FI MU

Faculty of Informatics
Masaryk University

Parallel Algorithms for Detection of Negative Cycles

by

Luboš Brim
Ivana Černá
Lukáš Hejtmánek

FI MU Report Series FIMU-RS-2003-04

July 2003

Parallel Algorithms for Detection of Negative Cycles*

Luboš Brim, Ivana Černá and Lukáš Hejtmánek

Faculty of Informatics
Masaryk University Brno
Botanická 68a
Czech Republic

{brim,cerna,xhejtman}@fi.muni.cz

July 8, 2003

Abstract

Several new parallel algorithms for the single source shortest paths and for the negative cycle detection problems on directed graphs with real edge weights and given by adjacency list are developed, analysed, and experimentally compared. The algorithms are to be performed on clusters of workstations that communicate via a message passing mechanism.

1 Introduction

The single source shortest path problem (SSSP) is a fundamental problem with many theoretical and practical applications and with several effective and well-grounded sequential algorithms. The same can be said about the closely related negative cycle detection problem (NCD) which is to find a negative length cycle in a graph or to prove that there are none. In fact, all known algorithms for NCD combine a shortest paths algorithm with some cycle detection strategy.

*This work has been partially supported by the Grant Agency of Czech Republic grant No.201/03/0509.

In many applications we have to deal with extremely large graphs (a particular application we have in mind is briefly discussed bellow). Whenever a graph is too large to fit into memory that is randomly accessed a memory that is sequentially accessed has to be employed. This causes a bottleneck in the performance of a sequential algorithm owing to the significant amount of paging involved during its execution. An obvious approach to deal with these practical limitations is to increase the computational power (especially randomly accessed memory) by building a powerful (yet cheap) distributed-memory cluster of computers. The computers are programmed in single-program, multiple-data style, meaning that one program runs concurrently on each processor and its execution is specialised for each processor by using its processor identity (id). The program relies on a communication layer based on Message-Passing Interface standard.

Our motivation for this work was to develop a distributed model checking algorithm for linear temporal logic. This problem can be reduced to the negative cycle detection problem as shown in [1]. The resulting graph is not completely given at the beginning of the computation through its adjacency-list or adjacency-matrix representation. Instead, we are given a root vertex together with a function which for every vertex computes its adjacency-list. A possible approach is to generate the graph at first and then to process it with a distributed NCD algorithm. However, this approach is highly non-efficient. If one processes the graph simultaneously with its formation it can happen that a negative cycle is detected even before the whole graph is formated. Moreover, this *on-the-fly* technique allows to generate the part of the graph reachable from the root vertex only and thus reduces the space requirements. As successors of a vertex are determined dynamically there is no need to store any information about edges permanently which brings yet another reduction in space complexity.

A natural starting point for building a distributed algorithm is to distribute an efficient sequential algorithm. Because of the aforementioned reasons we have concentrated on algorithms which admit graphs specified with the help of adjacency lists (and omit those presupposing an adjacency matrix representation of the graph). These algorithms (for an excellent survey see [2]), which are based on relaxation of graph's edges, are inherently sequential and their parallel versions are known only for special settings of the problem. For general digraphs with non-negative edge lengths parallel algorithms are presented in [3, 4, 5] (see [6] for a comparative study) together with studies concerning suitable decomposition [7]. For special cases of graphs, like planar digraphs [8, 9], graphs with separator decomposition [10] or graphs with small tree-width [11] more efficient algorithms are known. Yet none of these known algorithms are applicable to general digraphs with potential negative-length cycles. In this paper we propose several parallel algorithms for the general SSSP and NCD problems on graphs with real edge lengths and given by adjacency lists. We analyse their worst-case complexity and conduct an extensive practical performance study of these algorithms. We study various combinations of distributed shortest path algorithms and distributed cycle detection strategies to determine the best combination measured in terms of their *scalability*.

2 Serial Negative Cycle Problem

We are given a triple (G, s, l), where $G = (V, E)$ is a directed graph with n vertices and m edges, $l : E \to R$ is a *length function* mapping edges to real-valued lengths, and $s \in V$ is the *root* vertex. The *length of the path* $\rho =< v_0, v_1, \dots, v_k >$ is the sum of the lengths of its constituent edges, $l(\rho) = \sum_{i=1}^{k} l(v_{i-1}, v_i)$. *Negative cycle* is a cycle $\rho =< v_0, v_1, \dots, v_k, v_0 >$ with length $l(\rho) < 0$. The *negative cycle detection* (NCD) problem is to find a negative cycle in a graph or to prove that there are none.

