Experiments with the auction algorithm for the shortest path problem.

Jesper Larsen* Ib Pedersen†

February 3, 1997

Abstract

The auction approach for the shortest path problem as introduced by Bertsekas is tested experimentally. Parallel algorithms using the auction approach are developed and tested. Both the sequential and parallel auction algorithms perform significantly worse than a state-of-the-art Dijkstra-like reference algorithm.

1 Introduction

The shortest path problem is one of the classical problems in Operations Research. One usually classifies shortest path algorithms into one of two groups: the *label-setting* algorithms (Dijkstra-like) and the *label-correcting* algorithms (Bellman-Ford-like).

A recent approach to solving shortest path problems is the auction algorithm proposed by Bertsekas in [Ber91]. In [PS91] and [BPS92] the performance of the auction algorithm is enhanced by the use of graph reduction, thereby reducing the worst-case time-complexity from pseudo-polynomial to strongly polynomial.

Here we introduce the *improved graph reduction* scheme, which allows for additional reduction of the graph. Furthermore, suggestions in [Ber91] on how to parallelize the auction approach are investigated and used as off-set for constructing other parallel algorithms.

In section 2 the sequential auction algorithm is presented. In section 3 the auction algorithm is enhanced by the introduction of graph reduction. Section 4 describes parallel algorithms based on the auction approach, and experimental results are given in section 5. Finally the most important findings are summarised in the conclusion.

*Department of Computer Science (DIKU), University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, e-mail: friberg@diku.dk

†Dansk Data Elektronik A/S, Herlev Hovedgade 199, DK-2730 Herlev, e-mail: ibp@dde.dk

2 The auction algorithm

Let $\mathcal{G} = (\mathcal{N}, \mathcal{A}, l)$ be a weighted directed graph with node-set $\mathcal{N}$, arc-set $\mathcal{A}$ and length-function $l : \mathcal{A} \to \mathbb{R}$. We assume that there is at most one arc from node i to node j, denoted (i,j). Additionally we assume that all arc-lengths are non-negative. n denotes the number of nodes and m denotes the number of arcs in the graph.

The auction algorithm was first presented by Bertsekas in [Ber91] as a single-source single-destination algorithm but it can easily be extended to a single-source all-destination algorithm. Initially we assume that all cycles have positive length, and to simplify descriptions we assume that each node except the destination has at least one outgoing arc.

Let s be the source node and t the destination node. During the execution of the auction algorithm a path called the *candidate path* is maintained. The first node of the candidate path is always the source node s. The last node of the candidate path is called the *terminal node*. A node that is or has been part of the candidate path is called a *tree node*. A *border node* is a node that has an incoming arc originating from a tree node.

The auction algorithm consists of a (finite) number of iterations. In each iteration one of two operations, *contraction* or *extension*, is performed. The algorithm terminates when t becomes the terminal node.

The selection of operation is determined by the *price* of the terminal node. For each node i we maintain a price π_i such that

$$\pi_i \leq l_{ij} + \pi_j \text{ for all } (i,j) \in \mathcal{A} \quad (1)$$

$$\pi_i = l_{ij} + \pi_j \text{ for all } (i,j) \text{ where } P = \langle \ldots, i, j, \ldots \rangle \quad (2)$$

where P is the candidate path. π denotes the vector of prices.

Assume that the candidate path P is $\langle s, i_1, i_2, \ldots, i_k \rangle$ and there exists an arc (i_k, i) with

$$\pi_{i_k} = l_{i_k i} + \pi_i \quad (3)$$

We perform an *extension of* P by i by extending the candidate path P with node i resulting in a new candidate path $P = \langle s, i_1, i_2, \ldots, i_k, i \rangle$. The arc (i_k, i) is called a *candidate arc*. There may exist more than one candidate arc, and in that case a random one is selected to extend by.

If, however, no arc leaving the terminal node satisfies (3) a *contraction* is made. A contraction consists of two steps:

- first update the price vector, and then
- the terminal node is discarded from the candidate path – the candidate path P is reduced to $\langle s, i_1, i_2, \ldots, i_{k-1} \rangle$. However this is only done if P is not equal to the degenerate candidate path $\langle s \rangle$.

A pair (P, π) is said to satisfy *Complementary Slackness* (or CS for short) if:

1. P is a simple path.

2. π is a price vector that satisfies (1) and (2).

Initially (P, π) is required to satisfy *CS*. If all arc-lengths are non-negative this can easily be achieved by setting

$$P = \langle s \rangle \text{ and } \pi_i = 0 \text{ for all } i \in \mathcal{N}.$$

Otherwise a preprocessing algorithm (described in [Ber91]) can be run to initialize the variables.

An iteration starts by testing the inequality

$$\pi_i < \min_{(i,j) \in \mathcal{A}} \{l_{ij} + \pi_j\}$$

for the terminal node of P. If it does *not* hold then due to the CS conditions the corresponding equality holds and an extension using one of the candidate arcs is made. Otherwise a contraction is performed raising the price of the terminal node of P to $\min_{(i,j) \in \mathcal{A}} \{l_{ij} + \pi_j\}$, and discarding it from the candidate path (unless $P = \langle s \rangle$). The CS condition clearly remains valid.

The algorithm is shown in Figure 1. Note that by (2) $\pi_s - \pi_i$ is the length of the part of the candidate path P between s and i, and by (1) the length of every path from s to i is at least equal to $\pi_s - \pi_i$. So if a pair (P, π) satisfies CS, the part of P between s and any node $i \in P$ is a shortest path from s to i, and $\pi_s - \pi_i$ is the corresponding shortest distance.

Termination and correctness of the algorithm is shown in [Ber91, LP95]. The auction algorithm has a worst-case time-complexity $O(kn^2)$, where k is the length of the longest shortest path. Hence if an upper bound on the arc-lengths is imposed, the algorithm becomes strongly polynomial.

If a node i without outgoing arcs exists, we may set $\pi_i = +\infty$, when calculating $\min_{(i,j) \in \mathcal{A}} \{l_{ij} + \pi_j\}$. It can be viewed as the existence of an (i, t)-arc with length $+\infty$.

The extension from a single-destination algorithm to an all-destinations algorithm can easily be accomplished by not terminating the algorithm before all nodes have been terminal node.

