A PARALLEL ALGORITHM FOR MINIMAL COST NETWORK FLOW PROBLEMS

by

Jörg Peters

Computer Sciences Technical Report #762

April 1988

A PARALLEL ALGORITHM FOR MINIMAL COST NETWORK FLOW PROBLEMS *

by

Jörg Peters

Computer Sciences Department
University of Wisconsin - Madison
Madison, Wisconsin 53706 USA

Abstract

In this paper we explore how simplex-based algorithms can take advantage of multiprocessor capability in solving minimal cost network flow problems.

We present an implementation of such an algorithm that stresses the importance of parallelized pricing. This implementation solves all NETGEN test problems of size (5000 x 25000) in less than 50 seconds wall clock time on the Sequent Symmetry S-81 using 6 processors.

Additionally, we contrast this algorithm, based on specialized processes, with algorithms that execute a uniform code on each processor. We discuss why specialized processes perform better on the set of test problems.

* This research was supported in part by NSF grant CCR-8709952 and AFOSR grant AFOSR-86-0194

1.Introduction

This paper is centered around the question: given a class of algorithms, a set of standard test problems and a (small) number of processors - can we find a parallel algorithm, based on the characteristics of that class of algorithms, such that $j+1$ processors solve the test problems $\frac{j}{j+1}$ times faster than j processors?

In particular, we are concerned with simplex-based parallel algorithms for solving the minimal cost network flow problem :

$$(NF) \qquad \begin{aligned} &\min cx \\ &\text{s.t. } Ax = r \\ &0 \leq x \leq u, \end{aligned}$$

where c, u and $x \in R^n$, $r \in R^m$ and $A \in R^{m \times n}$ $(n \geq m)$. A is a node-arc incidence matrix.

The minimal cost network flow problem is a linear program. It is special however, in that the structure of the constraint matrix A allows for a natural, sparse representation. Using specialized data structures, simplex based algorithms remain efficient even for problems with hundreds of thousands of variables. It is this efficient *average case behavior* which we want to speed up.

In the sparse representation, the two non-zero entries in each column of the matrix A correspond to an *arc*: a '1' in row i and a '-1' in row j describe an arc from node i to node j. Thus A may be represented as a graph rather than as a full m by n matrix. Accordingly, the vector c contains the costs per unit of the flows, x, on the arcs, while the vector u contains their capacity. The right hand side element r_i, if negative (positive), represents demand (supply) at a node i. Furthermore, as we split the constraint matrix into basic (subscript B) and nonbasic (subscript N) parts ($A = A_B + A_N$), the subgraph corresponding to the basic part has the structure of a tree.

Before describing the data structures appropriate for a parallel implementation, Section 2 will give some insight into the parallelizability of the minimal cost network flow problem. Also, we define the terms "uniform parallelism" and "specialized parallelism" as a way of labelling two different classes of parallel algorithms for (NF). Implementation and comparison of these two basic approaches led to the main algorithm, which is described in Section 3. We implemented the main algorithm, NETPAR, and its variants on two versions of the SEQUENT *shared memory* multiprocessor. In Section 4 we demonstrate the performance of our code on a set of test data generated by a standard test problem generator, NETGEN [KNS74]. The problems range from 12,500 to 250,000 variables and from 1,000 to 50,000 constraint equations. We also present a comparison between the performance of our code and a standard sequential code. Our goal was to reduce the *total execution time* of the algorithm. Thus we measured wall clock time throughout. The computional paths of the test runs are analyzed in Section 5 to show *how* the speedup factors were achieved. In Section 6, we argue why methods based on uniform parallelism are in general not as successful as methods based on specialized parallelism. To this end, we discuss another implementation executing a uniform code on each processor. Section 7 concludes with a discussion of extensions of the main algorithm that can further enhance parallelism.

2. Selection of the algorithm

We first outline the basic features of simplex-based algorithms. Using these features we then characterize two basic types of multiprocessor algorithms. In a third part of this section we define our measure of efficient use of multiple processors. In particular, we contrast the notion of linear speedup in the context of a *problem* with that used to describe the degree to which a parallel *algorithm* can take advantage of additional processors.

2.1 Basic features of simplex-based algorithms

The overall algorithm contains some operations that are executed only once and others that are repeatedly executed. The former consist of reading the data, initialization, computation of the optimal objective function value and output of the solution flows, all of which can be parallelized in a straightforward fashion. Hence we focus our discussion on the latter, consisting of the *iterated* operations. The following paragraph will introduce the necessary notation. We caution that it will not by itself explain all the concepts of simplex-based network flow algorithms. We refer to [KH80] for a broader exposition.

There are three basic operations common to simplex-based algorithms.

Pricing ("Selection of the pivot column"). Pricing consists of finding arcs with negative "reduced costs". Recall the equation in (NF). If we split A, c and x into basic and nonbasic parts then $Ax = r$ can be solved for x_B: $x_B = A_B^{-1}(r - A_N x_N)$. Hence $c_B x_B + c_N x_N = const + (c_N - \pi_B A_N)x_N$, where $\pi_B = c_B A_B^{-1}$ is the "dual variable". Pricing a nonbasic arc i, from node k to node l, consists of checking whether the reduced cost of arc i, $(c_N - \pi_B A_N)_i$, is negative. Given the special structure of A, this is equivalent to checking whether $c_i - \pi_k + \pi_l < 0$.

Cycling ("Selection of the pivot row"). On adding an arc to a spanning tree, a cycle occurs. It is on this cycle that the flow changes. If the problem is unbounded, the change is infinite; otherwise an arc j along the cycle will reach its bounds (0 or u_j). The first arc that allows the least flow change leaves the basis[1], re–establishing the tree structure of the basis representation. The cycling operation determines this arc and identifies the four nodes *cut* , *notcut* , *join* and *severed* as illustrated in figure 2.1. *Cut* and *notcut* are the nodes connected by the entering arc. *Cut* is the node above which the old tree is cut as the bounding arc leaves the basis. *Severed* is the node on the leaving arc furthest away from the *root* of the basis tree. *Join* is the first common ancestor of *cut* and *notcut* .

Updating. Updating changes the flow, the dual variables and the structure of the tree. The flow is altered within the cycle according to the results of Cycling. Arcs leaving the basis at the upper bound reverse orientation ("reflection"). The subtree originally attached to *severed* has to be *re-attached* as a subtree of *notcut* . Finally the dual variables in the re-attached subtree change by the amount computed in Pricing.

A close examination of the operations shows that Pricing lends itself naturally to parallelization. The processors can be associated with a fixed set of arcs in a straightforward

[1] If this arc is the entering arc it is set to its upper bound and the basis remains unaltered.

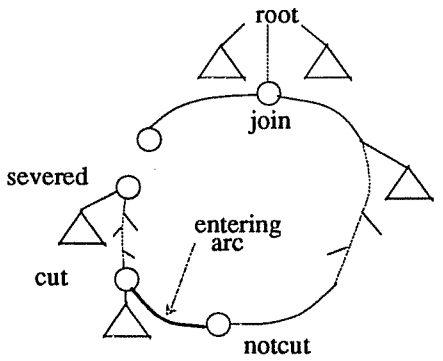


Figure 2.1: A network pivot.

manner [1]. The access to arc information does not need to be controlled by locking. Cycling and Updating on the other hand have dependencies, since changing the tree structure interferes with traversal. Finally, we note that Pricing will yield accurate results only if the duals involved are currently valid.

2.2 Two approaches to parallelizing simplex-based algorithms

We now classify techniques of parallelization into two groups based on the above operations.

2.2.1 Uniform Parallelism

If each processor has an identical code and performs Pricing, Cycling and Updating on its own, we call the parallellism **uniform**. We can subdivide this category further, based on the objects that have to be locked. The granularity of locking determines how dynamically processors can be associated with parts of the underlying graph.

In the **subgraph locking** approach each process owns a collection of subgraphs,each protected by a lock. If a process needs nodes or arcs outside its collection of subgraphs in order to pivot, the process acquires subgraphs from other processes. In the context of network flow problems, it is natural to consider *subtrees* as the units to be locked. See [CM88] for an application to Generalized Network Problems. Subtree locking is efficient only if there are a large number of "independent" subtrees so that the processors need not compete or wait for access to the locked parts of the graph.

In the **cycle locking** approach each process tries to acquire a cycle. The locks are associated with single nodes. The processes compete for the locks on the nodes and give up the locks as soon as possible. We describe an implementation of this approach in Section 6. Cycle locking is efficient only if there are a large number of non-overlapping cycles at all times during the computation.

---

[1] Arcs are often sorted using the "from" nodes as primary and the "to" nodes as secondary keys. We would then simply associate the i^{th} processor with the i^{th} part of the sorted list.

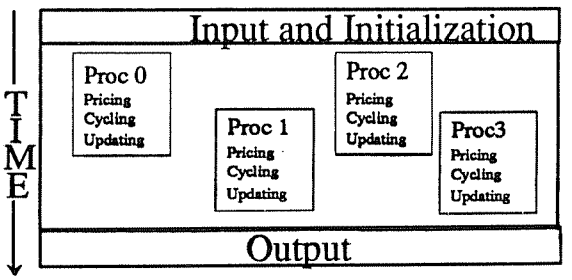


Figure 2.2.a: Uniform parallelism - Processes have identical code.

Figure 2.2.a illustrates the concept of uniform parallelism.

While the above approaches are well suited for largely independent subproblems, dependencies can cause considerable amounts of backtracking and idle waiting. The following two approaches cut down on the number of locks and the amount of contention at the cost of reduced parallelism. In particular, only one update will be performed in each time slot.

2.2.2 Specialized Parallelism

If processors have differing, specialized codes and perform only one or two of the three basic operations we call the parallellism **specialized.**

The **pricing heuristics** approach makes $n-1$ processors compete to find the most promising arc (based on a heuristic), while a single processor is dedicated to Cycling and Updating. The processors dedicated to Pricing continue their work (with possibly incorrect data) during the update.

In the **parallel update** approach, all n processor compete to deliver the promising candidate. Additionally, major parts of the updating task are shared by all processors.

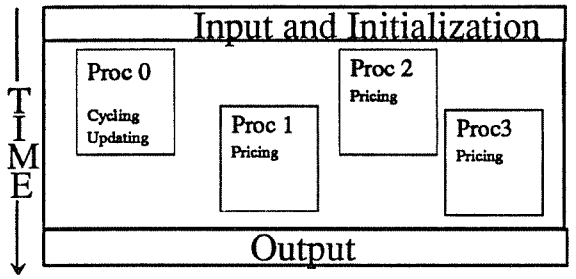


Figure 2.2.b: Pricing heuristics schema.

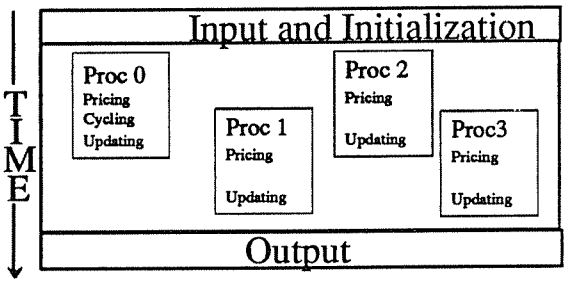


Figure 2.2.c: Parallel update schema.