Algorithms for the NCD problem that use the adjacency-list representation of the graph construct a shortest-path tree $G_s = (V_s, E_s)$, where V_s is the set of all vertices reachable from the root s, $E_s \subseteq E$, s is the root of G_s, and for every vertex $v \in V_s$ the path from s to v in G_s is a shortest path from s to v in G.

The *labeling method* maintains for every vertex v its *distance label* $d(v)$ and *parent* $p(v)$. Initially $d(v) = \infty$ and $p(v) = null$, the method starts by setting $d(s) = 0$. The method maintains for every vertex its status which is either *unreached*, *labeled*, or *scanned*, initially all vertices but the root are *unreached* and the root is *labeled*. The method is based on the *scan operation*. During scanning a labeled vertex v, all edges $(v,)$ out-going from v are *relaxed* which means that if $d(u) > d(v) + l(v, u)$ then $d(u)$ is set to $d(v) + l(v, u)$ and $p(u)$ is set to v. The status of v is changed to *scanned* while the status of u is changed to *labeled*. During the computation the edges $(p(v), v)$ for all v : $p(v) \neq null$ induce the *parent graph* G_p. If all vertices are either *scanned* or *unreached* then d gives the shortest path lengths and G_p is the shortest-path tree. On the contrary, any cycle in G_p is negative and if the graph contains a negative cycle then after a finite number of scan operations G_p always has a cycle [2]. This fact is used for the negative cycle detection.

2.1 Scanning strategies

Different strategies for selecting a *labeled* vertex to be scanned next lead to different algorithms.

The *Bellman-Ford-Moore algorithm* [12, 13] uses for selecting the FIFO strategy and runs in $\mathcal{O}(nm)$ time. The *D'Escopo-Pape algorithm* [14] makes use of a priority queue. The next vertex to be scanned is removed from the head of the queue. A vertex that becomes labeled is added to the head of the queue if it has been scanned previously, or to the tail otherwise. The *Pallotino's algorithm* [15] maintains two queues. The next vertex to be scanned is removed from the head of the first queue if it is nonempty and from the second queue otherwise. A vertex that becomes labeled is added to the tail of the first queue if it has been scanned previously, or to the tail of the second queue otherwise. Both last mentioned algorithms favour recently scanned vertices and run in $\mathcal{O}(n^2m)$ time in the worst case, assuming no negative cycles. The *network simplex algorithm* [16] maintains the invariant that in the current parent graph all edges have zero reduced cost (the reduced cost of an edge (v, u) is $l(v, u)+d(u)-d(v)$). Therefore, if the distance label of a vertex u decreases, the algorithm decreases labels of vertices in the subtree rooted at u by the same amount. Then a new edge with negative reduced cost (so called *pivot*) is found and the process continues. There are several heuristics to find a pivot. One can search the scanned vertices and choose the pivot according to a FIFO strategy or depending on the value of the reduced cost. The algorithm runs in $\mathcal{O}(n^2m)$ time.

There are several other algorithms, like e.g the Goldberg-Radzik and the Goldfarb et.al. [17, 18], which however make use of topological sorting and leveling of the parent graph respectively and thus are not directly convertable into distributed versions.

2.2 Cycle Detection strategies

Besides the trivial and non-efficient cycle detection strategies like time out and distance lower bound, the algorithms put to use one of the following strategies: *walk to root*, *subtree traversal* and *subtree disassembly*.

The *walk to root* method tests whether G_p is acyclic. Suppose the parent graph G_p is acyclic and the scanning operation relaxes an edge (v, u). This operation will create a cycle in G_p if and only if u is an ancestor of v in the current parent tree. Before applying the operation, the method follows the parent pointers from v until it reaches u or a vertex with null parent (on this path only the root can have null parent). If the walk stops at u a negative cycle has been found; otherwise, the scanning operation does not create a cycle. The walk to root method gives immediate cycle detection. However, since the path to the root can be long, the relaxation cost becomes $\mathcal{O}(n)$ instead of $\mathcal{O}(1)$. In order to optimise the overall computational complexity we propose to use amortisation to pay the cost of checking G_p for cycles. More precisely, the parent graph G_p is tested only after the underlying scanning algorithm performs $\Omega(n)$ work. A drawback of the amortised strategy is the fact that even if the relaxation of an edge (v, u) does not create a cycle in G_p, there can be a cycle on the way from v to the root. Therefore the strategy marks every vertex through which it proceeds. A cycle is detected whenever an already marked vertex is reached. If G_p is acyclic, all marks are removed. The running time is thus increased only by a constant factor. The correctness of the amortised strategy is based on the fact that if G contains a negative cycle reachable from s, then after a finite number of scanning operation G_p always has a cycle [2].