An improvement to the auction algorithm would be to calculate

$$j_i = \arg \min_{(i,j) \in \mathcal{A}} \{l_{ij} + \pi_j\}$$

while calculating the minimum. Saving the value, we may next time node i becomes terminal node start by checking

$$\pi_i = l_{ij_i} + \pi_{j_i}.$$

If the equation holds an extension can be performed without calculating the minimum. This is called the *best neighbor* improvement. Note that it does not change the theoretical time complexity, but it has a significant effect when implementing the algorithm.

```
P = ⟨s⟩
π set so CS is maintained
≪ let i be the terminal node of P ≫
i = s
while i ≠ t do
    if π_i < min_{(i,j)∈A}{l_ij + π_j} then
        ≪ Contraction ≫
        π_i = min_{(i,j)∈A}{l_ij + π_j}
        if i ≠ s then
            P = P − ⟨i⟩
            i = terminal node of P
    else
        ≪ Extension ≫
        j_i = arg min_{(i,j)∈A}{l_ij + π_j}
        P = P + ⟨j_i⟩
        i = j_i
return P
```

Figure 1: The single-destination auction algorithm.

3 Graph reduction

In an effort to decrease the time complexity of the algorithm, arcs that can not be a member of the solution, can be removed by reduction. Three types of reduction denoted *simple* (described in [PS91]), *extended* (described in [BPS92]) and *improved* reduction (described in [LP95]) are described. All reductions result in algorithms better than the classic auction algorithm both wrt. empirical and worst-case time complexity. The latter is improved from pseudo-polynomial to strongly polynomial.

When running the auction algorithm as an all-destinations algorithm, all three types of reductions delete all arcs not belonging to a shortest path tree. Upon termination the graph has "collapsed" into a shortest path tree. The difference between the three types is how early the arcs are deleted during the execution. An arc may be deleted earlier by improved reduction than extended reduction, which may delete the arc earlier than simple reduction.

In *simple* graph reduction, the first time a node becomes terminal node the shortest path from the source to this node is found as described earlier. No path from the source ending in this node is shorter. So all arcs going into the node except the arc used to extend to the node can be removed. The time complexity now becomes $O(m^2)$ as proved in [PS91].

In the *extended* graph reduction not only incoming arcs are deleted but also outgoing arcs. In order to do so a new variable u_j is introduced for each node j. For

node j, u_j is an upper bound on the length of the shortest path from s to j. Theses variables corresponds exactly to the temporary labels on the nodes generated by the Dijkstra algorithm. Therefore one sometimes refer to the variables u_j as *Dijkstra labels*. During the execution of the algorithm u_j is monotonically non-increasing, and as j becomes terminal node it equals the length of the shortest path from s to j. Initially we set

$$u_j = \begin{cases} 0 & \text{if } j = s \\ +\infty & \text{if } j \neq s \end{cases}$$

Like simple reduction extended reduction is only performed at first scan, where a scan is the calculation of the minimum of $l_{ij} + \pi_j$ (we say we "scan" the arcs).

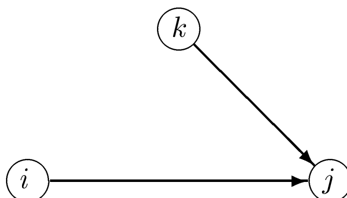

Figure 2: Node i has just become terminal node for the first time.

Consider the situation in Figure 2. Node i has just become the terminal node for the first time. If $u_j \leq u_i + l_{ij}$ the arc (i, j) can be deleted once the value of u_j indicates that another path has a smaller upper bound, thereby excluding i as intermediate node in the shortest path to j.

Otherwise $u_j > u_i + l_{ij}$ a new better upper bound using i as intermediate node is found. The Dijkstra label u_j is therefore updated to $u_i + l_{ij}$ and the arc (k, j), where $u_k + l_{kj}$ was equal to the former value of u_j, is deleted. This arc is easy to determine as it is the only arc going into j coming from a tree node.

In [BPS92] it is shown that the auction algorithm with extended reduction has a worst-case time complexity of $O(n \min\{m, n \log n\})$.

In an effort to delete even more arcs we developed the *improved* reduction. The theoretical worst-case time complexity is worse than extended reduction but remains strongly polynomial as we get $O(nm)$. The proof can be seen in [LP95].

Consider again node j in Figure 2. If we now calculate

$$\arg \min_{(k,j) \in \mathcal{A}} \{u_k + l_{kj}\} \tag{4}$$

we can use it to check whether the arc (i, j) can be discarded or not. If (4) is different from i another path has an *upper bound* better than the path using i. Hence the other path is better (or at least not worse) and, as it is only an upper bound, it may get even better. Notice that beside checking arcs outgoing from tree nodes we now

also check arcs outgoing from border nodes, which was not the case in the extended reduction.

The fact that all border nodes at most has one ingoing border arc enables us to make the check in constant time in the extended reduction, while all ingoing arcs has to be scanned in the improved reduction. This accounts for the difference in worst-case time complexity.

4 Parallel Algorithms

Before discussing the parallel algorithms developed we will describe the parallel computer used in our experiments.

The MEIKO parallel computer at the Department of Computer Science, University of Copenhagen (DIKU) is an asynchronous MIMD (Multiple Instruction-stream Multiple Data-stream) parallel computer equipped with 16 Intel i860 processors each having 16Mb of memory. The communication between the 16 i860's is controlled by 32 T800 communication transputers. The MEIKO is capable of both *synchronous* and *asynchronous* communication.

When communicating asynchronously sender and receiver must regularly test whether the communication has been completed. In practice the receiver has to test if any data has arrived in the prepared buffers.

As with most of the MIMD-class parallel computers the startup latency (the time used to setup communication) is very high compared to the time it takes to send information (e.g. an integer) and the time to do an integer operation (e.g. add two numbers).

4.1 The Parallel Reverse Algorithms

It should be quite obvious that in the single-source single-destination approach we may "reverse the calculations" building the path from the destination and back to the source instead of building the path from the source to the destination. This results in the *reverse algorithm* shown in Figure 3.

A way to parallelise the auction algorithm is by viewing the single-source all-destination problem as $n-1$ single-source single-destination problems. Imagine that we had an unlimited polynomial number of processors at our disposal. We could then run the reverse algorithm on $n-1$ processors each with the same source but a different destination.

This would give us a "naive" parallel algorithm where the results gained along the computation (intermediate shortest paths) are not shared among the processors.