Our main implementation favours the **pricing heuristics** approach, but incorporates parts of the **parallel update** idea. As the next four sections will document, this approach excelled on the available set of test data. In contrast, the performance of our **cycle locking** implementation was impeded by a considerable number of backtracks, and the negligible

number of "independent" subtrees made the **subtree locking** approach inefficient. Also, the size of the re-attached subtrees and thus the amount of work available proved to be too little to justify a pure **parallel update** approach.

2.3. The measure for efficient use of multiple processors

We start with the observation that algorithms based on specialized parallelism cannot run on a single processor machine. Thus we need a measure that judges the efficient use of additional processors with respect to an *arbitrary initial number* of processors. Let $t_j(p)$ be the *average real time* needed by j processors to solve the instance p of the problem (NF) and

$$s(j;p) := \frac{t_j(p) * j}{t_{j+1}(p) * (j+1)}.$$

We will say that we achieve **linear speedup from j to j+1 processors for a set $\mathcal{P}$ of instances** if $s(j;p) \geq 1$ for all $p \in \mathcal{P}$. In particular, we achieve **linear speedup for a k-processor machine on a set of test problems $\mathcal{P}$** if

$$1 \leq s_{m,k}(\mathcal{P}) := \min_{m \leq j < k; p \in \mathcal{P}} s(j;p),$$

where m is the minimum number of processors necessary to run the algorithm. The efficiency of the m-processor version has to be established by comparing its real time performance with a good (preferably the best) single processor algorithm.

We now set our notion of speedup into perspective. Let $\|p\|$ be the size of an instance p of the problem (NF) and let $f(\|p\|)$ be the worst case sequential time complexity for (NF). Then for $k < f(\|p\|)$ our notion is weaker than the notion of asymptotic linear speedup for (NF), which can be expressed as $1 \leq s_{m,f(\|p\|)}(\{p : p \in NF\})$. Another related notion is *efficient parallelizability*. Efficient parallelizability characterizes *problems* that can be solved in $O(\log^i(\|p\|))$ time using as many as $O(\|p\|^l)$ processors, where l and i are integers [1]. This notion is also weaker than asymptotical linear speedup. The relationship to our measure is not clear.

We take the opportunity, however, to point out that (NF) is most probably not efficiently parallelizable. For this we define P to be the class of problems solvable in polynomial time by a deterministic algorithm. A problem is then said to be *log-space complete for P* if it is in P and any problem Q in P can be reduced to it using no more than $O(\log(\|q\|))$ space, where q is an instance of Q. It is conjectured that log-space complete problems are not efficiently parallelizable [2]. Since the maximum flow problem is log-space complete for P [GSS82], so is the minimal cost network flow problem.

---

[1] This class is called NC. See [Pi79] for a formal definition of NC.

[2] In [Co83] the situation is compared to finding a polynomial time algorithm for an NP-complete problem.

3. The main implementation

We start by motivating the data structures of the implementation. The $m \times n$ constraint matrix A is represented as a graph with n arcs. Accordingly, we have arrays of size n: for each arc i we record the node from which the arc emanates ($FROM_i$), the node into which it leads ($INTO_i$), the associated cost ($COST_i$) and the capacity (CAP_i). A bit array ($FLIP$) indicates whether a variable is at its upper bound. All other arrays are of length m. Flows (the primal variables $FLOW$) and duals ($DUAL$) need only be recorded for basic variables. The basis tree is defined by specifying a predecessor ($PRED$) for each node.

Further arrays help speed up the dual update and the cycling process. A preorder list of nodes ($SUCC$) allows visiting all nodes in a subtree in an efficient manner. Storing the number of successors ($NSUC$) with each node helps to find *join* during Cycling. For a parallel implementation this data structure should be chosen over an array that stores the depth of the nodes in the tree. Since $NSUC$ records the size of subtrees, we can decide whether to split work among several processors or to use a single processor and avoid overhead. The algorithm also maintains an array $ASUC$ of pointers such that $i = ASUC[SUCC[i]]$. After the subtree under *severed* has been re-attached to the full tree as part of the update operation, the successor list of these nodes has to be adjusted. This entails finding the node i such that $SUCC[i] =$ *severed* . $ASUC$ helps us to find i without searching the (possibly whole) tree.

In the following, p_0 will stand for processor 0, and p_i for processors i with $i \in \{1, \ldots, n-1\}$. Besides updates of the shared memory representation of the basis tree the communication between the processes is restricted to two queues (see figure 3).

The **price-queue** is in general a priority queue [Se84]. It contains at its top position the name of the currently most promising arc. In our implementation we chose the priority queue to be a stack, filled by the p_i and emptied by p_0, and the most promising arc to be the arc with the most negative reduced cost. In particular, suppose that one of the p_i finds an arc with negative reduced cost in its subset of the arc array (see Section 2). It then locks the price-queue and deposits the newly found arc, if it is more promising than the arc currently at the top. The p_i perform pricing throughout (Cycling and) Updating. They price on the average one arc per dual node updated by p_0.

After completing an update, p_0 acquires the stack (making a fresh empty stack available to the p_i). If the stack acquired by p_0 is empty, [1] p_0 checks all arcs for negative reduced cost (interupting the search if the price-queue has a new entry). If p_0 finds no arc with negative reduced cost, p_0 notifies the p_i and the algorithm terminates.

An entry of the stack is *valid* if it has a negative reduced cost when the stack is acquired by p_0. Since the update progresses as the p_i price out, the ordering of the stack is more reliable towards the top. Yet there is no guarantee that the topmost element is valid or the most promising arc. Thus, if the stack is not empty, p_0 chooses the entering arc from a (small) number of valid entries starting from the top. In our implementation

[1] Measurements during the test runs indicate that empty stacks occur only about a total of 3 to 8 times towards the end of the computation.

p_0 chooses the best of at most 3 valid entries. The price-queue forces a comparison of the proposals of all pricing processors. In this sense the priority queue guarantees candidates drawn from a *global* comparison.

The **work-queue** is in general also a priority queue. It allows the transfer of work from p_0 to the p_i. Again we chose a stack structure for the priority queue. There were at most two entries in the stack at any time since we did not parallelize the update of the re-attached subtree (see Section 7 for an extension). In particular, suppose that the number of nodes (counted during Cycling) on the path from *join* to *notcut* is larger than some fixed number (*wqmin*). Then p_0 makes an entry in the work-queue describing the path to be updated and the changes in $NSUC$ and $FLOW$ along that path. The processes p_i check the work-queue frequently. If there is an entry, they lock the work-queue, remove the entry, unlock, and perform the update. The parameter *wqmin* should be chosen such that the extra work for p_0 in entering the information is less than doing the update itself. A reasonable choice for *wqmin* is 10.

The algorithm NETPAR can now be stated as follows.

step 0:

p_i: Read and initialize.[1]

p_0: Read and initialize.

step 1:

p_i: Price out. Check work-queue. If the work-queue is non-empty grab work. If notified by p_0 go to step 2.

p_0: If there is no valid entry in the price-queue check for termination. If no arc with negative reduced cost is found, notify p_i and go to 2. Take the most promising arc from the price-queue. Cycle and update (fill the work-queue). Repeat step 1.

step 2:

p_i: Output flows. Compute objective function value.

p_0: Output flows. Compute objective function value.

The general general idea of the algorithm is captured in figure 3.

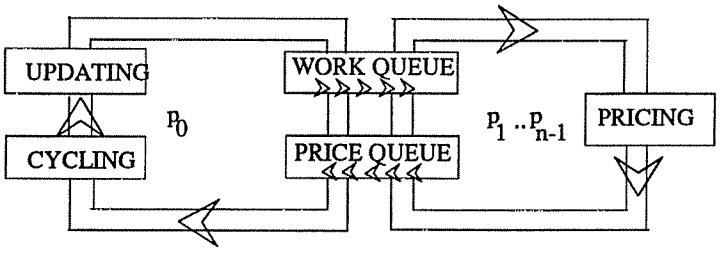


Figure 3: General idea of the algorithm

[1] NETPAR uses an artificial starting basis as discussed in [GKKN74]

4. Performance of *NETPAR*

NETPAR was tested on the (new) NETGEN benchmark problems [KNS74]. The generator allows the user to specify certain problem parameters, like the number of nodes, the percentage of bounded arcs etc.. Once the parameters, in particular the random seed, are specified, the problems are generated deterministically. Thus the benchmark problems are well defined by a list of parameters and the generating program.

The problems are divided into groups. In each group one problem parameter is varied to pinpoint the performance of the algorithm.

More specifically, the new NETGEN benchmark problems are variations of the following two basic problems:

basic **transportation** problem	
total supply	250000
nodes	5000
sources	2500
sinks	2500
arcs	25000
mincst	1
maxcost	100
transp. sources	0
transp. sinks	0
% high cost arcs	0
% capacitated arcs	100
min. capacity	1
max. capacity	1000

basic **transshipment** problem	
total supply	250000
nodes	5000
sources	500
sinks	500
arcs	25000
mincst	1
maxcost	100
transp. sources	500
transp. sinks	500
% high cost arcs	0
% capacitated arcs	100
min. capacity	1
max. capacity	1000

We were able to test NETPAR on two versions of the SEQUENT^{TM} multiprocessor. The older version, the SEQUENT Balance B-21000 [SB84], consisted of 8 NS 32032 processors. Each processor had a 8 kbytes cache attached. Physical memory was limited to 8 Mbytes. The version as of January 1988, the SEQUENT Symmetry S-81 [SS87], consists of 10 Intel 80386 processors. It is geared towards floating point arithmetic reducing the number of integer registers available to NETPAR from five to three. On the other hand the processors are approximately three times as fast and the cache size is increased to 16 kbytes. Physical memory is currently limited to 40 Mbytes.

We compare NETPAR first with a standard code, NETFLO [KH82]. Then, we measure speedup of NETPAR^{i+1} over NETPAR^{i}, where NETPAR^{k} stands for NETPAR using k processors.

4.1 NETPAR versus a single processor algorithm

NETPAR, as described in Section 3, needs at least 2 processors. Hence there will be no entry corresponding to a single processor in the speedup diagrams. However, we consider NETFLO, a close relative of our algorithm. To set the results of NETPAR into

perspective, we show how NETPAR using 2 processors compares with NETFLO on the SEQUENT Symmetry. From each group of test problems at least one is displayed below.

NETFLO vs. NETPAR^{2*}						
	101	104	106	110	115	116
NETFLO (t)	729.30	802.60	399.60	430.50	698.10	636.30
NETFLO (i)	50307	51006	32528	31342	44857	36618
NETPAR (t)	89.12	93.51	36.28	42.55	69.43	87.70
NETPAR (i)	19162	19352	12666	12738	15784	16917

* NETPAR using 2 processors
i:iterations t:time

NETFLO vs. NETPAR^2						
	117	121	122	126	130	134
NETFLO (t)	344.90	802.70	802.60	441.60	500.50	195.40
NETFLO (i)	23707	89088	78026	41036	43681	15749
NETPAR (t)	36.68	124.03	103.22	38.81	42.34	26.85
NETPAR (i)	10893	34297	27620	14050	14056	11830

NETFLO vs. NETPAR^2					
	138	142	144	147	150
NETFLO (t)	1308.80	802.60	794.20	2530.70	802.60
NETFLO (i)	108822	64964	71496	195150	85584
NETPAR (t)	126.24	83.79	96.22	187.90	90.62
NETPAR (i)	35844	23061	24768	34753	25777

We conclude that NETPAR^2 is 8-10 times faster than NETFLO on the test problems. Note in particular the different number of iterations. Section 5 will detail the influence of the total number of iterations on the computing time.