The *subtree traversal* method makes use of a symmetric idea: the relaxation of an edge (v, u) can create a cycle in G_p if and only if v is an ancestor of u in the current tree. This strategy fits naturally with the network simplex method as the subtree traversal can be combined with the updating of the pivot subtree.

The *subtree disassembly* method also searches the subtree rooted at u. However, this time if v is not in the subtree, all vertices of the subtree except u are removed from the parent graph and their status is changed to *unreached*. The work of subtree disassembly is amortized over the work to build the subtree and the cycle detection is immediate.

3 Distributed Negative Cycle Detection Algorithms

We develope distributed versions of aforementioned serial algorithms. These parallel algorithms are enriched by several novel ideas. To the best of our knowledge these are the first algorithms for the considered setting of NCD and SSSP problems. The pseudo-codes are given in Subsection 3.3.

The distributed algorithms are designed for a cluster of workstations. Each workstation executes the same algorithm. In addition, we consider a distinguished workstation (called the *manager*) which is responsible for the initialization of the entire computation, termination detection and synchronization. The vertices of the input graph are divided equally and randomly into disjoint parts by a *partition function*.

3.1 Distributed scanning strategies

The scanning strategies used in the first three serial algorithms can be converted into their distributed counterparts at no cost and preserving the asymptotic complexity. Correctness of the serial algorithms does not depend on the order in which relaxations are performed. Therefore we can maintain local queues and each processor scans vertices in their relative order.

The network simplex algorithm chooses the pivot according to the FIFO strategy. We only need to provide a distributed version of the subtree update to maintain the invariant concerning zero reduced costs. Thanks to the fact that the parent graph does not contain any cycle (first some cycle detection strategy is employed) one can traverse the subtree in the breadth first manner without the necessity to mark visited vertices. The breadth first search is well distributable. The asymptotic complexity is preserved.

3.2 Distributed Cycle Detection strategies

All the three considered cycle detection strategies can be modified preserving their asymptotic time complexity.

The subtree traversal method requires only a minor modification. The breadth first traversal of the parent tree can be distributed in a natural way. Thanks to the asynchronous relaxations it can happen that the structure of the subtree is modified before succesfull completion of the subtree traversal and thus a “false” negative cycle can be detected. To recognize such a situation it is enough to count the distance from the subtree root to the particular vertices.

The subtree disassembly strategy is more involved. When dissasemblig the subtree we need to maintain distances as in the previous strategy to discover “false” cycles. On top of that it can happen that a cycle in the subtree spanning over several processors is not discovered due to strictly synchronous sequence of relaxations and subtree disassembles. In such a situation the cycle is detected using the distance lower bound.

The walk to root strategy follows the parent pointers starting from the vertex where the detection has been invoked and marking the vertices through which it proceeds. At the same time several detections can be invoked (on different processors). Therefore every processor has its own mark and all marks are linearly ordered. If a walk reaches a vertex marked with a lower or higher mark it overwrites this mark or it stops respectively. A vertex marked with the same stamp implies the presence of negative cycle. After finishing the detection the strategy needs to remove the marks starting from the vertex where the detection has been initiated. To find all the marked vertices the parents of the vertices can not be changed in the meantime. This is guaranteed by locking the marked vertices.

3.3 The pseudo-codes

Program for a particular processor is specialised by using its processor identity α. The function $Owner(u)$ identifies the processor that owns vertex u.

Each processor runs the $Main$ procedure with root vertex s. The procedure $send\_msg$ sends a message to another processor. The procedure $process\_messages()$ checks the incoming messages queue and does an appropriate action.