In an effort to utilise the intermediate results we consider the price vectors. As each reverse algorithm is run independently each processor works on its own price vector. To share some of the information in the price vectors we observe that if two

```
R = ⟨t⟩
≪ π set so that CS is maintained ≫
π_i = 0  ∀i ∈ 𝒩
≪ let j be the terminal node of R ≫
j = t
while j ≠ s do
    if π_j > max_{(i,j)∈𝒜}{π_i − l_ij} then
        ≪ Contraction ≫
        π_j = max_{(i,j)∈𝒜}{π_i − l_ij}
        if j ≠ t then
            R = R − ⟨j⟩
            j = terminal node of R
    else
        ≪ Extension ≫
        i_j = arg max_{(i,j)∈𝒜}{π_i − l_ij}
        R = R + ⟨i_j⟩
        j = i_j
return P
```

Figure 3: The reverse algorithm.

processors P^i and P^j each maintain their price vectors π^i and π^j respectively then

$$\pi_p = \min\{\pi_p^i, \pi_p^j\} \qquad \text{for } p = 1, \ldots, n \tag{5}$$

will again satisfy CS (as proved in [Ber91, LP95]). So the price vector π defined by (5) is a valid price vector. Combining the price vectors of some (or all) of the processors tends to speed up the termination of the algorithm.

Note that even though CS is maintained for the new price vector we may have destroyed the equality within the candidate path. So we must traverse the candidate path from the destination and stop when the equality is not satisfied.

A more severe problem is that graph reduction may not be used when we exchange prices. An arc not present in processor P^i's representation of the graph may lead to a violation of the CS on P^j where the arc may still be present when we use the new "common" price vector. Due to the substantial effect of graph reduction this is a major drawback wrt. running times.

Communication between the processors is quite expensive on the MEIKO (as on most MIMD machines). Hence, the communication scheme must be kept as simple as possible. It is far too expensive to exchange π-values every time these are changed. It is also too expensive to communicate values from one processor directly to all other processors (broadcasting).

Instead we setup a different topology. We have chosen a ring. This gives a simple communication protocol, and we have only a small amount of processors at our disposal. Each processor sends its values only to one specific processor and receives values only from a specific processor as depicted in Figure 4.

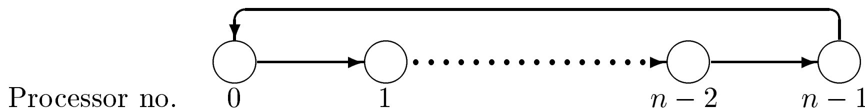

Figure 4: Communicating using a ring with n processors. The arrows indicate the direction of communication.

In addition to the simple structure we use asynchronous communication which is much faster than synchronous communication. Asynchronous communication is, however, more difficult to implement as we have to maintain communication buffers. In order to minimize the administrational overhead we implement our ring structure with only one processor being allowed to send at a time. This approach might although not be optimal on systems with more processors as the longest path a message has to travel before the information on every processor is updated grows linearly to the number of processors available.

The interesting prices are those that have changed since the last communication. Rather than sending all prices when communicating, it is enough to send updated prices. It does, however, seem that the overhead introduced by packing and unpacking the updated prices is too expensive. Therefor the whole price vector is passed along.

The prices will be communicated after a fixed number of changes. After testing different values, we have found that $200000/p$, where p is the number of processors used, gave the best result. The reason for dividing by p, is that with more processors, more information can be used by other processors, and the communication frequency should increase.

The set of destinations are partitioned equally among the processors. Each processor can solve the problems in two ways:

- Once we start working on a destination we only proceed to another destination when the shortest path is found.
- We run several iterations on one destination, then proceed to a new one, even if we did not find the shortest path. We then later return to the destinations that are not solved, run a number of iterations and so forth.

Using the first approach only one price vector is needed. Indeed as the CS is maintained after the first destination is solved we do not have to reset it, but may instead use the information when running the remaining destinations. Memory

allocation for one candidate path is also enough if the shortest path tree is saved in another way, e.g. by pointers to the predecessors. Note that when a destination reaches a node that is already in the shortest path tree the reverse algorithm can proceed to the next destination since the source now can be reached by a series of extensions from the terminal node. When solving one destination at a time the algorithm becomes sensitive to the order in which the destinations are processed. It would be best to solve the destinations in order of "closeness" to the source, starting with the nearest destinations. In this way the closest destination helps building up a path towards the destinations farthest away.

We can minimize the communication by communicating the nodes of the shortest path tree built so far instead of π-values. Each time a destination reaches the source, the nodes in the candidate path and their predecessors and the prices are added to a queue. At some fixed interval the queue is emptied sending the information to the next processor. By communicating the prices we had to communicate $O(n)$ elements in each communication, now the total amount of elements to communicate throughout the execution is $O(n)$. On the other hand the information is weaker since only the prices for tree nodes are communicated.

In order to exploit the information gathered, both prices and predecessors could be sent. All the three described ideas have been tested.

When the price vector is communicated a small series of tests showed that communication after $200000/p$ alternate in the prices (p is the number of processors) is optimal. When only the predecessors are communicated the tests showed that communication should be performed more frequently, namely after $50000/p$ changes in prices.

Our second idea is to alternate between the destinations. Ideally we would like to be able to have only one candidate path and only one price vector. If we alternate from one destination to another only when the candidate path has "collapsed" into the degenerate candidate path $\langle d_i \rangle$, only one candidate path is present at any point (because the first node of each candidate path can be implicitly represented, as it is the destination node). When a new node starts from this degenerate candidate path the price vector can not be violated by any of the operations, that is, we have to keep track of only one candidate path and one price vector.

Regarding the time complexity, the worst case is that each of destinations run independently, i.e. the nodes searched by each of the destinations are only reached by one of the destinations. Therefore the worst-case time complexity running ρ destinations on each processor will in the worst case be ρ times the time complexity for running the algorithm with only one destination.

When there are $p > 1$ processors and δ destinations, each processor solves the problem for $\rho = \frac{\delta}{p}$ destinations. The destinations are distributed using the communication ring. The first processor keeps the first $\frac{\delta}{p}$ destinations and sends the rest to the next processor, which keeps its amount of destinations, and so on. Once a processor has sent the rest of the destinations to the next processor it can start the

multiple destinations algorithm shown in Figure 5.

```
for i = 1 to δ
    found(i) = FALSE
while ⟨not all found⟩
    for i = 1 to δ
        if not found(i)
            ⟨run reverse for destination i until the candidate path only
             contains the destination, or the source is reached ⟩
            if ⟨ source reached ⟩
                found(i) = TRUE
```

Figure 5: Solving multiple destinations (interchangeably) on one processor.

As previously the communication is performed after $200000/p$ changes in the prices (p is the number of processors).

4.2 The Parallel Two-sided Algorithm

In [Ber91] Bertsekas suggests an algorithm for the single-source single-destination problem where we use both the auction algorithm and the reverse algorithm. Since this algorithm will be used in the following we will briefly describe it here.

We now consider combining the auction algorithm (hereafter also called the forward algorithm) with the reverse algorithm. Initially we have two candidate paths P (starting at the source) and R (starting at the destination) and one price vector π. We then run the forward algorithm using P and the reverse algorithm using R as candidate paths. The algorithm (called the *two-sided* auction algorithm) terminates when the candidate paths contain a common node. Since both P and R satisfies CS throughout the algorithm the composite path of P and R will be a shortest path from the source s to the destination t.