4.2. Speedup of NETPAR

Now we consider the performance of NETPAR^i as i varies. The resulting amount of data is considerable. Nevertheless, we hope to convey the essence of the information by presenting two diagrams for each problem group and displaying two graphs for each problem.

The two diagrams show results on the two versions of the SEQUENT multiprocessor. The results on the Balance version are displayed on the left unless indicated otherwise. All results on the Symmetry version are displayed on the right. The problems 101 through 120 are transportation problems, 121 to 150 are transshipment problems. The varied quantities are listed with each figure. For example, "total supply [103: 6,250,000]" means that problem 103 had the specifications of the basic transportation problem with the total supply altered to 6,250,000 units. The labels of the figures are *lower case* for transportation problems and *upper case* for transshipment problems. The last letter indicates the multiprocessor environment, i.e. "B" stands for Balance and "S" for Symmetry. Times are

given in seconds and measured from start to end of the program, including data input, initialization and computation and output of primal and dual solution. The times do not include output of the basis arcs and their flows.

We display two graphs for each test problem. The solid line displays timings as the number of processors increases [1]. Emanating from each data point there is a dotted line indicating the linear speedup as defined in Section 2: the dotted line indicates the maximal possible improvement in the computation time as the $j+1^{st}$ processor is added to the workforce *assuming the total amount of work remains unchanged.* The dotted line can lie above the solid line, if the additional processor decreases the overall work, for example by helping to chose better arcs. On the other hand, if the dotted line lies above the solid line at the $j+1$ processor mark, there is a better algorithm for j processors than the one used: namely an algorithm in which the j processors mimick $j+1$ processors.

We chose to display real time versus the number of processors, since we consider the real time performance of an algorithm the ultimate reason for developing parallel algorithms for numerical problems. So called "speedup diagrams" displaying the ratio $(\frac{\text{time for 1 processor}}{\text{time for n processors}})$ against the number of processors are often flawed. Their validity depends on the quality of the single processor entry. If the single processor algorithm is inefficient, "superlinear speedup" (in the sense of the speedup diagram) may be easily achieved but worthless as a performance criterion.

[1] The data points have been connected to help the eye.

Transportation problems: 101 to 120

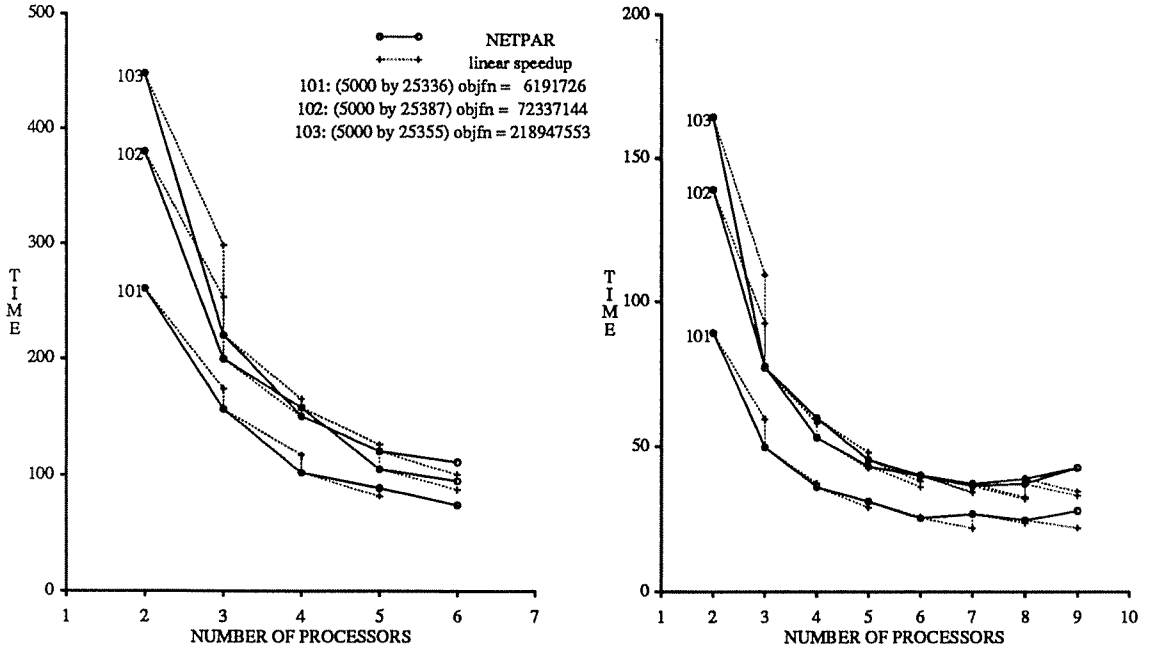


Figure 4.sdB/S : total supply [101: 250,000, 102: 2,500,000, 103: 6.250,000]

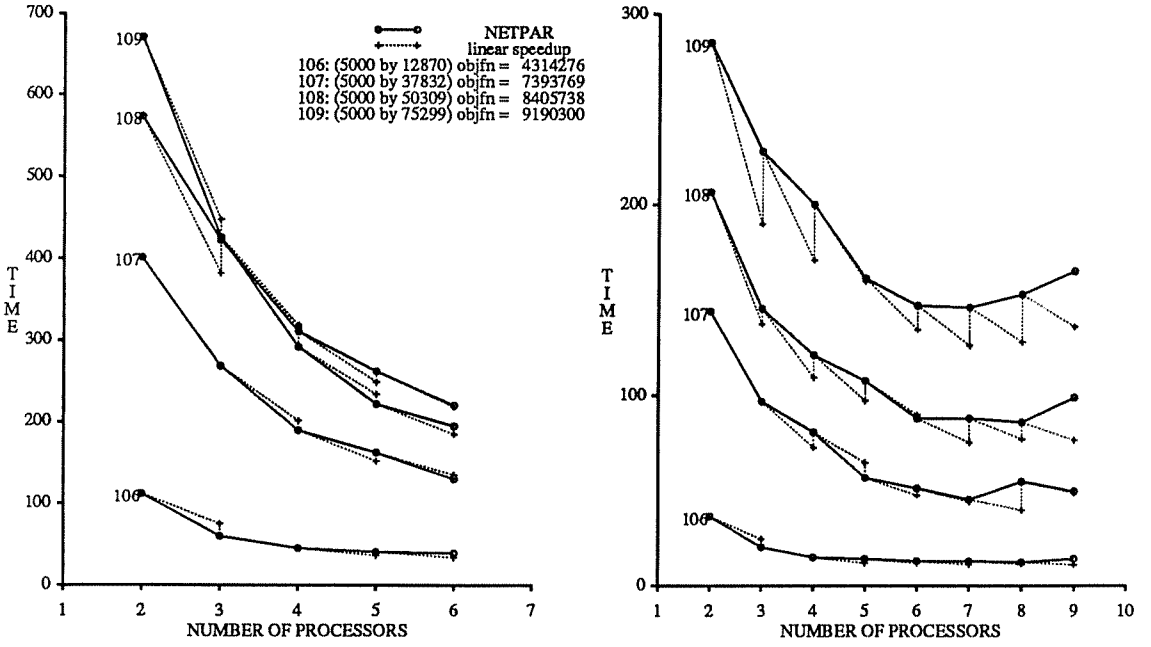


Figure 4.apB/S: arcs [106: 12,500, 107: 37,500, 108: 50,000, 109: 75,000]
(proportional) supply [106: 125,000, 107: 375,000, 108: 500,000, 109: 750,000]

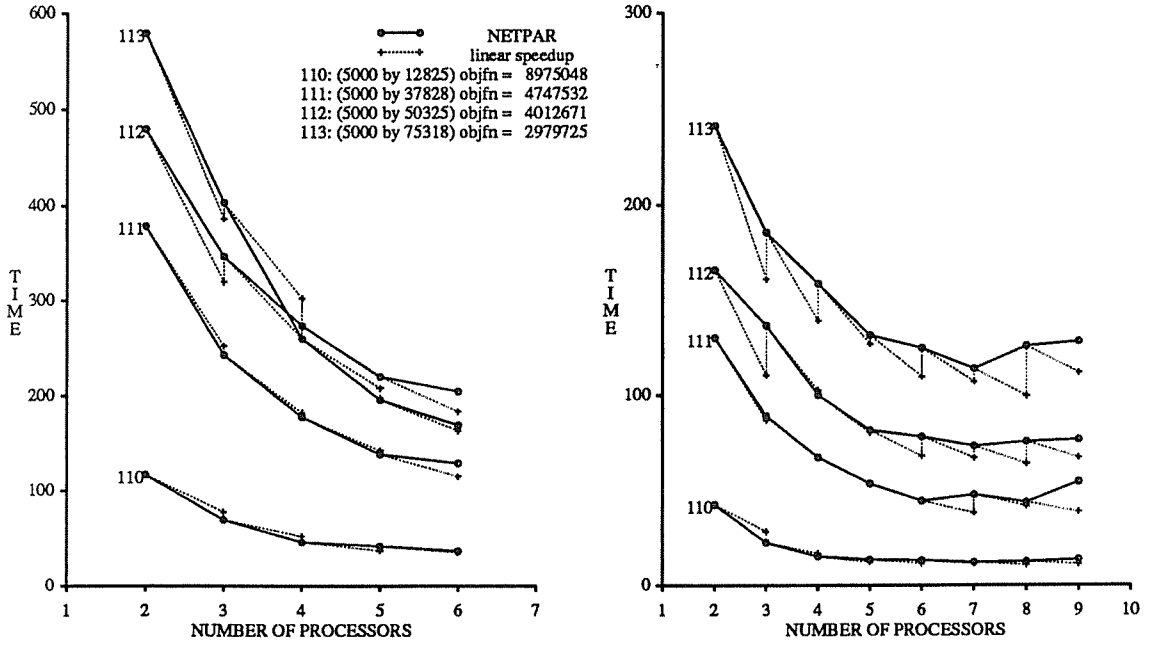


Figure 4.acB/S: arcs [110: 125,000, 111: 375,000, 112: 500,000, 113: 750,000] constant supply [250,000]