The pseudo-codes for D'Escopo Pape and Pallotino heuristics are not included as they can be obtained from the Bellman-Ford-More algorithm by a straightforward modification.

```
proc Main(s)
   InitializeSingleSource(); Q^α := empty;
   if α = Manager then push(Q^α, s); d(s) := 0; p(s) := nil; fi
   while not finished do
         if Q^α ≠ empty then u := pop(Q^α); {STD, STT, WTR}_Scan(u); fi
         process_messages();
   od
```

```
proc InitializeSingleSource()
   foreach v ∈ V do if Owner(v) = α then p(v) := nil; d(v) := ∞; fi od
```

Bellman-Ford-Moore with Subtree Disassembly

```
proc STD_Scan(u)
   foreach v ∈ Succ(u) do
      if Owner(v) = α
         then STD_Update(v, u, d(u) + l(⟨u, v⟩));
          else send_msg(Owner(v), "STD_Update(v, u, d(u) + l(⟨u, v⟩))"); fi
   od
```

```
proc Update(v, u, t)
   if d(v) > t
      then d(v) := t; p(v) := u;
            if d(u) < threshold then "Negative cycle found"; terminate; fi
            Std(v, u, l(v, u));
            if v ∉ Q^α then push(Q^α, v); fi
   fi
```

```
proc Std(v, p, l)
  if p(v) ≠ p then return; fi
  Local Q1; push(Q1, (v, l));
  while Q1 not empty do
        (v1, l1) := pop(Q1);
        if (v1 = a) ∧ (l1 < 0) then "Negative cycle found"; terminate; fi
        foreach u ∈ Succ(v1) ∧ p(u) = v1 do
          if Owner(u) = α
            then push(Q1, (u, l1 + l(⟨v1, u⟩))); p(u) := deleted;
                 if u ∈ Q^α then remove(Q^α, u); fi
            else send_msg(Owner(u), "Std(u, p, l1 + l(⟨v1, u⟩))"); fi
        od od
```

Network Simplex

```
proc STT_Scan(u)
  foreach v ∈ Succ(u) do
    if Owner(v) = α
      then STT_Update(v, u, l(⟨u, v⟩), d(u));
      else send_msg(Owner(v), "STT_Update(v, u, l(⟨u, v⟩), d(u))"); fi
  od
proc Update(v, u, luv, t)
  if d(v) > t + luv
    then if p(v) = nil then d(v) := t + luv; p(v) := u;
                       else p(v) := u; Pivot(v, u, d(v) − (t + luv), luv); fi fi
  if p(v) = u then push(Q^α, v); fi
proc Pivot(v, u, t, luv)
  LocalQ1; push(Q1, (v, luv));
  while Q1 not empty do
        (v1, l1) := pop(Q1);
        if (u = v1) ∧ (l1 < 0) then "Negative cycle found"; terminate; fi
        if (u = v1) then continue; fi
        d(v1) := d(v1) − t;
        foreach u1 ∈ Succ(v1) ∧ p(u1) = v1 do
          if Owner(u1) = α
            then push(Q1, (u1, l1 + l(⟨v1, u1⟩)));
            else send_msg(Owner(u1), "Pivot(u1, u, t, l1 + l(⟨v1, u1⟩))"); fi od
        if v1 ∈ Q^α then remove(Q^α, v1); fi od
```

Bellman-Ford-Moore with Walk to Root

```
proc WTR_Scan(v)
  foreach (v, u) ∈ E do
    if Owner(u) = α
      then WTR_Update(u, v, d(v) + l(v, u))
      else send_message(Owner(u), "WTR_Update(u, v, d(v) + l(v, u))") fi
  od
```

```
proc WTR_Update(u, v, t)
  if d(u) > t then if walk(u) ≠ nil
                     then if Owner(v) = α
                            then push(Q^α, v)
                            else send_message(Owner(v), "push(Q, v)") fi
                     else d(u) := t; p(u) := v;
                          if WTR_amortization then WTR([u, stamp], u);
                                                   stamp++ fi;
                          if u ∉ Q^α then push(Q^α, u) fi fi fi
```

```
proc WTR([origin, stamp], at)
  done := false;
  while ¬done do
        if owner(at) = α
          then if walk(at) = [origin, stamp] then "Negativecyclefound";
                                                  terminate; fi
               if (at = root) ∨ (walk(at) > [origin, stamp])
                 then if Owner(origin) = α
                        then REM([origin, stamp], origin)
                        else send_message(Owner(origin),
                             "REM([origin, stamp], origin)"); fi
                      done := true; continue; fi
               if walk(at) = [nil, nil]) ∨ (walk(at) < [origin, stamp])
                 then walk(at) := [origin, stamp];
                      at := p(at);
               fi
          else send_message(Owner(at), "WTR([origin, stamp], at)");
               done := true; fi
  od
```

```
proc REM([origin, stamp], at)
  done := false;
  while ¬done do
        if Owner(at) = α
          then if walk(at) = [origin, stamp] then walk(at) := [nil, nil];
                                                  at := p(at);
                                             else done := true fi
          else send_message(Owner(at), "REM([origin, stamp], at)");
               done := true fi
  od
```