Correctness and termination of the two-sided auction algorithm can only be proved under the assumption that the potentials and lengths are integers as is done in [Ber91, LP95]. Without the requirement of π_s and π_t being increased respectively decreased at least once in each step it is possible to construct examples where the two-sided algorithm never terminates (see [LP95]).

The MEIKO parallel computer has no shared memory. This creates a problem since when we parallelise the two-sided auction algorithm the two candidate paths *must* operate on the *same* price vector. If we are to parallelise this algorithm we must be able to maintain the common price vector. Here we may use some of the ideas described earlier, namely to communicate prices and predecessors for nodes in the shortest path tree found so far.

An example of the execution is shown in Figure 7. When the forward algorithm

```
do
    ≪ Step 1 ≫
    Run several iterations of the while-loop of the forward algorithm,
    at least once increasing π_s
    if P and R contains a common node then exit
    ≪ Step 2 ≫
    Run several iterations of the while-loop of the reverse algorithm,
    at least once decreasing π_t
while P and R does not contain a common node
```

Figure 6: The sequential two-sided auction algorithm.

scans the node i for the first time, it sets u_i^s to the shortest distance from the source s to i. If π_i^t is set to $-u_i^s$ in the reverse algorithm, an extension to i in the reverse algorithm means that a shortest path from t via i to s has been found. If the reverse algorithm furthermore receives the predecessors from the forward processor, the shortest path is known.

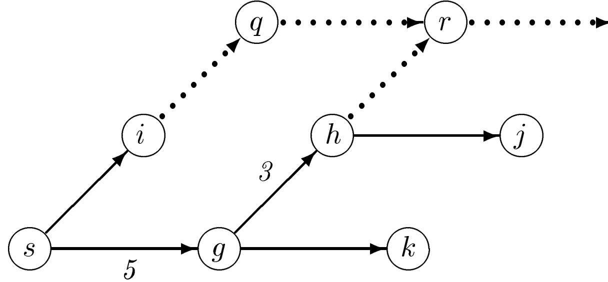

Figure 7: The solid arcs are arcs in the shortest path tree found so far by the forward algorithm. The dotted arcs are some of the other arcs. As the shortest path from s to h is found, we set $\pi_h^t = -u_h^s = -8$. When the reverse algorithm can make an extension from r to h, it means that it can extend further on to g and s, as the CS equality holds for these nodes. Therefore is the shortest path found from the destination to the source when the reverse algorithm reaches h.

It should be noted that the CS condition may be violated if a node in the candidate path of the reverse algorithm is updated by the forward algorithm. If we mark the nodes for which we have received information from the forward algorithm and vice versa the problem is solved.

A simple method to communicate is as follows: Each time the forward processor expands its shortest path tree by a node (i.e. first time node i becomes terminal node) it communicates $-u_i^s$ to the other processor, which sets $\pi_i^t = -u_i^s$. Similar

action is performed by the reverse algorithm. Even as asynchronous communication is used on the MEIKO this may be an expensive communication strategy because of:

- The large startup latency.
- In the first iterations of the algorithm, there will be many first scans. Later on, more iterations will be performed before a first scan is encountered.
- Running the parallel program on a "big" problem the source and the destination are likely to be far apart. Communicating the u-values before a certain number of first scans is hence a waste of time.

To minimize communication, the nodes are placed in a queue where they are scanned for the first time (i.e. becomes tree nodes). When the queue reaches a certain size (denoted *the critical size*), all the nodes in the queue are sent to the other processor, and the queue becomes empty. The size of the queue before it is sent should decrease as the algorithm iterates. There are two reasons for this:

- During the execution of the parallel program, the trees are closing in on each other. So the chances of a possible common node is increasing, which should be reflected by intensifying the communication.
- In the beginning of the algorithm first scans are performed more often. Hence the queue grows faster in the beginning of the algorithm.

After each communication the critical size is halved. To determine the initial communication threshold, we ran some tests, where the threshold was set to different percentage values of the number of nodes. The best result was obtained with the initial threshold set to 1%.

4.3 The Parallel Combined Algorithm

Like we solved the single-source single-destination problem with the two-sided auction algorithm, the same effect for the single-source all-destinations problem could be obtained by running $n-1$ combined algorithms each using 2 processors. It would, however, be a waste of processors as the source is the same for all problems. Instead, we propose to use one processor to run the forward algorithm and the rest to run reverse algorithms, virtually combining the two previous ideas (hence we call it the combined algorithm).

Let us initially assume that we have enough processors at our disposal. A simple parallel scheme would then be to run the forward algorithm on one processor (from now on called the forward processor) and $n-1$ reverse algorithms each on their own processor (called the reverse processors). Each of the reverse processors communicates with the forward processor making the forward processor virtually acting as $n-1$ forward processors.

We want to keep the communication as simple as possible, which can be achieved by ensuring that at most one message is "in transit" between the forward processor and any given reverse processor at a time. The forward processor will therefore communicate only with those reverse processors that have acknowledged the communication. Also, only the reverse processors detect when the paths have met. Hence we do not have to communicate the shortest path trees of all the reverse processors to the forward processor.

By sending to a reverse processor only when it has acknowledged the last message, we may get into a situation in which we want to send some new tree nodes to a reverse processor that has not yet acknowledged the previous message. This problem may be solved by having a separate queue for each reverse processor. It does, however, seem to be a waste of memory as many of the elements would be identical, and it would require updating. Instead we use a queue, in which we do not physically erase the nodes when they have been sent. The queue for processor i, which we will refer to as the *sub-queue for i*, is defined as the elements from *first*(i) pointer and up to but not including the *last* pointer. The *last* pointer points to the first free space in the queue, and it is equal for all sub-queues (therefore no index is required) because when we send the tree nodes we "locally" empty the sub-queue. So if *first*(i) = *last* the sub-queue for i is empty.

When the threshold is exceeded the elements in the sub-queues are sent to the particular processors and *first*(i) is set to *last*. This is only done if the processor has acknowledged the last message, otherwise the sub-queue is left unchanged and no communication is performed.

The forward processor terminates when it has received the solutions from all the reverse processors.

If we do not have enough processors at our disposal to run one destination per processor we can use a technique similar to that described earlier. The set of destinations is partitioned evenly among the available processors.

Our parallel program now virtually runs $n-1$ two-sided auction algorithms but at the expense of only one forward processor. The reverse processors are nevertheless all working independently of each other. We may therefore improve our parallel algorithm by sharing prices.