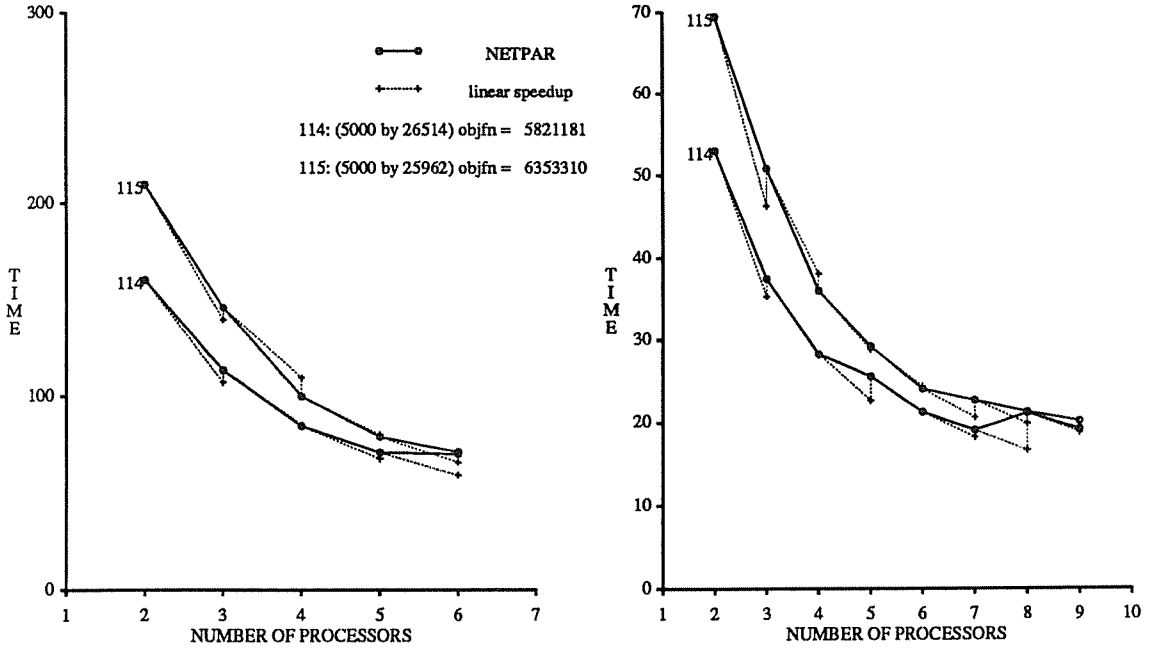


Figure 4.rB/S: transportation rectangularity [114: 500 sources and 4500 sinks, 115: 1500 sources and 3500 sinks]

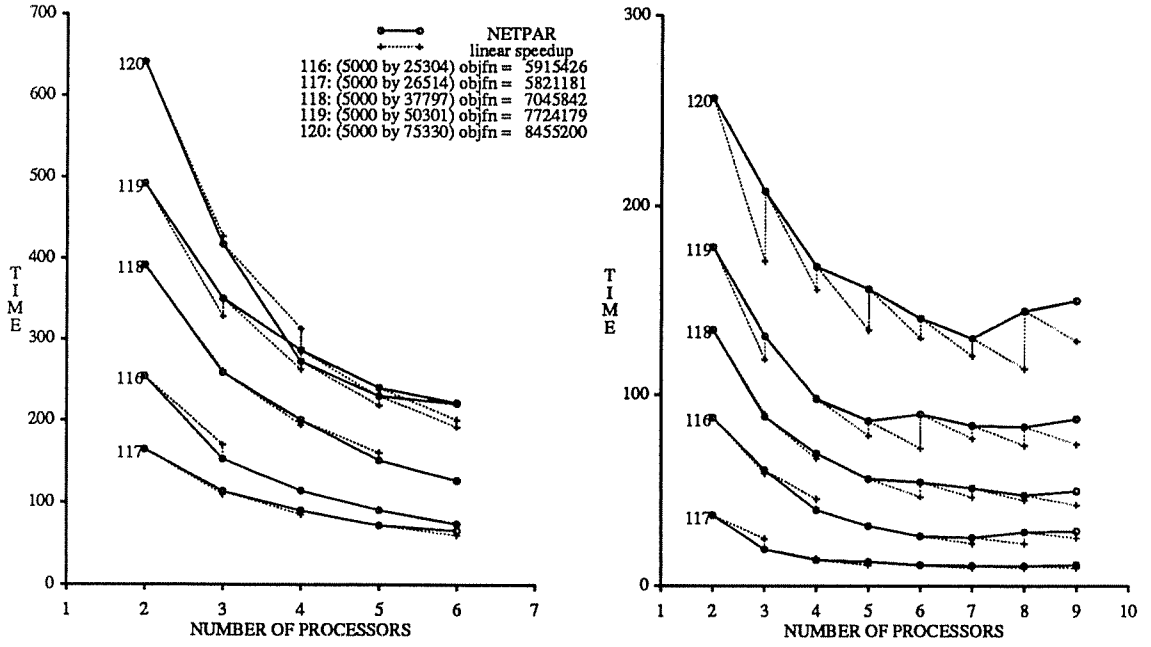


Figure 4.auB/S: arcs [106: 12,500, 107: 37,500, 108: 50,000, 109: 75,000] (proportional) supply [116: 125,000, 117: 375,000, 118: 500,000, 119: 750,000] unlimited capacities

Transshipment problems: 121 to 150

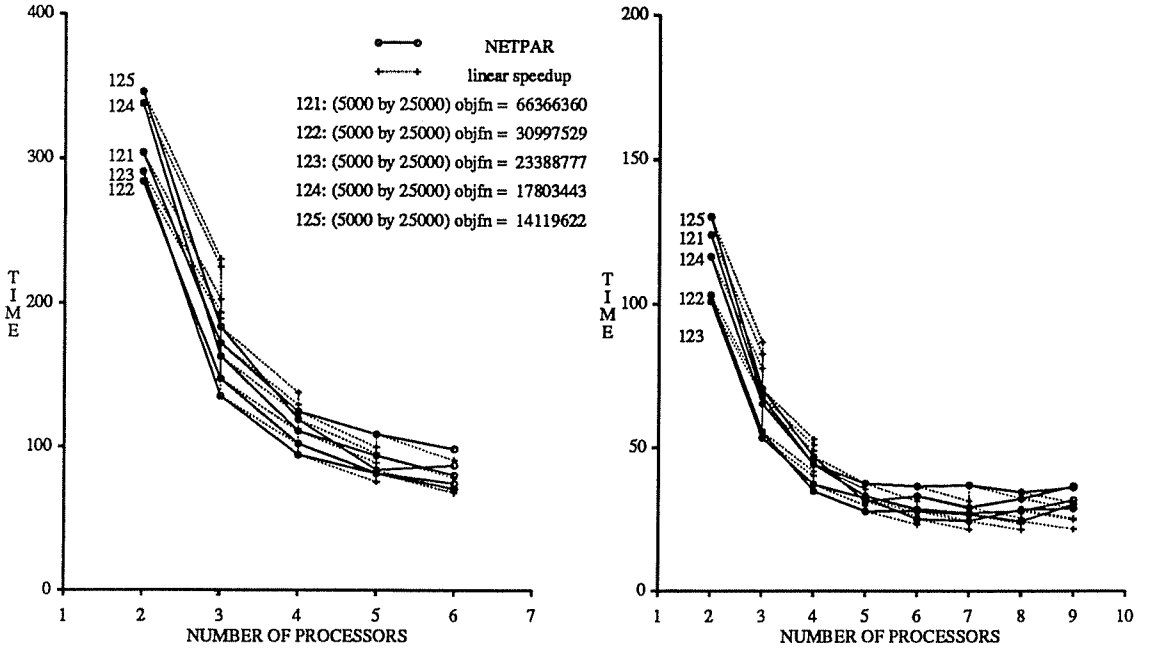


Figure 4.SDB/S: number of sources / sinks [121: 50/50, 122: 250/250, 123: 500/500, 124: 1,000/1,000, 125: 1,500/1,500]

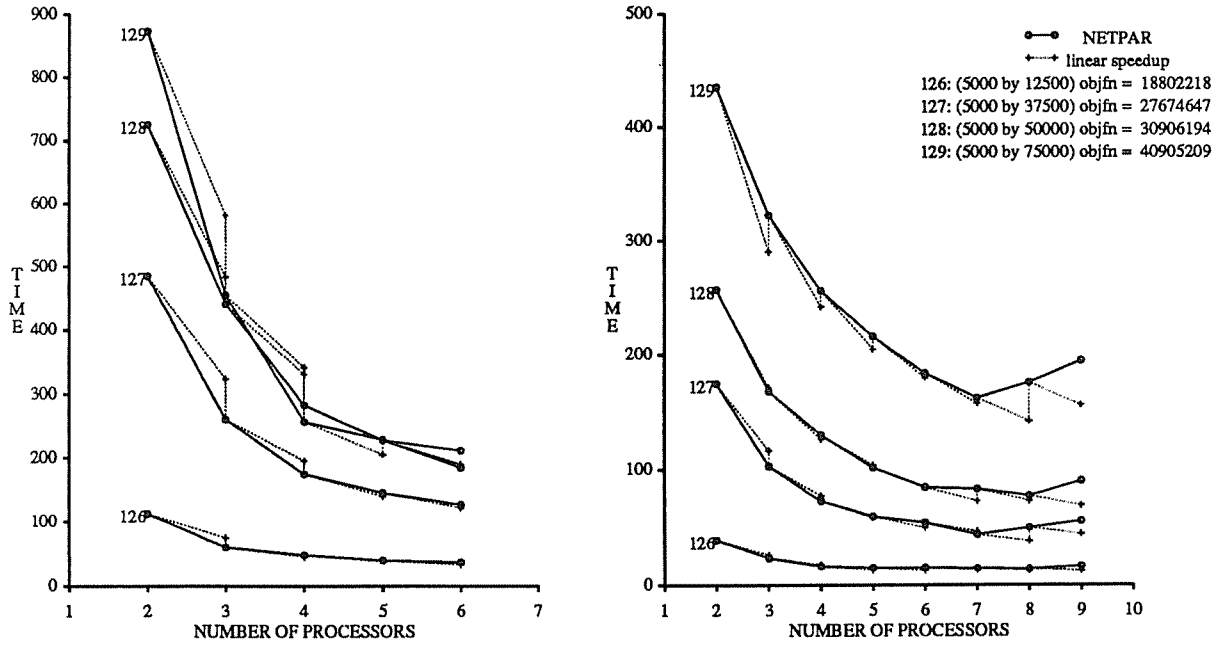


Figure 4.APB/S: arcs [126: 12,500, 127: 37,500, 128: 50,000, 129: 75,000] (proportional) supply [126: 125,000, 127: 375,000, 128: 500,000, 129: 750,000]