4 Comparison of Distributed Algorithms

The challenge for distributed algorithms is to beat their (usually very efficient) static counterparts. However, their actual running time may depend on many parameters that have to do with the type of the input, the distribution of the graph, and others. Hence, it is inevitable to perform a series of experiments with several algorithms in order to be able to select the most appropriate one for a specific application.

A parallel execution is characterized by the time elapsed from the time the first processor started working to the time the last processor completed. We present the average of a filtered set of execution times. Our collection of datasets consists of a mix of real (representing verification problems) and generated instances, the instances scale up linearly with the number of processors.

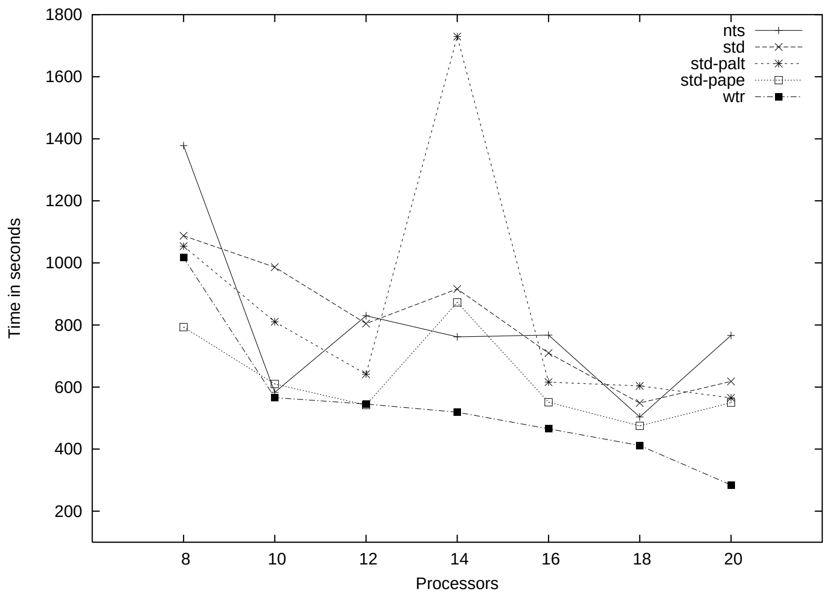

Figure 1: Real graph (model of a lift with 14 levels)

The algorithms have been implemented in C++ and the experiments have been performed on a cluster of 20 Pentium PC Linux workstations with 512 Mbytes of RAM each interconnected with a 100Mbps Ethernet and using Message Passing Interface (MPI) library.

We compared the following combinations of SSSP and NCD algorithms: *Network Simplex with Subtree Traversal* [`nts`], *Nellman-Ford with Subtree Disassembly and distance lower bound* [`std`], *Pallottino with Subtree Disassembly* [`std-palt`], *D'Escopo Pape with Subtree Disassembly* [`std-pape`], *Bellman-Ford with Walk to Root* [`wtr`].

The results are summarised in Figures 1 to 4. From the experiments we can draw some remarkable conclusions. The first and the most impor-