In the simple implementation the forward processor communicates directly with all reverse processors.

When a reverse processor (r_1) receives a price $-u_i^s$ from forward, it updates $\pi_i^{r_1}$, and it knows that when it reaches the node i it should terminate. Suppose another reverse processor (r_2) has not yet received the price $-u_i^s$ from forward, and it receives the updated prices from r_1. r_2 now has the updated $\pi_i^{r_2}$ values, but it does not know that i must terminate when this node is reached. A situation where the CS is violated can therefore occur.

When communicating the u_i^s-values from the forward processor to the reverse processors, we must ensure that all reverse processors receive the same information at the same time. Using synchronous communication will, however, slow down the

forward processor, especially when using many reverse processors.

Another solution is to let the forward processor communicate only with one reverse processor (a "master"), who then distributes the information received from the forward processor to the other reverse processors through the ring. This also minimize the work load on the forward processor. The idea is depicted in Figure 8 and works as follows: Forward sends its tree nodes to one reverse processor (r_1), who acknowledges the received data. The next time r_1 sends data to the next reverse processor (r_2), it furthermore sends information on which nodes now have become tree nodes in the forward processor. When a reverse processor receives this information, it updates the local information and passes it on to the next reverse processor (except for the last processor r_3).

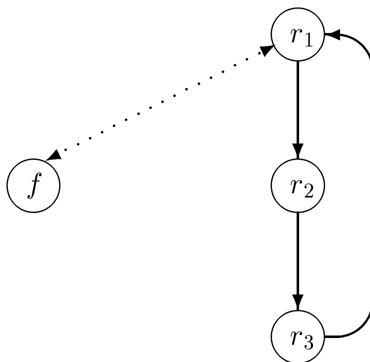

Figure 8: An example of the communication with 1 forward processor (f) and 3 reverse processors (r_1, r_2 and r_3).

Earlier we discussed what can be sent between the reverse processors. In the current implementation we only implement the communication of all the prices, along with the nodes in the forward tree. We have also tested the program to find the best communication frequency in the ring. Among a number of different values 100000/#(reverse processors) updates gave the best result. The forward processor sends information to the "master" reverse processor after first scans of 1 % of the nodes. As with the two-sided auction algorithm this threshold is halved after each communication.

5 Experimental Results

In our evaluation of the algorithms we have used the generator-programs `Spgrid`, `Sprand` and `Spacyc` made by Cherkassky et al. in connection with [CGR93] (available in the SPLIB package). For each experiment the running time is the average over 5 runs. All running times are given in seconds. If at least one instance for a problem size was not solved after at least 1000 sec. the calculation was terminated (indicated

in the tables by a 'T'), because early results indicated that times above 1000 sec. already were magnitudes worse than the best. In some of the tests of the parallel algorithms we have, however, allowed running times to exceed 1000 sec. because solving the same problem with more processes the running time came below 1000 sec. making it possible to comment on a wider range of results. An entry 'M' in the tables means that the test could not be run due to lack of memory. As the auction algorithm without reduction can not handle graphs with cycles of length zero all the graphs have been tested for such cycles.

Some of the graphs generated may have multiple arcs. Since our algorithms are constructed under the assumption that there are no multiple arcs we have developed a program `cleanup` that deletes multiple arcs. Only the shortest of the multiple arcs between two nodes remains in the graph. Thereby the solution is not changed with respect to the original graph. This only decreased the number of arcs slightly, thereby maintaining the desired characteristics e.g. a dense graph remains dense.

The sequential and parallel algorithms were evaluated by running the (to our knowledge) best state-of-the-art shortest path algorithm, namely the *Dijkstra algorithm with double buckets* (for a description see [CGR93]). The program (hereafter referred to as DIKBD) was provided by the authors of [CGR93] in the SPLIB program package. The generator-program `Spgrid` generates rectangular grid networks. Grid graphs are interesting as they are very sparse and resembles city maps and other transit networks. Nodes of these graphs correspond to points in the plane with positive integer coordinates (x, y), $1 \leq x \leq x_{\max}$, $1 \leq y \leq y_{\max}$. Node (x, y) has an outgoing arc to $(x+1, y)$ unless $x = x_{\max}$. These arcs will be called *level arcs*. For each "layer" of nodes (that is, for x fixed) there is a two-way cycle of arcs, that is, for the node (x, y) there exist outgoing arcs from and ingoing arcs to the nodes $(x, (y-1) \bmod y_{\max})$ and $(x, (y+1) \bmod y_{\max})$. These arcs are denoted *layer arcs*. Additionally a source is connected to the grid by arcs from the source to all nodes in the first layer. The lengths of the arcs are selected uniformly at random from intervals to be specified later.

We experiment with two types of grid graphs. First we have made *wide* grids (denoted GRIDW). Here $y_{\max}$ is fixed at 16, while $x_{\max}$ take on the values 128, 256, 512 and 1024. For these experiments all arc lengths are selected at random from the interval $[L, U] = [0, 10000]$.

Secondly we have used the *hard* grid networks as they are called in [CGR93]. Here $y_{\max}$ is fixed at 16, while $x_{\max}$ takes on the values 32, 64, 128 and 256. In the hard graphs (referred to as GRIDH) each layer is only a simple cycle (arcs in this cycle is of length 1) *plus* a collection of arcs connecting randomly selected pairs of nodes on the cycle (their length is choosen at random from the interval $[0; 100]$). Also arcs from lower to higher numbered layers are added for node (x_1, y) outgoing arcs are made to (x_2, y) where $x_2 = x_1 + 1 + 5i$ $(i = 0, 1, \ldots, 4)$. If we make an arc form layer x_1 to layer x_2 the randomly selected length (selected from the same interval $[L, U] = [1000, 10000]$) is multiplied by $(x_2 - x_1)^2$.

The random graphs were constructed using the generator `Sprand`. The random

graphs are constructed by making a Hamiltonian cycle and then adding arcs with distinct random end points. In our experiments we set the length of the arcs on the Hamiltonian cycle to 1, and except for the RANDL experiment the length of the other arcs are chosen at random from the interval $[0, 10000]$.

Our first test is the RAND4. The graphs in this family have $m = 4n$, which makes the graphs sparse. Here the graphs were generated for $n = 8192$, 16384, 32768 and 65536.

The graphs in the RAND14 family have $m = \frac{n^2}{4}$. These are dense graphs and they are generated for $n = 128$, 256, 512 and 1024.

Finally, we have in the RANDL test used the random graph generator to test the algorithms for their arc length dependency. For fixed $n = 65536$ and $m = 262144$ we have used different intervals for the arcs that does not belong to the Hamiltonian cycle. These were $[L, U] = [1, 1]$, $[0, 10]$, $[0, 100]$, $[0, 10^4]$ and $[0, 10^6]$.