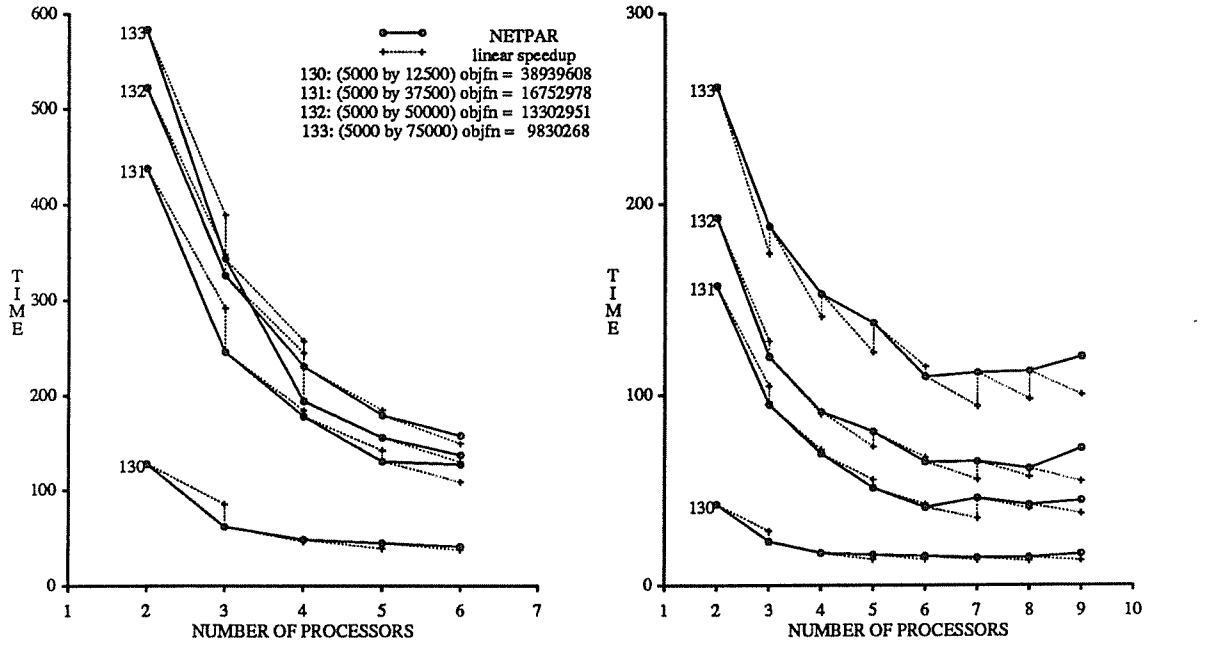


Figure 4.ACB/S: arcs [130: 12,500, 131: 37,500, 132: 50,000, 133: 75,000] constant supply

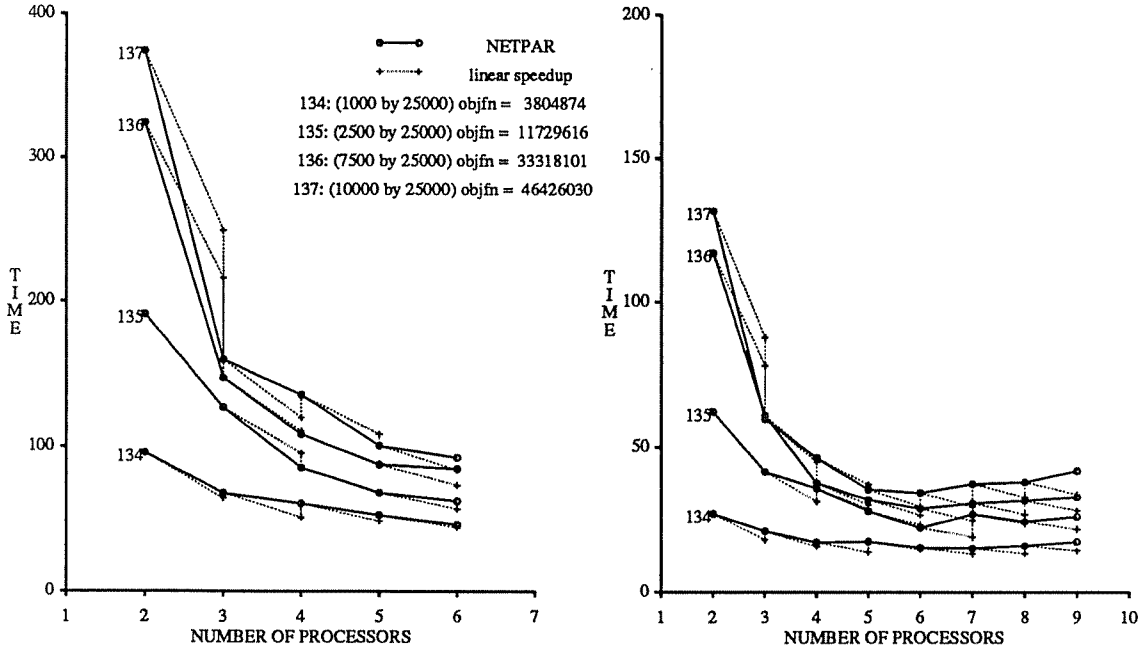


Figure 4.NB/S: nodes [134: 1,000, 135: 2,500, 136: 7,500, 137: 10,000]

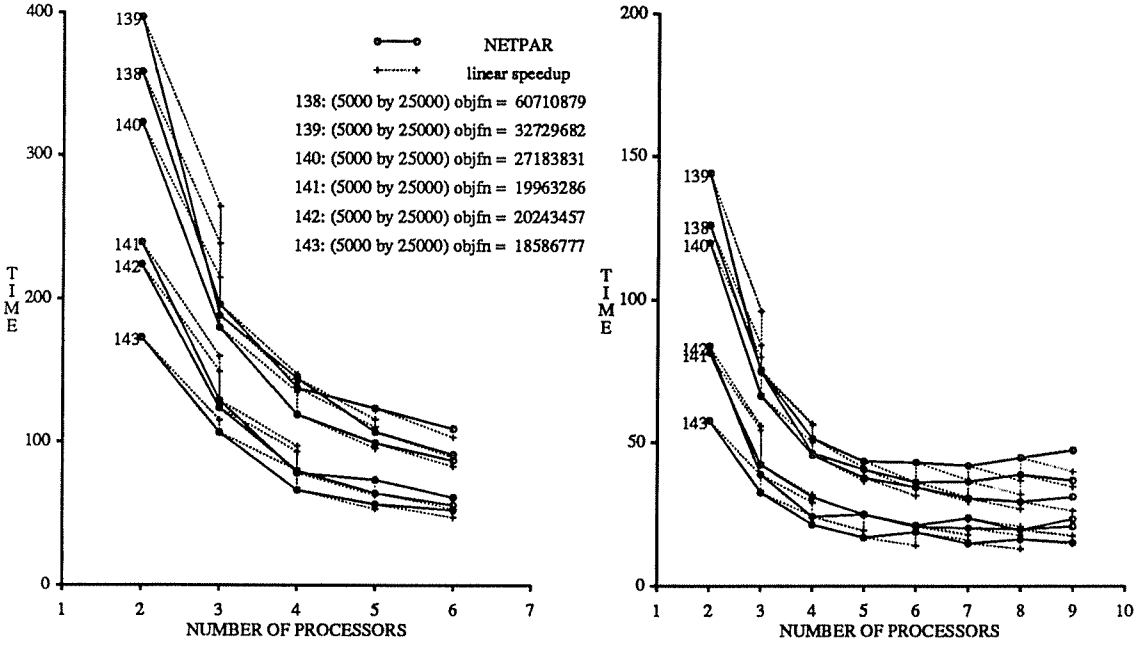


Figure 4.UBB/S: (upper bound range) capacitation [138: 50, 139: 250, 140: 500, 141: 2,500, 142: 5,000, 143: ∞]

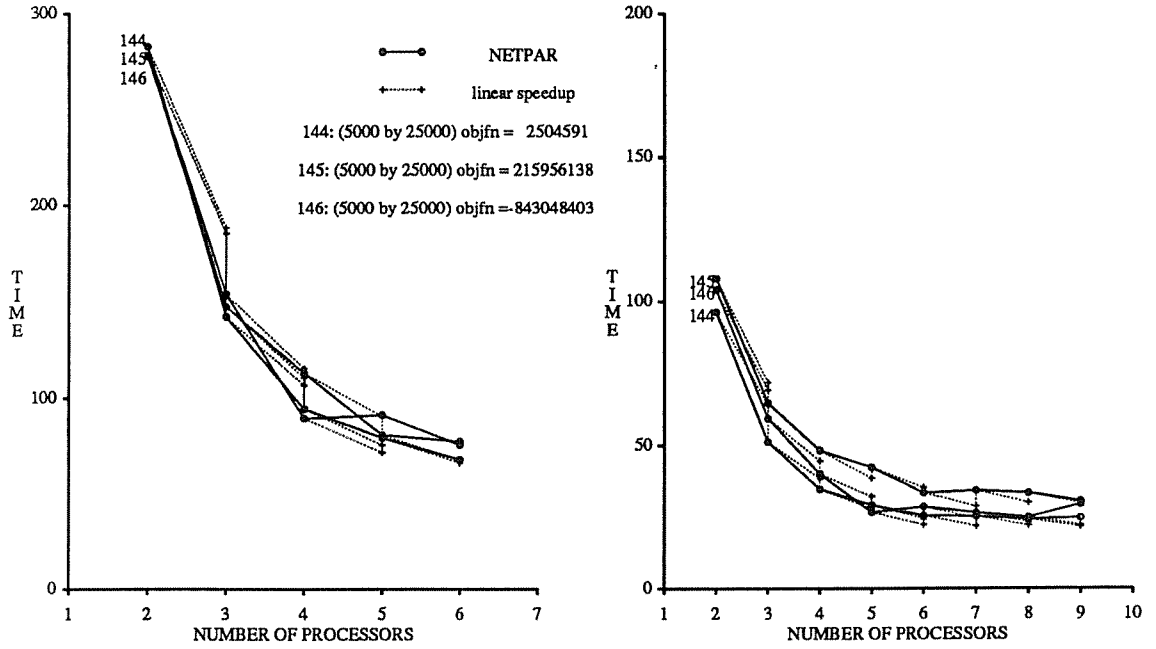


Figure 4.CRB/S: cost range [144: 1-10, 145: 1-1000, 146: 1-10,000]

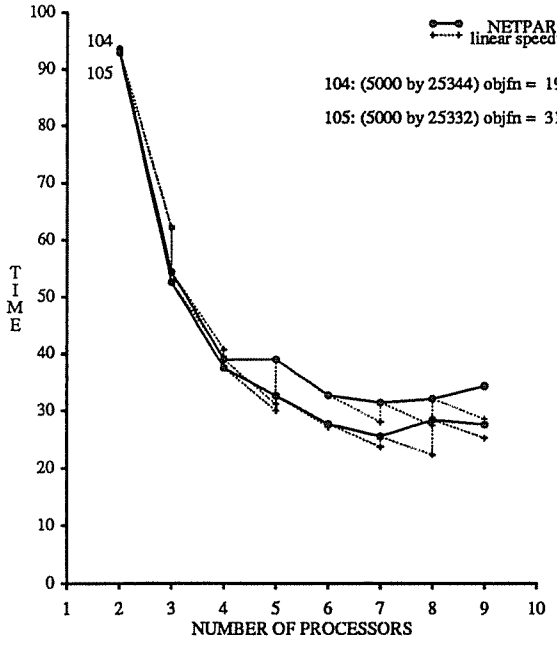


Figure 4.csS: cost shift [104: -100 to -1, 105: 1 to 100]

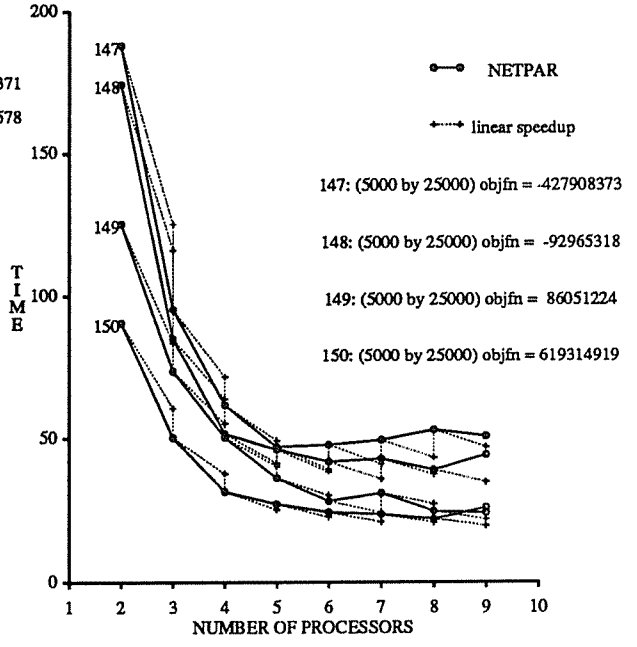


Figure 4.CSS: cost shift [147: -100 to -1, 148: -50 to 49, 149: 101 to 200, 150: 1001 to 1100]

To test the performance of NETPAR on some *larger* problems we created problems 201 through 203 drawing from the characteristics of problems 101 through 103. We simply prescribed 10 times as many nodes, arcs, sinks, and sources:

Large NETGEN problems 201 to 203		
Problem	random seed	supply
201	13502460	250000
202	4281922	2500000
203	44820113	6250000

common properties of 201-3	
nodes	50000
sources	25000
sinks	25000
arcs	250000
mincst	1
maxcost	100
transp. sources	0
transp. sinks	0
high cost	0
capacitation	100
min. capacity	1
max. capacity	1000

Below, we show the performance of NETPAR on the problems 201, 202 and 203 compared with the problems 101 to 103.

