Presented at 11th IFAC/IFORS/IMACS/IFIP Symposium on Large Scale Systems Theory and Applications, Gdańsk, Poland, 23-25 July 2007

PARALLEL ALGORITHMS FOR MULTICORE ROUTERS IN LARGE COMPUTER NETWORKS - A REVIEW

Andrzej Karbowski [*,**]

[*] *NASK (Research and Academic Computer Network), ul. Wąwozowa 18, 02-796 Warsaw, Poland*
[**] *Institute of Control and Computation Engineering, Warsaw University of Technology, ul. Nowowiejska 15/19, 00-665 Warsaw, Poland,*
E-mail: A.Karbowski@ia.pw.edu.pl

Abstract: In the paper several classical and new algorithms for calculation of routing tables in large networks, e.g.: Dijkstra, Bellman-Ford, SLF-LLL, auction, Q-routing and ant(swarm)-routing are compared - both theoretically and computationally. The special attention is paid to the possibility of making use of current multicore processors in routers and the effectiveness of the parallel implementations.

Keywords: Routing-algorithms, Computer Networks, Optimization problems, Adaptive algorithms

1. INTRODUCTION

In the last 3-4 years a noticeable change in computer hardware took place. Namely, because of the heat problems, the clock frequency of processors stopped to double every 1.5 year and instead, in a single processor die we have several logically independent CPUs, so-called cores. Since the number of computers connected to networks is continually growing, so the number of computers in big domains, to keep up the speed of the routing algorithms, it is necessary to apply parallel calculations of the routing tables. Otherwise, they will adapt to the changing situation in the networks slower and slower, and the Quality of Service of the networks will decrease.

Fortunately, computers with current dual or quad core processors are shared memory machines, what means, that to write the parallel programs a programmer should only know software tools for concurrent computing, that is for thread programming [1] , such as: POSIX threads or OpenMP directives.

One may distinguish two classes of dynamic routing algorithms in computer networks (Perlman, 1999): distance-vector and link-state algorithms. The former are actually fine-grain type and consist in exchanging between the neighboring routers during the calculations the information concerning the estimates of the costs of the best connections to remote routers. At the beginning the routers had no knowledge about the topology and the state of the network (including the current

[1] Using `fork` function from UNIX system library and IPC library is also possible, but not so convenient.

costs of transmission through arcs), they only know their neighbors and the costs of sending packets to them. A representative of this approach is the Bellman-Ford algorithm implemented in the RIP protocol. Because during the optimization all routers have to maintain the tables with current estimates of - usually - the shortest distances to all routers and often communicate with all neighbors, this approach scales badly and is efficient only in small networks.

In the link-state protocols every router has the same global data concerning the state of the whole network, that is: the complete graph, all current costs of transmission along arcs, and calculates independently the best routes from itself to other routers. At present, the Dijkstra algorithm - implemented in OSPF protocol - is used. The problem is, that this algorithm is sequential from its nature, what is a serious drawback in multicore routers.

Hence, there is a need for a good parallel routing algorithm, which may be applied on computers with shared memory, such as equipped with multicore processors modern routers.

In the paper we will review several classical and new routing/shortest path algorithms[2], such as: Dijkstra, Bellman-Ford, SLF-LLL, auction algorithms, Q-routing and ant(swarm)-routing, paying a special attention to their potential for parallelization. Then, we will describe the results of a series of tests on the problems obtained from GRIDGEN, which is a popular network problems generator. At the end some recommendations will be given.

2. ORIGINAL VECTOR BELLMAN-FORD ROUTING ALGORITHM

We consider a directed graph consisting of n nodes (routers). Let us denote by N the set of all these nodes and by A the set of all arcs in the directed graph (that is, the set of all links in the network). Let us assume, that every arc $(i,j) \in A$ is characterized by a positive scalar value a_{ij}, which we will treat as the cost of passage from i to j, that is the distance measure (metric). In most routing problem statements it is taken $a_{ij} = 1 \; \forall (i,j) \in A$, that is the length of the path is simply the number of links (hops).

Let us fix the source node $s \in N$ and assume, that all other nodes may be reached from it. Denote by $d_i^s \triangleq d_i$ the cost of passage from the node s to a node i and by $\hat{d}_i$ its minimum value. It can be easily proved (e.g., by the contradiction), that the paths of the minimum costs (so-called shortest paths) can be obtained through the solution of the following set of equations:

$$\hat{d}_j = \min_{(i,j)\in A} \left(\hat{d}_i + a_{ij} \right) \quad j \neq s \tag{1}$$

Let us take now

$$h_j(d) = \begin{cases} \min\limits_{(i,j)\in A} (d_i + a_{ij}) & j \neq s \\ 0 & j = s \end{cases} \tag{2}$$

and

$$d = [d_1, d_2, \ldots, d_n]$$
$$h = [h_1, h_2, \ldots, h_n] \tag{3}$$

To find the solution we may apply the Bellman-Ford algorithm

$$d := h(d) \tag{4}$$

starting from $d_s = 0$; $d_j = \infty, j \neq s$. This algorithm is based on the order preserving (monotone) mapping h, which may be implemented in the parallel, totally asynchronous version with the arbitrary division of the vector d (Bertsekas and Tsitsiklis, 1997).

The optimization algorithm (2)-(4) may be applied repetitively - to adapt routing to the current situation in a network. In that case the cost a_{ij} should be a measure of the quality of transmission, dependent on the current flow (transmission rate) f_{ij} between nodes i and j. A very popular flow cost function $a_{ij}(.)$ is:

$$a_{ij}(f_{ij}) = \frac{f_{ij}}{(c_{ij} - f_{ij}) + \varepsilon_{ij}} + \delta_{ij} \cdot f_{ij} \tag{5}$$

where c_{ij} is the transmission capacity of arc (i,j) and δ_{ij} is the processing and propagation delay (of course we assume $0 \leq f_{ij} \leq c_{ij}$; $\varepsilon_{ij} > 0$ is a small constant to avoid zero in the denominator). However, in this - adaptive - case one should remember, that the cost functions (5), dependent monotonically on flow, should be augmented with constant components σ_{ij} (so-called bias factors, interpreted as link costs/lengths at zero load), because otherwise oscillations (in subsequent optimal routings) may occur (Bertsekas and Gallager, 1992).

The presented adaptive routing approach is used in the Internet in demons *routed*, *gated* and protocols RIP and Hello (Comer, 1991). In the first protocol so-called "hop count metrics" is used, what means, that simply all elementary arcs are counted for; in the second - "network delay metrics", that is the time of transmission is taken

[2] Routing tables may be calculated by solving shortest-path problems for connections from a given router to every possible destination. In fact, due to Bellman Principle of Optimality, it is not necessary to consider all the destinations.

into account. In the active state, all messages used to the optimization of the routing tables (i.e., the tables of the shortest path neighbors for different destinations) are sent by every computer to all direct neighbors every 30 seconds.

3. GENERIC SHORTEST PATH ALGORITHM

The algorithm presented in the previous section may be treated as a representative of a more general class of algorithms, namely label correcting algorithms. The name "label" refers to the distance $d_t^s \triangleq d_t$ from the source node "s" to a node $t \neq s$, the correction is its change in subsequent iterations of the algorithm leading toward the optimum. We assume, that for each node $t \in N$ we want to find a path of the minimum length (cost) that starts at the node s and ends at t.

A notion of the candidate list of nodes is very useful here. Let us denote it by V. Assuming, that V is nonempty, a typical iteration of a shortest path algorithm (not necessarily of label correcting type) is as follows (Bertsekas, 1993):

Initialization:

$$d_s = 0, \quad d_i = \infty \quad \text{for} \quad i \neq s$$

$$V = \{s\}$$

Typical Iteration of the Generic Shortest Path Algorithm:

Remove a node i from the candidate list V. For each outgoing arc $(i,j) \in A$, with $j \neq s$, if $d_j > d_i + a_{ij}$, set:

$$d_j := d_i + a_{ij} \tag{6}$$

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

Different shortest path algorithms are distinguished by the method of selecting the node to exit the candidate list V at each iteration. For example in Dijkstra method the node exiting V is the node whose label is minimum over all other nodes in V. This guarantees, that every node enters and exits V exactly once and its label is not changed in later iterations; however, finding it is quite costly. Because of that, this method is called label setting method. Label correcting methods avoid the overhead associated with finding the minimum label node at the expense of multiple entrances of nodes into V. In these methods a queue Q is used to maintain the candidate list V. Bellman-Ford method is the simplest method from this family. In the terms of the above generic shortest path algorithm it maintains V in a FIFO queue Q; nodes are removed from the top of the queue and are added at the bottom of Q.

4. SLF AND LLL STRATEGIES

More sophisticated label correcting methods maintain V in one or in two queues (eg. in so-called threshold algorithms the candidate list is partitioned into two separate queues on the basis of some threshold parameter) and use more complex removal and insertion strategies. The objective is to reduce the number of node reentries in V.
In the SLF (Small Label First) method the candidate list V is maintained as double ended queue Q (Bertsekas, 1993). At each iteration, the node removed is the top node of Q. The rule for inserting new nodes is as follows:

SLF method

Let i be the top node of Q, and 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 LLL (Large Label Last) method defines a more complicated strategy of removal a node from Q, which aims to remove from Q nodes with small labels (Bertsekas, 1993). 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 $\overline{\overline{Q}}$ of Q), this node is not removed from Q, but instead it is repositioned to the bottom of Q. This algorithm may be described as follows:

LLL method

Let i be the top node of Q, and let
$c = \sum_{j \in Q} d_j / \overline{\overline{Q}}$.
if $d_i > c$, then move i to the bottom of Q.
Repeat until a node i such that $d_i \leq c$ is found and is removed from Q.

It is possible to combine the SLF queue insertion and the LLL node selection strategies. The resulting method is denoted by SLF-LLL. The proof of convergence of these two algorithms closely resembles the proof of the convergence of Bellman-Ford algorithm and is based on the monotonicity property of the label modification mapping (Bertsekas *et al.*, 1996).

In the parallel implementation, each processor removes the top node from Q (perhaps after some shifts of the queue in the case of LLL strategy), updates the labels of its adjacent nodes and adds these nodes (if necessary) into Q according to SLF insertion strategy. This means, that several nodes can be simultaneously removed from the candidate list and the labels of the adjacent nodes can be updated in parallel. Because the algorithm is in fact asynchronous, a new node may be removed from the candidate list by some processor while other processors are still updating the labels

of other nodes. Of course, only one processor at a time can modify a given label. Hence, it is very easy to implement this method on a parallel multicore shared-memory (or NUMA) machines using locks to assure the consistency of data. Unfortunately, the synchronization overhead may be quite big. To reduce it, a multiple queues version of this algorithm was proposed (Bertsekas *et al.*, 1996), in which each processor uses a separate queue (a node can reside in at most one 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. For example, as proposed in (Bertsekas *et al.*, 1996), it may be the one with the minimum current value of the sum of the outgoing arcs in that list.

In this work a simpler method was implemented: for a given removed node, all its neighbours which are not present in queues, are added to the same queue, namely the shortest one.

These heuristics ensure good load balancing among the processors. For checking whether a node is present in the candidate list (that is, in some queue) it is suggested (Bertsekas *et al.*, 1996), to associate with every node a Boolean variable, which is updated each time a node enters or exits a candidate list.

Other label correcting methods, e.g. with a threshold dividing queues are presented in (Bertsekas *et al.*, 1996).

5. AUCTION ALGORITHM

An alternative to the presented algorithms seem to be auction algorithms (Polymenakos and Bertsekas, 1994). The most suitable for multicore routers is so-called "reverse" version, where the algorithm maintains a path ending at a terminal node t and a negative price for each node. The starting node of the path "bids" for neighboring nodes basing on their prices and the lengths of the connecting arcs. At each iteration the path is either extended by adding a new node or contracted by deleting its starting node. When the starting node becomes the source node of the path s (i.e., that of the current router), the algorithm terminates.

To present the algorithm in a formal way, let us denote by P a path ending at the terminal node, that is: $P = (i_k, i_{k-1}, \ldots, i_1, t)$, where $(i_l, i_{l-1}) \in A,\ l = 1, \ldots, k,\ i_0 = t$. We assume that $i_{j_1} \neq i_{j_2},\ j_1 \neq j_2$, that is a path does not contain any cycle. The node i_k is called the starting node of P. The degenerate path $P = (t)$ may be also obtained in the course of the algorithm.
If i_{k+1} is a node that does not belong to a path $P = (i_k, i_{k-1}, \ldots, i_1, t)$ and (i_{k+1}, i_k) is an arc, extending P by i_{k+1} means replacing P by the path $(i_{k+1}, i_k, i_{k-1}, \ldots, i_1, t)$. If P does not consist of just the terminal node t, contracting P means replacing P with the path $(i_{k-1}, \ldots, i_1, t)$.

In addition to the path, the algorithm maintains a price λ_i for each node $i \in N$ in the network. Let us denote by λ the vector of prices λ_i. We say that a path-price pair (P, λ) satisfies complementary slackness (CS) conditions if the following relations hold:

$$\lambda_i \leq a_{ij} + \lambda_j, \quad \forall (i,j) \in A \tag{7}$$

$$\lambda_i = a_{ij} + \lambda_j, \text{ for all pairs of successive nodes } i \text{ and } j \text{ of } P \tag{8}$$

An important property is that if a path-price pair (P, λ) satisfies CS, then portion of P between any node $i \in P$ and the terminal node t is the shortest path from i to t and $d_t^i = \lambda_i - \lambda_t$ is the corresponding shortest distance.

The algorithm proceeds in iterations, transforming a pair (P, λ) satisfying CS into another pair satisfying CS. At each iteration the path P is either extended by a new node or else is contracted by deleting its starting node. In the latter case the price of the starting node is decreased strictly. In may be described in the following way:

Typical iteration of the Reverse Auction Algorithm

Let j be the starting node of P.

- Step 0: (Scanning of predecessor nodes) If
$$\lambda_j > \max_{\{i|(i,j)\in A\}} \{\lambda_i - a_{ij}\} \tag{9}$$
go to Step 1; else go to Step 2.
- Step 1: (Contract path) Set
$$\lambda_j := \max_{\{i|(i,j)\in A\}} \{\lambda_i - a_{ij}\} \tag{10}$$
and if $j \neq t$, contract P, that is, delete j from P. Go to the next iteration.
- Step 2: (Extend path) Extend P by node i_j where
$$i_j = \arg \max_{\{i|(i,j)\in A\}} \{\lambda_i - a_{ij}\} \tag{11}$$
that is add i_j as the starting node of P. If i_j is the origin s, stop; P is the desired shortest path. Otherwise, go to the next iteration.

The algorithm starts with the default pair:

$$P^t = (t), \quad \lambda_i = 0,\ \forall i$$

and stops when the source node s becomes the starting node of the path.

This algorithm may be easily parallelized for one-source many destinations problem by maintaining reverse paths for different destinations $t \in \{N \setminus \{s\}\}$. There is a queue of the terminal nodes. A free processor picks the first node from the queue and performs the above algorithm taking this node as the terminal node t. To ensure the consistency of the data, when a path P^t extends to a node i, the memory location of the price of i is locked by the thread processing this path (i.e., the one which took the terminal node t). Thus no other path can extend to i or to change the contents of the memory location. To reduce the synchronization overheads, if a processor is unsuccessful in locking on the price of a node, while working with a path P^t, it stops processing, cancels calculations, puts the node t at the end of the queue and takes a new terminal node from the top of the queue. If the shortest distance path from the source to a terminal node is found, all the nodes along this path are removed from the queue permanently.

6. Q-ROUTING - A NEW ALGORITHM BASED ON SIMULATION

Q-routing is quite a new approach based on the reinforcement learning. The algorithm is distributed from its nature and needs a simulator for generating the traffic from the source node s.
In this algorithm every node $i \in N$ maintains a table of values $Q_{ij}^t, j \in N_i$, of all costs (usually they are simply times) to go from this node to all destination nodes t via all the neighbors N_i of this node. The packet is routed to this neighbor $j \in N_i$ for which the value Q_{ij}^t is minimum. Learning takes place by updating the Q_{ij}^t values: after sending a packet to j, the node i gets back j's estimate for the cost of the remaining part the trip, that is:

$$Q_j^t \triangleq \min_{k \in N_j} Q_{jk}^t \tag{12}$$

If the denote by q_{ij} the cost of visiting the node i and the passage from i to j (e.g., it is time spent by a packet in the i-th node queue and transmission between i and j), then the node i can revise its estimate using the correction (Littman and Boyan, 1993):

$$\Delta Q_{ij}^t = \eta(q_{ij} + Q_j^t - Q_{ij}^t) \tag{13}$$

where $\eta \in (0, 1)$ is the learning rate. So, the new estimate of the local cost is the convex combination of the old and the new estimate:

$$\begin{aligned} Q_{ij}^t &:= Q_{ij}^t + \Delta Q_{ij}^t \\ &= (1-\eta)Q_{ij}^t + \eta(q_{ij} + Q_j^t) \end{aligned} \tag{14}$$

7. ANT ROUTING - AN ALGORITHM INSPIRED BY THE NATURE

Ant routing imitates the behavior of ants. In nature, ants lay down a thin layer of signalling chemicals, called pheromones, wherever they travel to find food. When other ants detect these pheromones, they instinctively follow the path the chemicals mark. The thicker the pheromone trail, the more likely other ants will follow the path (Lu *et al.*, 2004). The intensity of pheromones is represented in node i by the probabilities p_{ij}^t, of choosing the neighbor $j \in N_i$ by the packet for which the destination node is t; of course

$$\sum_{j \in N_i} p_{ij}^t = 1, \ \forall t \in \{N \setminus \{i\}\} \tag{15}$$

In a source node, at regular intervals an ant is launched, which goes to a randomly selected destination node t. The route of this ant depends on the random selections in the visited nodes, due to the probabilities p_{ij}^t. On the way to the node t, the ant calculates and memorizes the costs of the travel, so to estimate the cost d_{ij}^t of achieving the destination node t from i via the neighboring node j. This cost will be then used to modify the probability (pheromone) tables p_{ij}^t, $j \in N_i, t \in \{N \setminus \{i\}\}$ by so called "backward" ant, which in turn is deterministic and much faster ("with a higher priority") and goes back on the same way as the first - "forward" - ant. When the backward ant on the way from the node t arrives to the current node i via the neighboring node j, it updates the estimated cost (usually delay) D_{ij}^t in the following way:

$$\begin{aligned} D_{ij}^t(k+1) = D_{ij}^t(k) + \\ \eta\left((1-\alpha)\Delta D_{ij}^t(k|k) + \alpha \cdot \Delta D_{ij}^t(k|k-1)\right) \end{aligned} \tag{16}$$

$$\Delta D_{ij}^t(k|k) = d_{ij}^t - D_{ij}^t(k) \tag{17}$$

$$\Delta D_{ij}^t(k|k-1) = D_{ij}^t(k) - D_{ij}^t(k-1) \tag{18}$$

where:

$\eta \in (0, 1)$	- learning coefficient (eg. 0.2)
α	- momentum parameter ($0.2 \div 0.45$)
k	- the number of iteration

Then, the routing table at the node i giving the probabilities p_{ij}^t of selecting arc (i, j) for a packet with destination t is recalculated as follows:

$$p_{ij}^t = \frac{\left(\frac{1}{D_{ij}^t(k)}\right)^\beta}{\sum_{l \in N_i}\left(\frac{1}{D_{il}^t(k)}\right)^\beta}$$

where $\beta > 1$ is a nonlinearity parameter (e.g. $\beta = 4$).

8. COMPUTATIONAL RESULTS AND CONCLUSIONS

The algorithms presented above were tested on a 4 core Sun Fire V440 computer, with 1.593 GHz clock. Four networks consisting of: 1000, 2500, 5000 and 10000 nodes were generated by the GRIDGEN generator (Lee and Orlin, 1991). They will be denoted by "Gn", where n is the number of nodes. The lengths of arcs (i.e., the costs of transmissions through them) were chosen randomly with the uniform distribution over the $[1, 1000]$ interval. The nodes in all networks had the average degree 10.

The calculations in the case of the sequential versions of the Q- and Ant-routing were very long - about three orders of magnitude longer than the other algorithms and because of that their results are neglected. It is not surprising, because they are both variants of Monte-Carlo based simulation. They both were designed for distributed systems and are actually vector-distance algorithms. They assume, that there is no knowledge about the system and lose a lot of time on acquiring this knowledge.

Some results for the other algorithms in sequential versions are presented in Tab. 1. The symbol "BFv" denotes Bellman-Ford algorithm in the original, vector version, "BFq" - the queue version of this algorithm, "BFq-LLL" - a combination of "BFq" removing with "LLL" insertion strategy.

Table 1. Execution times (in seconds) of sequential versions

	Problem			
Method	G1000	G2500	G5000	G10000
Dijkstra	0.03	0.20	0.87	3.97
BFv	0.14	1.41	6.43	39.27
Auction	1.97	18.48	122.66	-
BFq	0.04	0.24	1.02	4.72
SLF	0.03	0.18	0.85	3.57
BFq-LLL	0.03	0.19	0.88	3.87
SLF-LLL	0.03	0.17	0.78	3.46

It is seen, that the fastest is SLF-LLL algorithm, the second is SLF algorithm; only these two are better than the simplest Dijkstra algorithm. A little slower than Dijkstra is BFq queue algorithm, more than one order of magnitude slower is BFv algorithm and totally disappointing - 2 orders slower than Dijkstra - is auction algorithm.

The fastest of these algorithms and the original BFv algorithm were implemented in a parallel version in C language with the help of Pthreads library. The results are shown in the Tab. 2. It is seen, that in the parallel version LLL method showed its drawback - higher overheads on synchronization while exploring the queue to find the right node to remove (locks!). The algorithms using LLL were much slower than their sequential versions. The best was again SLF algorithm giving on 4 cores 40% of speedup over the Dijkstra algorithm. Also BFq was faster but not so much.

Hence, the best algorithm for multicore routers: simple, fast and scalable seems to be SLF.

Table 2. Execution times (in seconds) of parallel versions

	Problem G5000/ G10000		
Method	2 cores	3 cores	4 cores
BFv	4.22/20.83	3.03/15.37	2.38/12.22
BFq	1.21/5.40	0.78/4.24	0.67/3.50
SLF	1.07/4.03	0.60/2.98	0.53/2.48
BFq-LLL	4.17/17.90	2.64/11.31	1.88/7.95
SLF-LLL	3.86/16.12	2.53/9.84	1.69/6.87

REFERENCES

Bertsekas, D. P. (1993). A simple and fast label correcting algorithm for shortest paths. *Networks* **23**, 703–709.

Bertsekas, D. P. and J. Tsitsiklis (1997). *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific. Belmont.

Bertsekas, D. P. and R. Gallager (1992). *Data Networks*. Prentice-Hall. Englewood-Cliffs.

Bertsekas, D. P., F. Guerriero and R. Musmanno (1996). Parallel asynchronous label-correcting methods for shortest paths. *Journal of Optimization Theory and Applications* **88**, 297–321.

Comer, D.E. (1991). *Internetworking with TCP/IP, Vol. I : Principles, Protocols and Architecture*. Prentice-Hall. Englewood-Cliffs.

Lee, Y. and J. Orlin (1991). Gridgen. ftp://dimacs.rutgers.edu/pub/netflow/generators/network/gridgen/.

Littman, M. and J. Boyan (1993). Packet routing in dynamically changing networks: A reinforcement learning approach. In: *Advances in Neural Information Processing Systems* (J. D. Cowan, G. Tesauro and J. Alspector, Eds.). Vol. 6. pp. 671–678. Morgan Kaufmann. San Francisco.

Lu, Y., G.-Z. Zhao, F.-J. Su and X.-R. Li (2004). Adaptive swarm-based routing in communication networks. *Journal of Zhejiang University Science* **5**, 867–872.

Perlman, R. (1999). *Interconnections: Bridges, Routers, Switches and Internetworking Protocols*. Addison Wesley Professional. Reading.

Polymenakos, L. C. and D.P. Bertsekas (1994). Parallel shortest path auction algorithms. *Parallel Computing* **20**, 1221–1247.