The last generator program `Spacyc` generates acyclic graphs. Experiments with acyclic graphs are interesting since shortest path problems on acyclic graphs appear in many applications. The nodes are numbered from 1 to n, and there is a path of arcs $(i, i+1)$, $1 \leq i < n$. Furthermore additional arcs are introduced by picking two nodes at random making an arc from the lower indexed to the higher indexed. All arc lengths are chosen at random from the interval $[L, U] = [0, 10000]$. The graphs in this experiment is constructed with $m = 16n$ for $n = 4096$, 8192, 16384 and 32768.

5.1 The sequential algorithms

We first performed the experiments with the sequential auction algorithms. We have implemented the auction algorithm with best neighbor improvement as our "basic" auction algorithm (called AUC hereafter). Afterwards we extended the AUC algorithm with extended and improved reduction (calling the resulting algorithms AUCEX and AUCIM). The results are presented in the Tables 1 and 2.

Clearly **none** of our auction algorithms have been able to obtain better running times than DIKBD. The DIKBD algorithm is at least a factor 10 better than any of the auction algorithms.

Comparing the auction algorithms there is a significant improvement in performance when graph reduction is added. Both the extended (AUCEX) and improved reduction (AUCIM) results in substantially better execution times, in some cases gaining a factor 250 – 300 (in the GRIDH tests). Among the two algorithms with graph reduction the difference is small, although often slightly in favor of AUCEX. AUXEX is better in all the RANDX test and also the acyclic graphs (ACYC), while AUCIM is the fastest algorithm on the grid graphs.

The RANDL test (see Table 2) shows that while DIKBD is almost unaffected of the range of the arc-length interval, all auction algorithms are highly sensitive to the size of the $[L, U]$ interval.

It is noteworthy that when we test different size of instances DIKBD typically doubles in execution time as the size of the instances are doubled, which complies

Graph type	Size	Algorithm			
		AUC	AUCEX	AUCIM	DIKBD
GRIDW	128	2.247	0.317	0.318	0.026
	256	7.994	0.990	0.957	0.050
	512	31.991	3.647	3.350	0.098
	1024	125.592	14.213	12.616	0.197
GRIDH	32	79.566	0.352	0.307	0.014
	64	325.531	1.464	1.258	0.030
	128	T	5.466	4.817	0.058
	256	T	23.177	20.331	0.117
ACYC	4096	5.211	1.704	1.943	0.201
	8192	10.054	3.130	3.899	0.257
	16384	22.491	6.476	8.412	0.615
	32768	52.794	14.413	18.111	1.344

Table 1: Execution times for the grids and acyclic graphs.

with a theoretical time complexity of $O(m+n\tau)$ for some constant τ. For the auction algorithms we typically observe an increase by a factor 3 or 4 in the running time as the size of the instances are doubled.

5.2 The parallel algorithms

We have developed 5 parallel algorithms. The first is the parallel reverse algorithms described in subsection 4.1. This has resulted in a parallel reverse algorithm where we solve one destination at a time and exchange only prices (called PREVO). Furthermore we tried to communicate predecessor values instead of prices, which resulted in the PREVP algorithm (as with PREVO we work on one destination at a time). Still using PREVO as basic parallel algorithm we have tried to communicate both prices and predecessor values. This is called the PREVB algorithm. Finally we developed a version of the parallel reverse algorithm where we do some iterations on every destination, changing when the candidate path degenerates. This version is called PREVI. In order to keep the number of algorithms on a reasonable level we did not try to communicate the predecessor values on this algorithm.

Finally we integrate the two-sided algorithm with the parallel reverse algorithm and get the combined algorithm. This is called PCOM. Like with PREVI we restrict the algorithm to communicate only the prices among the reverse processors.

The results on the different graph types can be seen in Tables 3 to 8 on the following pages.

Extensive tests showed that PCOM using extended reduction in the forward processor was slightly better than using improved reduction in the forward processor. We therefore only report on the PCOM algorithm where extended reduction is used in

Graph type	Size	Algorithm			
		AUC	AUCEX	AUCIM	DIKBD
RAND4	8192	5.900	4.251	4.934	0.113
	16384	12.323	8.968	10.223	0.269
	32768	26.109	18.654	21.561	0.599
	65536	54.112	37.383	43.401	1.248
RAND14	128	0.111	0.025	0.042	0.004
	256	0.308	0.070	0.133	0.014
	512	0.969	0.221	0.543	0.064
	1024	2.980	0.799	2.229	0.261
RANDL	$[1,1]$	4.319	2.698	4.045	0.854
	$[0,10]$	5.237	3.607	4.797	1.041
	$[0,100]$	8.872	6.125	7.755	1.149
	$[0,10^4]$	53.395	37.168	42.682	1.252
	$[0,10^6]$	469.609	330.605	374.597	1.236

Table 2: The "random" graphs.

the forward processor. This algorithm was therefore only tested in this configuration. All algorithms are tested for 2, 3, 4, 8 and 16 processors.

We have drawn the following main conclusions: In order to analyse the test of the 5 parallel algorithms we have counted the number of times each algorithm was the best algorithm for each processor×size-group. As the PCOM algorithm can not run using only 2 processors these results are left out. Doing so the PCOM algorithm is the best parallel algorithm in 88 out of 104 cases. It is always the best in the GRIDH (Table 4) and ACYC (Table 8) tests (PCOM can, however, not be tested on the largest instance of the ACYC test due to lack of memory). Additionally only once in the RAND4 test (see Table 5) and twice in the RANDL test (see Table 7) another algorithm was faster. In the remaining tests (GRIDW and RAND14) there is no clear winner, especially the results obtained in the RAND14 test contains no indication what so ever. Noteworthy is though that PCOM not once in the RAND14 test is the best algorithm.

Among the PREV algorithms the best ones are PREVB and PREVP which accounts for over 80% of the best running times, where PREVB quite consistently is the best in the RAND4 and ACYC tests. The worst algorithm clearly seems to be PREVI which is the best algorithm very few times.

In PCOM we use one processor to run a forward algorithm, thereby assigning more destinations to the remaining $p-1$ processors than in the PREV algorithms. In most cases it is worth using one processor to run the forward algorithm. The reasons must be the efficiency of the two-tree approach as displayed in [LP95, HKS93], and the fact that we are able to use graph reduction in the forward algorithm.The last conclusion is confirmed by the huge advantage in the GRIDH test (see Table 4).