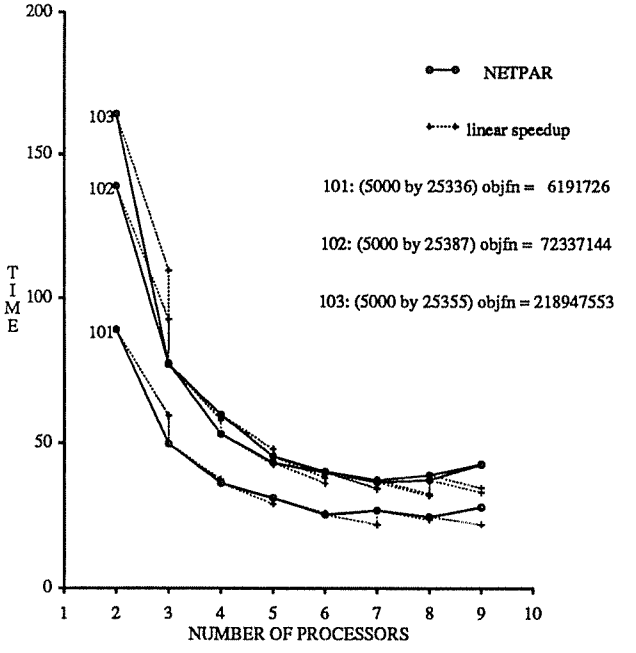


Figure 4.sdS: supply / demand magnitude (Symmetry version)

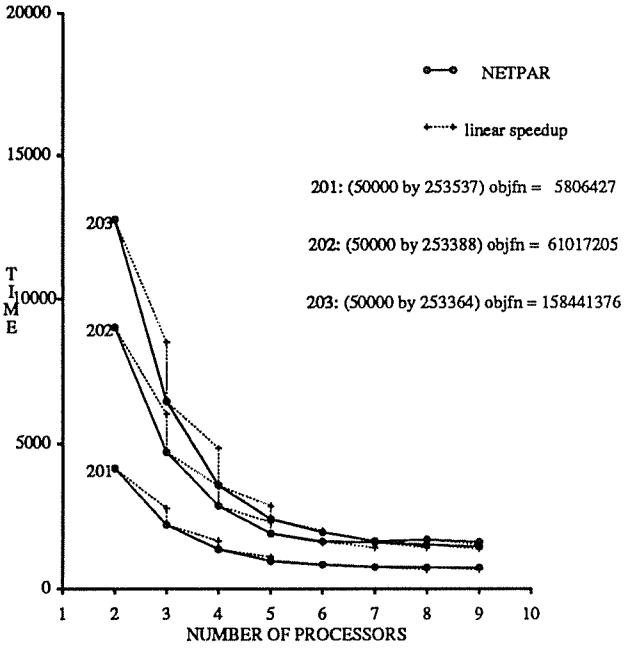


Figure 4.sdS': supply / demand magnitude (Symmetry version)

We observe that for the smaller NETGEN problems, performance does not improve proportionally when we use more than 6 processors. In fact, since additional processors compete for the price-queue,we have at times an increase in the overall running time. For problems 201 through 203 this point of decreased effectiveness is moved to the right. That is, we *observe that more processors can be used efficiently as the problem size increases.*

5. Analysis of the computations

In the previous section we observed that NETPAR performs well in real time (when compared to NETFLO) and achieves superlinear speedup. In this section we exhibit characteristics of the computational paths that shed some light on *why* the **pricing heuristics** approach is efficient. To this end we analyze the distribution of work over time.

While collecting the data for such an analysis, we have to make sure that the additional code and the additional time spent on measurement does not significantly alter the variables we want to measure. (This can be viewed as a version of the "Heisenberg uncertainty principle" for parallel programs.) Thus we restricted the time-dependent analysis to simple measurements *during the test runs.* (The timings in Section 4 include the time spent for data collection.)

First, we display time versus pivots to show how expensive the pivots are. We display two typical graphs showing the problems 102 and 123 solved on the Balance version. The number of processors used is indicated after the problem name, i.e. 123_3 is the graph corresponding to problem 123 solved with three processors. For example, the first 6,000 pivots on problem 102 using 5 processors took about 19 seconds, while the first 14,000 pivots took 51 seconds.

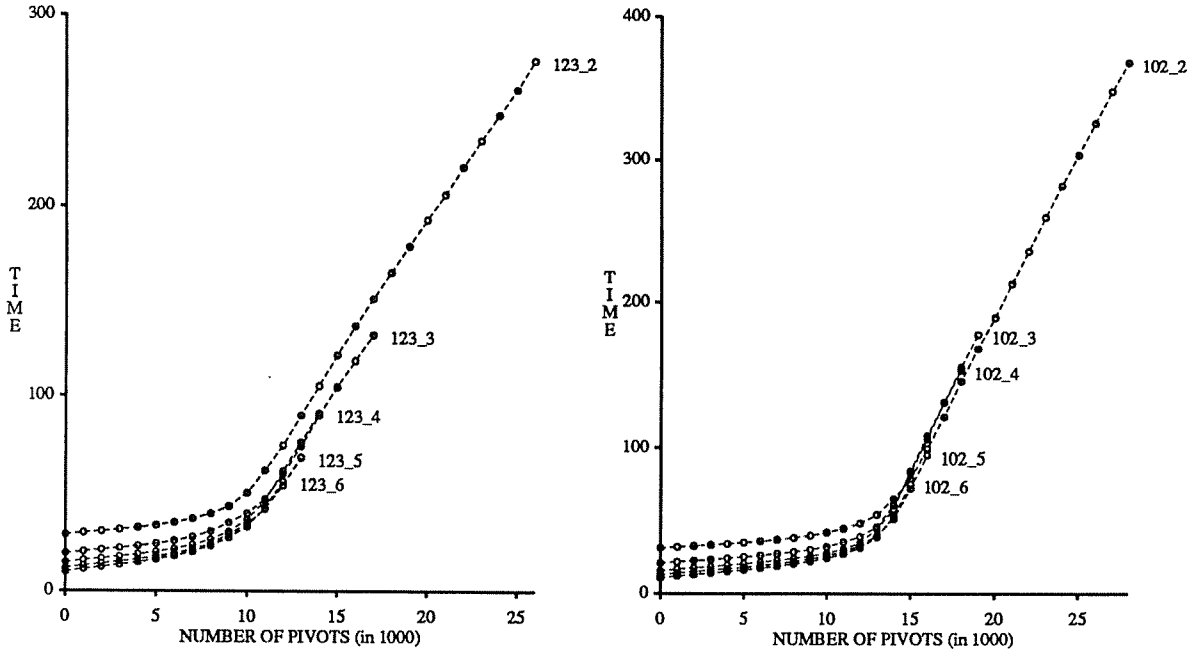


Figure 5.a: **Pricing heuristics** decrease the number of expensive pivots

The increased amount of pricing due to more processors thus reduces not only the number of pivots, but in particular the number of *expensive* pivots

Next, we analyze *why* the pivots are expensive. For this we display the number of nodes whose $FLOW$ and $NSUC$ fields have to be updated, that is, the number of nodes on the *cycle.* We also display the number of nodes whose dual variables have to be altered,

which is the number of nodes in the re-attached subtree. The former is labelled "avg. CYCLE size", while the latter is displayed as the solid "avg. SUBTREE size" graph.

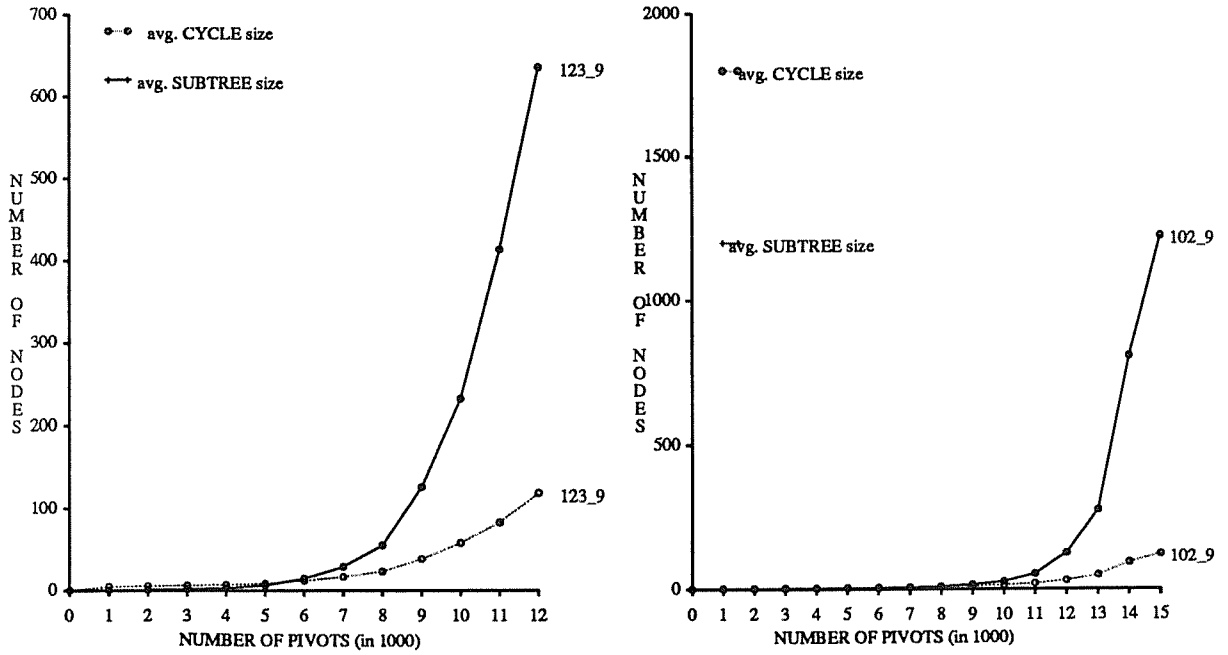


Figure 5.b: **Expensive updates** coincide with the re-attachment of large subtrees.

