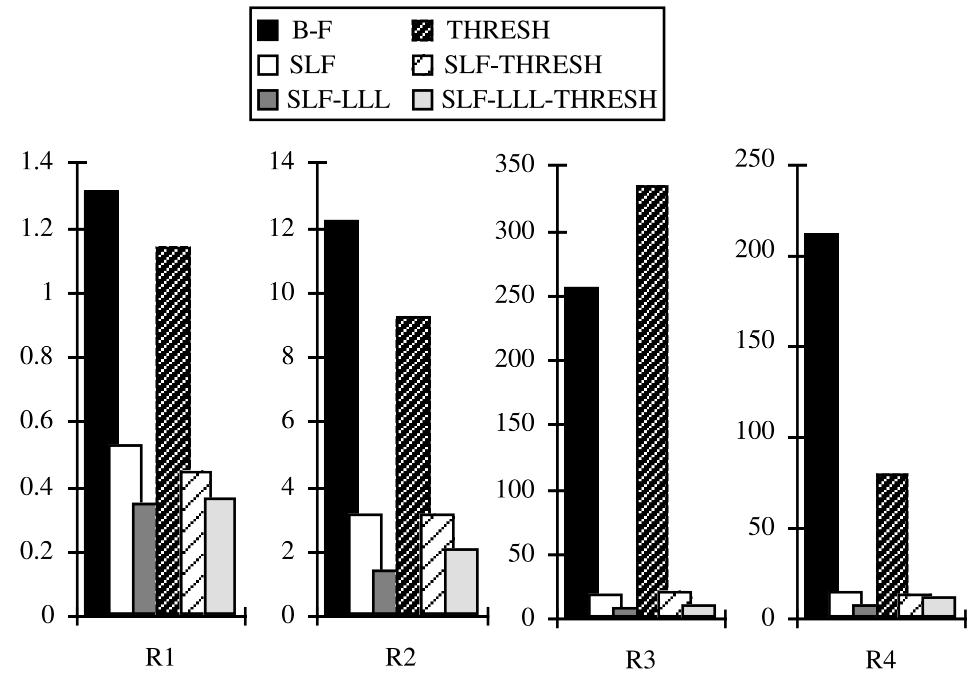

Fig. 11. Time in secs required to solve road network problems with the

parallel asynchronous codes using 8 processors.

In Table 7 and Fig. 12 we aim to summarize the performance of the various methods and also to show the advantage of the asynchronous implementations versus their synchronous counterparts. In particular, we compare the methods following an approach that is similar to the one proposed in Ref. 21, by giving to each method and for each test problem, a score that is equal to the ratio of the execution time of this method over the execution time of the fastest method for the given problem. Thus, for each method, we obtain an average score, which is the ratio of the sum of the scores of the method over the number of test problems. This average score, given in Table 7, indicates how much a particular method has been slower on the average than the most successful method.

Table 7. Average scores of all implemented methods

Code	SEQ	SYN	ASYN
BF	17.86	26.97	4.67
SLF	9.39	5.72	1.21
SLF-LLL	6.10	6.62	1.03
THRESH	8.50	10.14	4.41
SLF THRESH	3.33	2.77	1.44
SLF-LLL THRESH	2.63	2.23	1.33

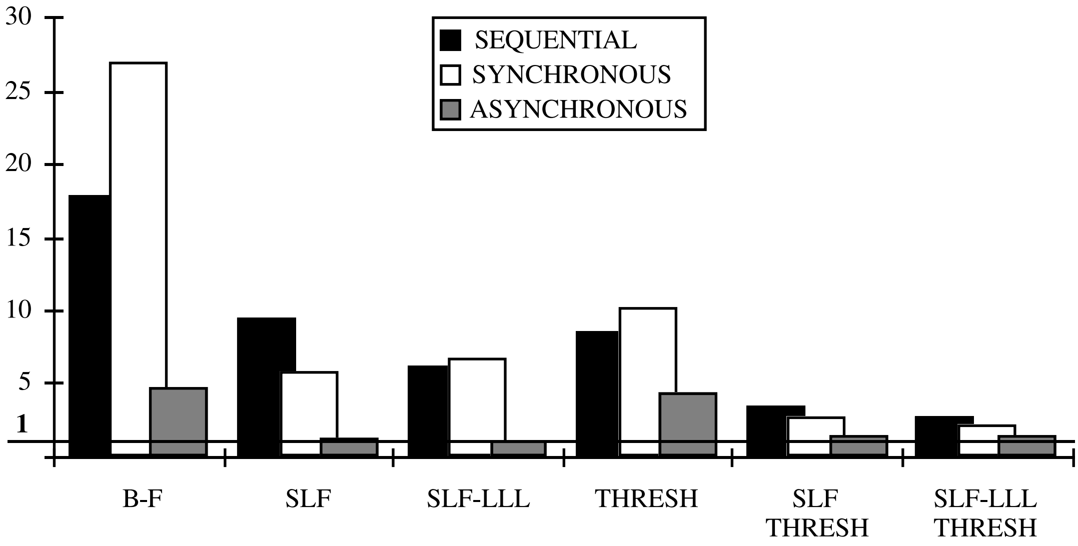

Fig. 12. Plot of the average scores of all implemented methods as per Table 7

In conclusion, the use of multiple queues seems to work very well in conjunction with the SLF and LLL strategies, and the asynchronous parallel algorithms consistently outperform their synchronous counterparts. The threshold method, which is robust and efficient for serial computers, is not well suited for parallelization. Finally, the SLF and LLL strategies maintain their efficiency when implemented in parallel, and when combined with other methods, significantly improve their performance both in a serial and in a parallel environment.

References

1. GALLO, G., and PALLOTTINO, S., Shortest Path Algorithms, Annals of Operations Research, Vol. 7, pp. 3-79, 1988.

2. BERTSEKAS, D. P., An Auction Algorithm for the Shortest Path Problem, Mathematical Programming Study, Vol. 26, pp. 38-64, 1986.

3. BERTSEKAS, D. P., PALLOTTINO, S., and SCUTELLA', M. G., Polynomial Auction Algorithms for Shortest Paths, Massachusetts Institute of Technology, Report No. LIDS-P-2107, 1992.

4. HELGASON, R. V., and STEWART, D., One-to-One Shortest Path Problem: An Empirical Analysis With Two-Tree Dijkstra Algorithm, Computational Optimization and Applications, Vol. 2, pp. 47-75, 1993.

5. POLYMENAKOS, L., and BERTSEKAS, D. P., Parallel Shortest Path Auction Algorithms, Parallel Computing, Vol. 20, pp. 1221-1247, 1994.

6. GALLO, G., and PALLOTTINO, S., Shortest Path Methods: A Unified Approach, Mathematical Programming Study, Vol. 26, pp. 38-64, 1986.

7. BELLMAN, R., Dynamic Programming, Princeton University Press, Princeton, New Jersey, 1957.

8. BERTSEKAS, D. P., A Simple and Fast Label Correcting Algorithm for Shortest Paths, Networks, Vol. 23, pp. 703-709, 1993.

9. GLOVER, F., GLOVER, R., and KLINGMAN, D., The Threshold Shortest Path Algorithm, Networks, Vol. 14, pp. 256-282, 1986.

10. PAPE, U., Implementation and Efficiency of Moore - Algorithms for

the Shortest Path Problem, Mathematical Programming, Vol. 7, pp. 212-222, 1974.

11. BERTSEKAS, D. P., Linear Network Optimization: Algorithms and Codes, M.I.T. Press, Cambridge, Massachusettes, 1991.

12. BERTSEKAS, D. P., Distributed Dynamic Programming, IEEE Transactions on Automatic and Control, Vol. 27, pp. 610-616, 1982.

13. BERTSEKAS, D. P., and GALLAGER, R. G., Data Networks, 2nd edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

14. BERTSEKAS, D. P., and TSITSIKLIS, J. N., Parallel and Distributed Computation: Numerical Methods, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

15. MOHR, T., and PASCHE, C., Parallel Shortest Path Algorithm, Computing, Vol. 40, pp. 281-292, 1990.

16. TRÄFF, J. L., Precis: Distributed Shortest Path Algorithms, Proceedings of the 5th International PARLE Conference, Munich, Germany, 1993, Springer-Verlag, Berlin, pp. 720-723, 1993.

17. DUNG, T., HAO, J., and KOKUR, G., Label Correcting Shortest Path Algorithms: Analysis and Implementation, Unpublished Report, GTE Laboratories, Waltham, Massachusettes, 1993.

18. BERTSEKAS, D. P., and CASTANON, D. A., Parallel Asynchronous Implementations of the Auction Algorithm, Parallel Computing, Vol. 1, pp. 707-732, 1991.

19. KLINGMAN, D., NAPIER, A., and STUTZ, J., NETGEN - A Program for

Generating Large Scale (Un)Capacitated Assignment, Transportation and Minimum Cost Flow Network Problems, Management Science, Vol. 20, pp. 814-822, 1974.

20. BERTSEKAS, D. P., GUERRIERO, F., and MUSMANNO, R., Parallel Asynchronous Label Correcting Methods for Shortest Paths, Massachusetts Institute of Technology, Report No. LIDS-P-2250, 1992.

21. BROWN, A. A., and BARTHOLOMEW-BIGGS, M. C., Some Effective Methods for Unconstrained Optimization Based on the Solution of Systems of Ordinary Differential Equations, Numerical Optmisation Centre, Hatfiled Polytechinc, Hatfiled, England, Report No. 78, 1987.