Procs.	Algorithm	128	256	512	1024
2	PRevO	0.945	1.703	3.329	7.439
	PRevP	0.656	1.306	2.760	5.871
	PRevB	0.924	1.633	3.075	6.576
	PRevI	0.633	0.985	1.445	2.868
	PCom	0.430	0.886	1.624	3.653
3	PRevO	0.690	1.221	2.559	5.862
	PRevP	0.492	0.992	2.093	4.421
	PRevB	0.678	1.171	2.371	5.200
	PRevI	0.573	0.728	1.433	2.870
	PCom	0.457	0.693	1.221	2.784
4	PRevO	0.630	1.201	2.077	4.127
	PRevP	0.425	0.693	1.253	2.519
	PRevB	0.618	1.186	2.020	3.658
	PRevI	0.476	0.814	1.471	3.045
	PCom	0.342	0.698	1.256	2.477
8	PRevO	0.576	1.055	1.987	4.358
	PRevP	0.346	0.550	0.901	1.712
	PRevB	0.603	1.125	2.412	4.482
	PRevI	0.573	0.879	1.638	3.688
	PCom	0.337	0.678	1.267	2.744
16	PRevO	0.797	1.531	2.912	7.041
	PRevP	0.417	0.761	1.420	2.758
	PRevB	0.839	1.869	3.572	7.070
	PRevI	0.689	1.210	2.206	5.617
	PCom	0.337	0.790	1.628	3.492

Table 3: Parallel single-all. GridW.

Procs.	Algorithm	32	64	128	256
2	PRevO	79.376	352.715	1343.924	T
	PRevP	78.964	349.735	1334.090	T
	PRevB	79.073	329.736	1352.781	T
	PRevI	85.231	320.215	1129.772	T
	PCom	1.320	1.967	6.917	28.157
3	PRevO	52.896	246.924	932.918	T
	PRevP	60.258	276.114	1067.202	T
	PRevB	56.219	245.639	954.789	T
	PRevI	70.945	260.964	853.052	T
	PCom	1.527	2.294	6.792	28.218
4	PRevO	42.812	187.458	742.329	T
	PRevP	52.679	244.631	948.004	T
	PRevB	44.276	176.535	757.543	T
	PRevI	61.759	217.112	700.024	T
	PCom	1.109	2.033	6.792	28.209
8	PRevO	18.797	91.834	382.155	T
	PRevP	33.877	177.168	696.376	T
	PRevB	19.785	84.079	365.750	T
	PRevI	34.207	126.767	455.175	T
	PCom	0.677	1.882	6.762	28.158
16	PRevO	7.591	42.495	192.644	T
	PRevP	21.622	123.593	524.768	T
	PRevB	7.556	37.591	196.558	T
	PRevI	19.121	75.773	276.684	T
	PCom	0.583	1.889	6.877	28.750

Table 4: Parallel single-all. GridH.

Procs.	Algorithm	8192	16384	32768	65536
2	PRevO	9.137	18.768	36.166	83.144
	PRevP	8.933	19.205	40.662	88.320
	PRevB	7.501	13.671	26.852	58.378
	PRevI	14.235	26.900	48.986	103.820
	PCom	3.915	8.041	16.825	35.564
3	PRevO	6.927	13.616	26.370	59.067
	PRevP	8.735	18.773	40.082	86.856
	PRevB	5.844	10.648	20.611	43.472
	PRevI	10.680	21.129	36.789	76.938
	PCom	4.502	8.164	16.178	32.676
4	PRevO	5.500	10.696	20.847	47.124
	PRevP	8.461	18.502	38.913	86.361
	PRevB	4.688	8.456	16.703	37.272
	PRevI	8.693	17.073	29.303	62.540
	PCom	4.024	7.411	14.741	30.466
8	PRevO	3.695	7.416	14.957	34.012
	PRevP	7.138	17.167	36.087	85.717
	PRevB	3.643	7.135	13.842	32.052
	PRevI	5.276	12.089	22.802	51.995
	PCom	3.543	6.566	13.415	29.498
16	PRevO	3.396	7.759	14.929	34.789
	PRevP	5.896	15.515	32.494	84.908
	PRevB	3.824	8.079	17.315	36.608
	PRevI	4.855	10.682	22.375	51.062
	PCom	3.242	6.533	13.553	30.260

Table 5: Parallel single-all. Rand4.

Procs.	Algorithm	128	256	512	1024
2	PRevO	0.118	0.439	1.578	5.945
	PRevP	0.107	0.429	1.568	5.928
	PRevB	0.111	0.462	1.538	6.059
	PRevI	0.155	0.569	2.044	7.397
	PCom	0.163	0.842	2.144	7.092
3	PRevO	0.107	0.405	1.631	5.791
	PRevP	0.108	0.409	1.625	5.791
	PRevB	0.096	0.432	1.558	5.899
	PRevI	0.133	0.509	1.917	6.728
	PCom	0.153	0.609	1.996	7.062
4	PRevO	0.124	0.395	1.538	5.744
	PRevP	0.121	0.392	1.528	5.724
	PRevB	0.111	0.412	1.498	5.812
	PRevI	0.138	0.442	1.725	6.240
	PCom	0.123	0.539	1.861	6.433
8	PRevO	0.114	0.379	1.474	5.406
	PRevP	0.107	0.379	1.454	5.372
	PRevB	0.101	0.402	1.423	5.476
	PRevI	0.114	0.396	1.538	5.640
	PCom	0.121	0.456	1.541	5.543
16	PRevO	0.114	0.368	1.484	5.289
	PRevP	0.121	0.365	1.358	4.766
	PRevB	0.105	0.398	1.442	5.325
	PRevI	0.131	0.437	1.543	5.332
	PCom	0.119	0.446	1.407	4.773

Table 6: Parallel single-all. Rand14.

Procs.	Algorithm	$[1,1]$	$[0,10]$	$[0,100]$	$[0,10^4]$	$[0,10^6]$
2	PRevO	10.678	12.326	18.760	80.205	606.757
	PRevP	9.124	10.265	16.922	88.459	767.789
	PRevB	10.588	11.361	16.564	58.570	439.428
	PRevI	10.236	13.222	22.227	100.600	1250.546
	PCom	4.572	5.101	7.629	34.202	262.343
3	PRevO	9.975	11.372	15.438	57.693	438.262
	PRevP	9.004	9.863	16.439	87.183	723.618
	PRevB	10.683	10.674	14.677	44.302	329.691
	PRevI	10.329	12.195	18.475	74.340	951.963
	PCom	5.558	5.979	7.989	32.441	235.216
4	PRevO	9.956	10.273	14.000	45.089	338.822
	PRevP	9.224	10.049	16.188	86.636	732.312
	PRevB	11.639	11.900	14.150	35.950	253.527
	PRevI	9.965	12.140	16.713	60.374	672.986
	PCom	5.186	5.492	7.732	30.899	222.724
8	PRevO	11.744	11.834	14.764	32.793	179.879
	PRevP	10.723	11.405	16.992	84.266	613.481
	PRevB	14.320	14.049	17.017	30.223	135.204
	PRevI	11.954	13.186	19.061	49.758	352.138
	PCom	6.726	7.229	8.935	28.910	199.879
16	PRevO	16.716	16.801	19.781	31.926	113.177
	PRevP	13.741	14.977	19.450	81.705	487.712
	PRevB	22.441	22.254	21.465	33.050	89.312
	PRevI	16.796	16.873	22.753	48.615	221.042
	PCom	8.249	9.427	12.449	29.124	163.108

Table 7: Parallel single-all. RandL.