We observe that updates are expensive when large subtrees are re-rooted. This means that our implementation is successful, because large trees change little. Since we use additional processing power largely to improve pricing, we deduce that *the efforts spent on pricing have a stabilizing influence on the evolving basis trees.*

6. Arguments against Uniform Parallelism

6.1 The subgraph locking approach

The success of **subgraph locking** depends on the existence of an efficient computational path that allows for parallel non-conflicting updates. That is, this approach needs groups of nodes and arcs that can be altered independently over some length of time. While such a clustering problem is in general very difficult, an analysis of the update operations suggests that we look for independent subtrees. To get an idea of how many subtrees would be available for a **subtree locking** approach, we collected the entering arcs' indices during a couple of test runs. This allowed us to **replay** the computation and analyze the dynamic tree structures thoroughly without distorting the computational path. (This replay was also done in parallel by slightly modifying our main implementation.) In particular we could afford to label the nodes at each check interval and thus measure not only how many subtrees coexist on the average, but also how fast the subtrees change. The significance of the following analysis relies on two assumptions. First, that the NETGEN problems are representative of network flow problems in general. Secondly, that most *efficient* computational paths are similar to the paths traced out by our main algorithm. To determine the potential of the subgraph locking approach we considered two types of subtrees.

- **Primal subtrees** are unconnected by flow to the rest of the tree.[1]
- **Dual subtrees** have their root directly connected to *root* .

We logged both the dynamically evolving subtrees and the tree distribution at the solution stage.

First, we present the dynamic picture. The interval chosen for counting the subtrees and the change in the subtrees as well as relabelling with the nodes with their root node label is five iterations. The subtrees are grouped in intervals of 100 nodes and the results are averaged over every 1000 iterations. That is, for example, all observations made on subtrees with more than 400 and less than 500 nodes are recorded together and anew for each interval of 1,000 iterations. We give the labelling code in the Appendix. Thus, in figure 6.a below, we observe an average of 0.19 primal trees in the '400 to 500' row for the interval 6,000 to 7,000 iterations. The black portion of each bar indicates the percentage of change for the group of subtrees, i.e. the number of nodes found with a label different than the subtree root label. A blank bar indicates stability.

Problem 135, solved on the Symmetry version, is chosen as a representative problem for our observations. Problem 135 needed 9,380 pivots for this particular run. We omitted the bar graphs corresponding to the first 6,000, resp. 4,000, iterations since the number of subtrees was essentially 0.

First, we show the primal subtree count.

[1] We include subtrees connected by nonbasic arcs at their upper bounds to the rest of the tree.

Figure 6.a.

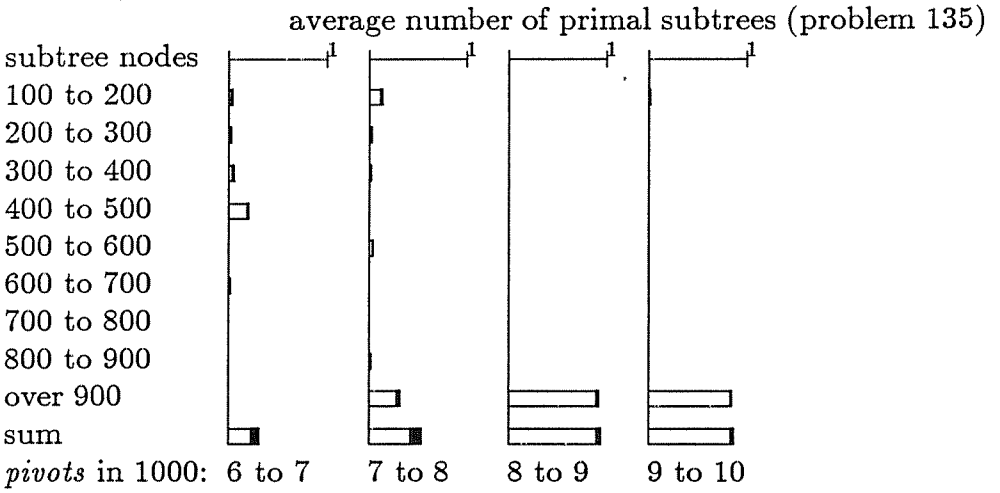
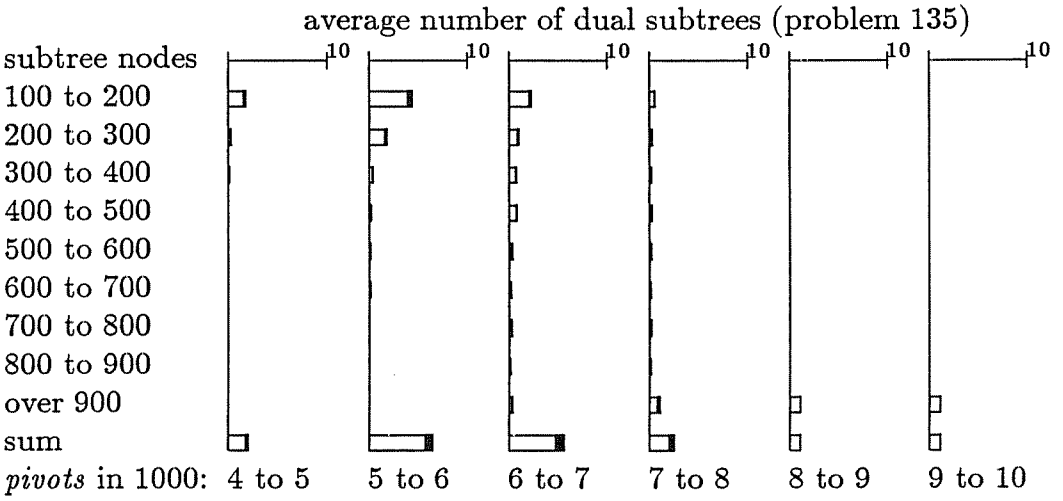


Figure 6.a illustrates that *primal subtrees are scarce.* The averaged number of subtrees stayed below 1. In any case, considering primal subtrees as independent units in the sense of the subgraph locking approach is a flawed decision: unless the subtree root is directly attached to the root of the full tree, dual updates will propagate into it. Primal subtrees are naturally independent units only at *the end of the computation.*

Consequently, we focus on dual subtrees.

Figure 6.b.



Figures 6.b shows that dual trees of an interesting size are more abundant (up to 8 in our example), but become rare towards the end of the computation. Yet the changes in the subtrees occur typically in the large subtrees despite the stabilizing influence of our scheme on the subtree structure; and all subtrees coalesce in the final stages of the computation.

We now look at the picture at the end of the computations. The roots of primal subtrees with 30 and more nodes are counted in the rows labelled *big,primal.* Subtrees that are both primal and dual independent are labelled *big,independent.*

NETGEN Solution Subtrees						
Problem no.	101	102	103	104	105	106
total,primal	626	235	192	647	613	888
big,primal	8	1	2	9	6	12
big,independent	1	1	1	1	1	1

Problem no.	107	108	109	110	111	112
total,primal	595	166	457	802	570	535
big,primal	5	1	5	7	9	7
big,independent	1	1	1	1	1	1

Problem no.	113	114	115	116	117	118
total,primal	486	1520	1058	333	375	328
big,primal	2	4	8	3	15	6
big,independent	1	1	1	1	1	1

Problem no.	119	120	121	122	123	124
total,primal	9	16	2540	2615	2244	1467
big,primal	3	3	5	1	2	2
big,independent	1	1	1	1	1	1

Problem no.	125	126	127	128	129	130
total,primal	1000	2124	2098	1927	1497	1892
big,primal	3	3	1	1	2	3
big,independent	1	1	1	1	1	1

Problem no.	131	132	133	134	135	136
total,primal	2303	2374	2382	40	680	3866
big,primal	2	4	3	2	2	3
big,independent	1	1	1	1	1	1

Problem no.	137	138	139	140	141	142
total,primal	5529	202	1188	1768	2602	2730
big,primal	5	11	4	3	1	1
big,independent	1	1	1	1	1	1

Problem no.	143	144	145	146	147	148
total,primal	2887	2361	2206	2227	24	126
big,primal	1	1	1	3	2	1
big,independent	1	1	1	1	1	1

Problem no.	149	150
total,primal	2660	2698
big,primal	2	3
big,independent	1	1

Thus, according to our criteria, the NETGEN problems end up with a basis consisting of a single big subtree. We conclude that the average number of subtrees during and at the end of the computation does not justify a **tree locking** approach.

6.2 The cycle locking approach.

Similarly, **cycle locking** encounters problems due to the lack of independent subgraphs. In our implementation of this approach we used two different types of locks. The first type, **cycle locks**, are exclusive. If any other lock is encountered while locking the cycle the process must give up its entering arc and erase its locks (backtrack). The locks on the path to *root* from the common ancestor of *cut* and *notcut* , called **path locks**, are not exclusive. A node may have several locks of this kind. But if it has at least one, it can not have a cycle lock. Path locks avoid the "cycle in a cycle" situation where a cycle occurs in a subtree about to be rehung. Without further costly locking, this situation leads to errors in the update of dual variables. The path locks are non-exclusive to avoid unnecesary blocking of paths.

For figures 6.c and 6.d, we counted the number of backtracks *per processor* per 1000 iterations of our cycle locking implementation. The number of collisions reflects wasted work. For example, the 500 backtracks per 1000 iterations using 9 processors to solve problem 121 indicate that 4.5 attempts were necessary to lock one complete cycle.

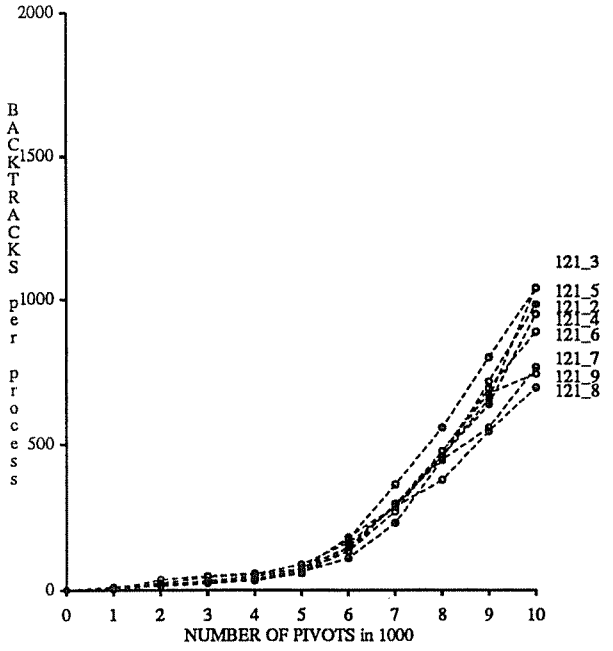


Figure 6.c: lots of collisions (121)

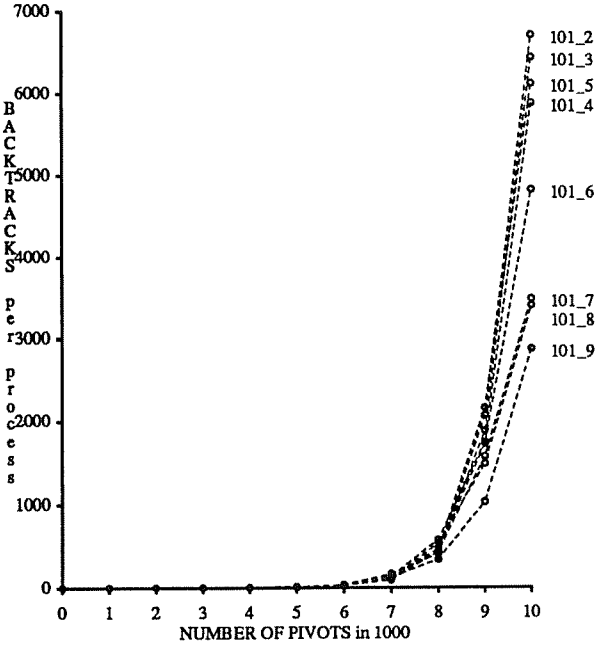


Figure 6.d: more collisions (101)

Our general observation was that the number of collisions increases rapidly after the first 5,000 iterations and more pivots are aborted than completed. To take advantage of the low backtrack rate in the initial phase, we tried to use cycle locking as a first phase, switching to pricing heuristics when the number of backtracks reached a certain level. It turned out that the pure pricing heuristics implementation using 6 processors needed fewer iterations than the pricing heuristics part of this hybrid implementation. This is illustrated by figures 6.e and 6.f.