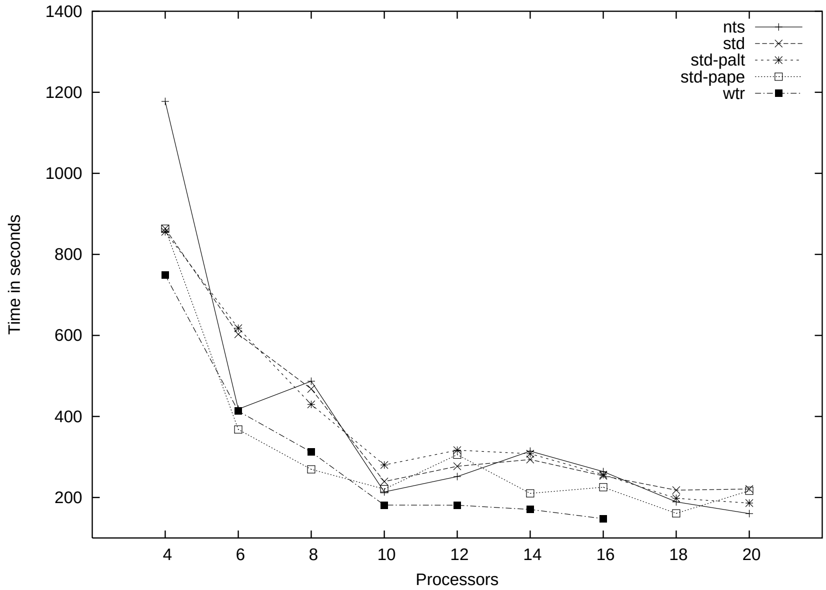

Figure 2: Real graph (lift with 12 levels, 500000 nodes, no negative cycle)

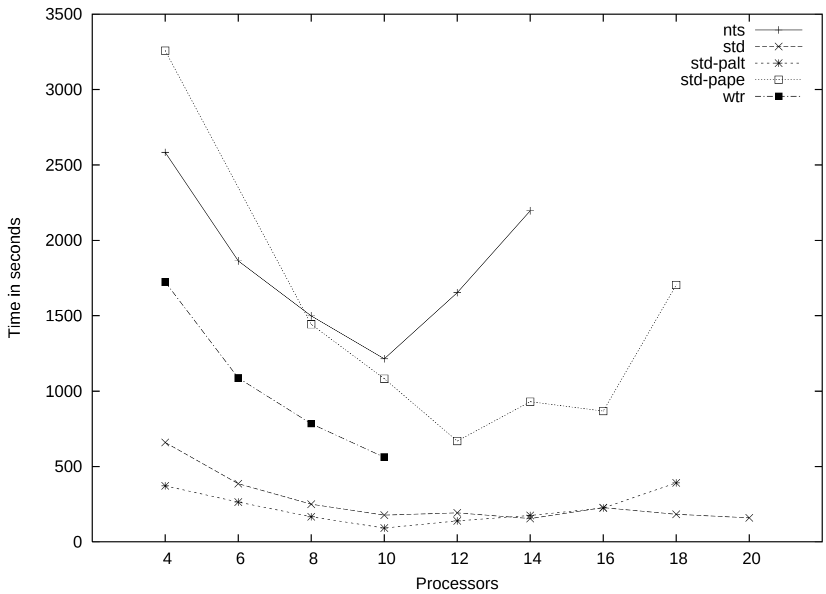

Figure 3: Generated graph (no negative cycle, 64000 nodes)

tant one is that distributing SSSP and NCD algorithms (even if done in a very straightforward manner) allows to solve these problems for huge graphs within reasonable amount of time. This is in particular important in applications like model checking – more realistic systems can be verified. All the implemented algorithms show good *scalability*, the exception is the situation which represents the worst case (the negative cycle in grid-like graph passes through all the processors) where the measurements correspond to the theoretical worst case complexity (Figure 4). Proper choice of cycle detection strategy is in the distributed environment more important than the labelling strategy. As regards cycle detection the algorithms behave differently depending on the "type" of the input graph. For example for randomly generated graphs with negative-valued edges and without cycles we could conlude that `std-palt` is the best choice and the `nts` has the worst behaviour. For generated graphs with positive-valued edges all the algorithms scale well and are reasonably fast. For graphs resulting from model-checking problems the `wtr` approach proved to beat all the others regardless of the presence or absence of a negative cycle. The experiments also demonstrated the fact that spliting the graph into too many small parts does not bring additional speedup of the computation due to increase in communication.