Procs.	Algorithm	4096	8192	16384	32768
2	PRevO	6.575	14.292	31.380	70.237
	PRevP	6.069	13.357	30.208	68.658
	PRevB	6.436	13.465	30.405	66.584
	PRevI	10.112	21.195	53.297	112.879
	PCom	2.815	3.135	6.557	M
3	PRevO	5.851	12.537	27.083	60.467
	PRevP	5.496	12.051	27.558	63.363
	PRevB	5.818	11.857	26.678	58.607
	PRevI	8.549	17.180	40.108	87.318
	PCom	4.522	6.608	9.054	M
4	PRevO	5.426	11.479	24.156	52.771
	PRevP	5.118	11.271	25.308	58.279
	PRevB	5.431	10.822	24.085	51.905
	PRevI	6.950	14.597	31.716	73.574
	PCom	3.511	5.302	8.009	M
8	PRevO	4.026	8.086	17.377	38.971
	PRevP	3.895	8.524	20.160	48.469
	PRevB	4.090	8.064	17.399	39.335
	PRevI	4.384	8.601	18.487	42.862
	PCom	2.265	3.765	6.502	M
16	PRevO	2.869	5.588	12.809	30.095
	PRevP	2.526	5.677	13.941	36.696
	PRevB	2.889	5.528	13.331	30.553
	PRevI	3.010	5.893	13.258	28.770
	PCom	1.609	3.106	6.421	M

Table 8: Parallel single-all. Acyc.

The advantage of PCOM clearly drops with increasing number of processors, e.g. in the RAND4 test the advantage drops from a factor 2.3 over PREVB in the 4 processor case to 1.6 in the 8 processor instances and further to 1.1 when we use 16 processors, which means that PCOM does not scale as well as PREVB. Generally the PREV-algorithms scales rather well.

As PCOM is the best of our parallel algorithms we have calculated the absolute speed up (defined as $\frac{T_p}{T^*}$ where T_p is the running time of the algorithm using p processors, and T^* is the fastest sequential algorithm) and efficiency (defined as speedup divided with the number of processors used) for it. The results are tabulated in Table 9. The speedup and efficiency presented is the minimum and maximum over all tests with p processors.

Procs.	S_p^{min}	$S_p^{\max}$	E_p^{min}	$E_p^{\max}$
3	$4.7 \cdot 10^{-3}$	0.16	$1.6 \cdot 10^{-3}$	$5.2 \cdot 10^{-2}$
4	$5.2 \cdot 10^{-3}$	0.17	$1.3 \cdot 10^{-3}$	$4.3 \cdot 10^{-2}$
8	$5.2 \cdot 10^{-3}$	0.14	$7.7 \cdot 10^{-4}$	$1.7 \cdot 10^{-2}$
16	$5.1 \cdot 10^{-3}$	0.12	$4.7 \cdot 10^{-4}$	$7.0 \cdot 10^{-3}$

Table 9: The speed up and efficiency for PCOM.

Clearly our parallel algorithms behaves very poorly when we compare with DIKBD. Not a single place we obtain speed up above 1, in fact, best absolute speed up is 0.17. Note also that the speedup values deteriorate as we use more processors.

In order to evaluate the relative performance of the PCOM algorithm we have calculated the *scalability*, that is $Sc_p = \frac{T_2}{T_p}$. These figures are shown in Table 10 below.

Procs.	$Sc_p^{\min}$	$Sc_p^{\max}$
3	0.4744	1.3826
4	0.5913	1.5621
8	0.7056	1.9498
16	0.5411	2.2642

Table 10: The scalability for PCOM.

These figures are not impressive but in general we almost always achieve something by adding more processors. Only for very few instances we get a scalability below 1. The reason for the low scalability values stems from the difference between the forward and the reverse algorithms. The scalability of the PREV algorithms is much better, but as they are substantially slower than the PCOM algorithm it is of no use.

6 Conclusion

The experimental results are very clear both for the sequential and the parallel algorithms.

None of the auction algorithms were even close to the execution times of the DikBd algorithm. Even though graph reduction was effective in reducing the execution time of the auction algorithm it is still significantly slower than DikBd. For us the auction approach is beautiful, but it is and will remain inferior to Dijkstra-like algorithms when implemented.

For our parallel algorithm we obtain very low speed-up values. Here the problem is two-fold: we have based our parallel algorithms on an algorithm (the auction algorithm) that behaves poorly compared to DikBd, and the efficiency of the sequential DikBd is difficult to compete with since we have to overcome the expensive communication. A major drawback is that we can not use graph reduction in our reverse processors in combination with exchange prices.

Acknowledgement

We would like to thank Jens Clausen (DIKU) for valuable comments upon the draft.

References

[Ber91] Dimitri P. Bertsekas. An auction algorithm for shortest paths. *SIAM Journal on Optimization*, 1:425 – 447, 1991.

[BPS92] Dimitri P. Bertsekas, Steffano Pallottino, and Maria Grazia Scutellá. Polynomial auction algorithms for shortest paths. Technical Report TR-16/92, Dipartimento di Informatica, University of Pisa, 1992.

[CGR93] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest path algorithms: Theory and experimental evaluation. Draft, July 1993.

[HKS93] Richard V. Helgason, Jeffrey L. Kennington, and Douglas B. Steward. The one-to-one shortest-path problem: An empirical analysis with the two-tree dijkstra algorithm. *Computational Optimization and Applications*, 1:47 – 75, 1993.

[LP95] Jesper Larsen and Ib Pedersen. The auction approach for the shortest path problem: Theory and experiments. Master's thesis, Department of Computer Science, University of Copenhagen (DIKU), 1995.

[PS91] Stefano Pallottino and Maria Grazia Scutellá. Strongly polynomial auction algorithms for shortest path. Technical Report TR-19/91, Dipartimento di Informatica, University of Pisa, 1991.