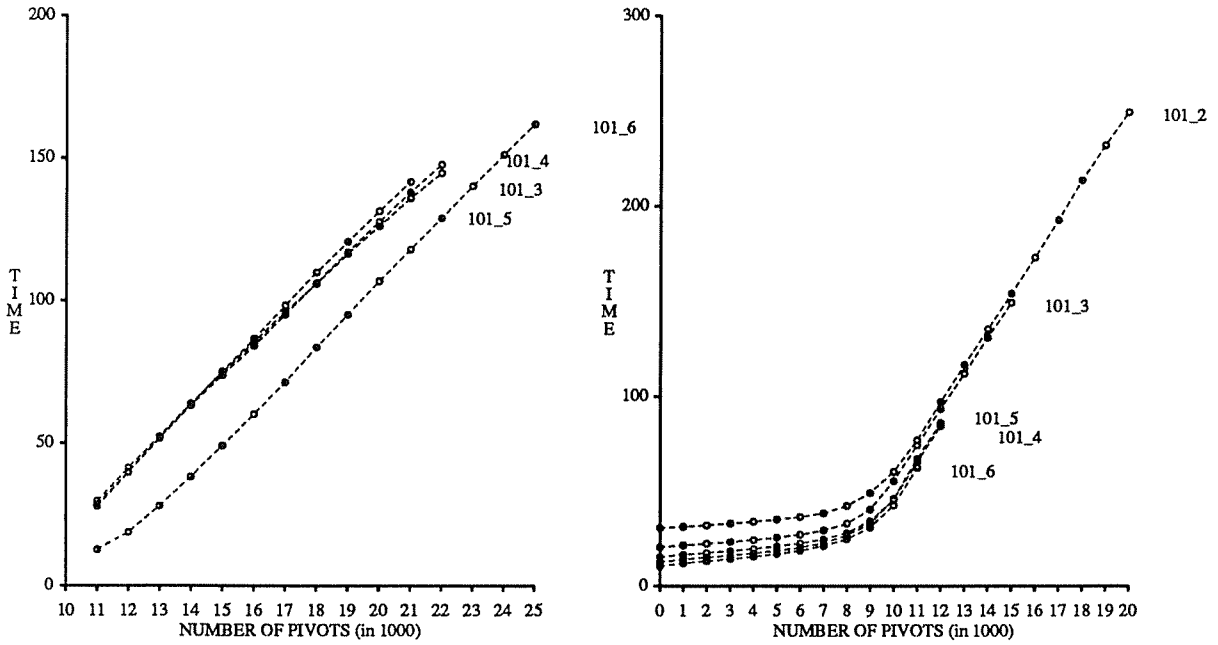


Figure 6.e: The **cycle locking - pricing heuristics hybrid** (left) performs worse than the pricing heuristics approach (right).

Judging by the slope of the graph in figure 6.e and our previous observation on the reasons for expensive updates, it seems that cycle locking grows medium sized trees that have to be re-shuffled. A possible explanation for the poor performance of the cycle locking approach is therefore that it tends to create *local optima*: better arcs are blocked and do not enter because of contention. In particular, arcs that produce large cycles have a lower chance to enter the basis.

7. Extensions of the algorithm.

Recall the graph that plots pivots versus subtree size in Section 5. It shows that the total amount of work available for a **parallel update** is in the order of 50 nodes or less for most of the pivots. On the Symmetry version of the Sequent multiprocessor, updating the dual values of 100,000 nodes takes $\approx$ 1 second if properly coded. Thus it does not seem reasonable to maintain additional data structures or incur a communication overhead for a pure **parallel update** approach. Nevertheless, we present the basic idea, since it might be useful in conjunction with the **pricing heuristics** approach as trees and problems grow larger.

The key data structure is $LSUC$. It contains a pointer to the last element of the subtree of a node. We need this last element to be able to *efficiently* cut the re-attached subtree into pieces. Without $LSUC$ we would have to use $SUCC$ to traverse the whole subtree, which is almost as expensive as doing the update itself. Again, using $NSUC$ we can decide what sub-subtrees should be added to the work-queue. The work-queue entry is of a similar format as in the cycle update case: the subtree root node, the number of nodes to be updated and the changes in $FLOW$ and $DUAL$ and $NSUC$. Special care has to be taken if the subtree root is *cut* . A disadvantage of $LSUC$ is that it potentially needs to be updated all the way up to *root* .

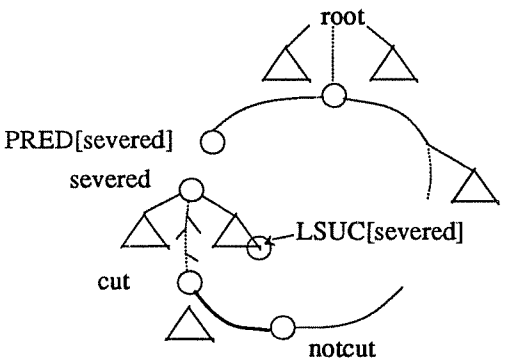


Figure 7.a: An example of $LSUC$.

Another extension aims at reducing the contention for the **price-queue**. We can dispense with locking all together, if we maintain several price-queues, one for each p_i. An additional processor is dedicated to checking these local queues periodically and selecting the best arc for p_0.

Acknowledgements

I thank R.R. Meyer for his support, J. L. Kennington for his version of NETFLO and J. Mote for NETGEN and the test problem set.

Appendix

We now present the code used to analyze the tree structure to make our claims precise. Code sections embraced by the pair **lock** and **unlock** are only accessible to one processor at a time.

code for analyzing **dual** subtrees

```
id = m_get_myid();
for ( cand = id; cand <= maxNode; cand += procs) {
  if ((nsuc[cand] ≥ 100) && (pred[cand] == artNode)) {
  /* a primal root candidate */
    ctr = 0; k = cand;
    for (i = nsuc[cand]; i; i--) { /* change in subtree */
      if (label[k]! = cand) {
        ctr++;
        label[k] = cand;
      }
      k = succ[k];
    }
    ns = nsuc[cand]/100; /* i.e. slots 50,100,200,... */
    if (ns >= 10) ns = 9;
    change = (1.0 * ctr)/nsuc[cand];
    lock;
    treect[ns]++;
    treecg[ns] += change;
    unlock;
  }
}
```

The code for analyzing primal subtrees is slightly more complicated, since we have to make sure that no tree is counted twice.
E.g., instead of

```
if ((nsuc[cand] ≥ 100) && (pred[cand] == artNode)) {
/* a primal root candidate */
```

we have

```
if ((flow[cand]) && (nsuc[cand] > 20)) { /* a dual root candidate */
  lock;
  if (not_in_queue(cand))
    enter_queue(cand)
  else
    continue; /* back to "for-loop" */
  unlock;
```

References

[CM88] M.D.Chang, M.Enqquist, R. Finkel, R.R. Meyer, "A parallel algorithm for generalized networks", *Technical Report no.* **642**, *Dept. of Computer Sciences, UW Madison,* 1987 (to appear in *Parallel Optimization on Novel Computer Architectures*)

[Co83] Stephen A. Cook, "An overview of computational complexity", *CACM* **8**, *no.***6**, 1983

[GKKN74] F. Glover, D. Karney, D.Klingman, A.Napier, "A computation study on start procedures, basis criteria, and solution algorithms for transportation problems", *Management Science* **20**, 1974

[GSS82] Leslie M. Goldschlager, Ralph A. Shaw, John Staples, "The maximum flow problem is log-space complete for P", *Theoretical Computer Science* **21**, 1982

[KNS74] D. Klingman, A. Napier, J. Stutz, "NETGEN: a program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems", *Management Science* **20** *no.* **5**, 1974

[KH82] J. Kennington, R. Helgason, "Algorithms for network flow programming", *John Wiley and Sons, New York,* 1982

[Pi79] N.J. Pippenger, "On simultaneous resource bounds", *Proc.* **20***th IEEE Symposium on Foundations of Computer Sciences, Los Angeles,* 1979

[Se84] Robert Sedgewick, "Algorithms", *Addison Wesley,* 1982

[Sm82] Stephen Smale, "On the average speed of the simplex method of linear programming", *Preprint* 1982

[SB84] "Balance^{TM} technical summary", *Sequent Computer Systems, Inc.,* 1984

[SS87] "Symmetry^{TM} technical summary", *Sequent Computer Systems, Inc.,* 1987

n = d = ω =

	1 proc	10 procs					
	1 sweep	1 sweep	2 sweeps	3 sweeps	5 sweeps	7 sweeps	10 sweeps
	it sec	it sec	it sec	it sec	it sec	it sec	it sec
1							
2							
3							
4							
5							
ave							

- 1 big row (row n-1)
- $m_{ii} = \sum_{i \neq j} m_{ij}$

n = d = ω =

	1 proc 1 sweep	10 procs					
		1 sweep	2 sweeps	3 sweeps	5 sweeps	7 sweeps	10 sweeps
	it sec	it sec	it sec	it sec	it sec	it sec	it sec
1							
2							
3							
4							
5							
ave							

- 1 big row (row n-1)
- $m_{ii} = \sum_{i \neq j} m_{ij}$

n = d = ω =

	1 proc 1 sweep	10 procs					
		1 sweep	2 sweeps	3 sweeps	5 sweeps	7 sweeps	10 sweeps
	it sec	it sec	it sec	it sec	it sec	it sec	it sec
1							
2							
3							
4							
5							
ave							

- 1 big row (row n-1)
- $m_{ii} = \sum_{i \neq j} m_{ij}$

n = d = ω =

	1 proc 1 sweep	10 procs					
		1 sweep	2 sweeps	3 sweeps	5 sweeps	7 sweeps	10 sweeps
	it sec	it sec	it sec	it sec	it sec	it sec	it sec
1							
2							
3							
4							
5							
ave							

- 1 big row (row n-1)
- $m_{ii} = \sum_{i \neq j} m_{ij}$