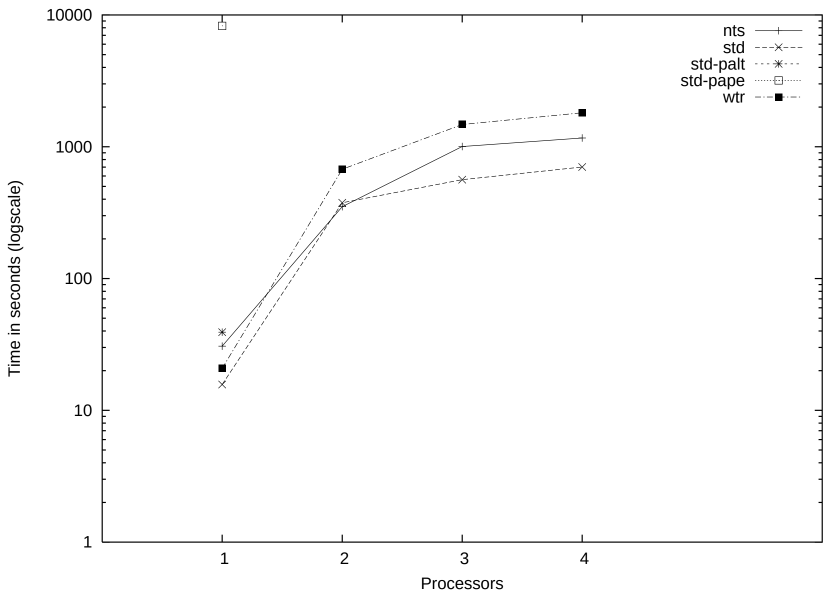

Figure 4: Generated graph – the worst case

5 Conclusions

We provide and analyse parallel algorithms for the general SSSP and NCD problems for graphs specified with adjacency lists. The algorithms are designed for networks of workstations where the input graph is distributed over individual workstations communicating via a message passing interface.

Based on our experiments we conclude that in situations where no a priori information about the graph is given the best choice is the Subtree Disassembly algorithm (in the sequential version known also as Tarjan's algorithm). For specific applications other algorithms or their combinations can be more suitable, as demonstrated by Walk to Root in case of application to model checking.

References

[1] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model mhecking based on negative nycle detection. In *FST TCS 2001*, number 2245 in LNCS. Springer-Verlag, 2001.

[2] B. V. Cherkassky and A. V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming, Springer-Verlag*, 85:277–311, 1999.

[3] U. Meyer and P. Sanders. Parallel shortest path for arbitrary graphs. In *EUROPAR: Parallel Processing, 6th International EURO-PAR Conference*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[4] K. Ramarao and S. Venkatesan. On finding and updating shortest paths distributively. *Journal of Algorithms*, 13:235–257, 1992.

[5] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *Proc. 23rd MFCS'98*, volume 1450 of *Lecture Notes in Computer Science*, pages 722–731. Springer-Verlag, 1998.

[6] M. Hribar, V. Taylor, and D. Boyce. Parallel shortest path algorithms: Identifying the factors that affect performance. Technical Report CPDC-TR-9803-015, Center for Parallel and Distributed Computeing, Norhwetern University, 1998.

[7] M. Hribar, V. Taylor, and D. Boyce. Performance study of parallel shortest path algorithms: Characteristics of good decompositions. In *Proc. ISUG '97 Conference*, 1997.

[8] J. Traff and C.D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *Parallel algorithms for irregularly structured problems (IRREGULAR-3)*, volume 1117 of *LNCS*. Springer-Verlag, 1996.

[9] D. Kavvadias, G. Pantziou, P. Spirakis, and C. Zaroliagis. Efficient sequential and parallel algorithms for the negative cycle problem. In *Proc. 5th ISAAC'94*, volume 834 of *Lecture Notes in Computer Science*, pages 270–278. Springer-Verlag, 1994.

[10] E. Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21(2):331–357, 1996.

[11] S. Chaudhuri and C. D. Zaroliagis. Shortest path queries in digraphs of small treewidth. In *Automata, Languages and Programming*, pages 244–255, 1995.

[12] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[13] L.R. Ford. Network flow theory. Rand Corp., Santa Monica, Cal., 1956.

[14] U. Pape. Implementation and efficiency of Moore-algorithms for the shortest path problem. *Mathematical Programming*, 7:212–222, 1974.

[15] S. Pallottino. Shortest-path methods: Complexity, interrelations and new propositions. *Networks*, 14:257–267, 1984.

[16] G.B. Dantzig. Application of the simplex method to a transportation problem. *Activity Analysis and Production and Allocation*, 1951.

[17] A. V. Goldberg and T. Radzik. A heuristic improvement of the Bellman-Ford algorithm. *AMLETS: Applied Mathematics Letters*, 6:3–6, 1993.

[18] D. Goldfarb, J. Hao, and S.R. Kai. Shortest path algorithms using dynamic breadth-first search. *Networks*, pages 29 – 50, 1991.

Publications in the FI MU Report Series are in general accessible via WWW and anonymous FTP:

`http://www.fi.muni.cz/informatics/reports/`
`ftp ftp.fi.muni.cz (cd pub/reports)`

Copies may be also obtained by contacting:

Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic