PARALLEL ARC-ALLOCATION ALGORITHMS OPTIMIZING GENERALIZED NETWORKS

by

R. H. Clark and R. R. Meyer

Computer Sciences Technical Report #862

July 1989

Parallel Arc-Allocation Algorithms for Optimizing Generalized Networks*

R.H. Clark and R.R. Meyer

Computer Sciences Department and Center for the Mathematical Sciences
The University of Wisconsin-Madison
Madison, Wisconsin 53706 USA

Abstract

Two parallel shared-memory algorithms are presented for the optimization of generalized networks. These algorithms are based on the allocation of arc-related operations in the (generalized) network simplex method. One method takes advantage of the multitree structure of basic solutions and performs pivot operations in parallel, utilizing locking to ensure correctness. The other algorithm utilizes only one processor for (sequential) pivoting, but parallelizes the pricing operation and overlaps this task with pivoting in a speculative manner The relative performance of these two methods (on the Sequent Symmetry S81 multiprocessor) is compared and contrasted with that of a fast sequential algorithm on a set of large-scale test problems of up to 1,000,000 arcs.

* This research was supported in part by
NSF grant CCR-8709952 and AFOSR grants AFOSR-86-0194 and 88NM458.

We will consider parallel algorithms for the efficient solution of the *generalized network flow problem*, expressed as follows:

$$\begin{array}{c} \min \quad cf \\ s.t. \quad Gf = b \\ 0 \le f \le u \end{array} \qquad \text{(GP)}$$

The matrix $G \in R^{m \times n}$ is such that each column has no more than two non-zero elements. Columns with two non-zero elements are assumed to be scaled and reflected as needed, so that one of them is 1. The other element is then a *flow multiplier*, which can be smaller or larger than 1 in magnitude. (If a flow multiplier is larger than 1 in magnitude, it generally corresponds to an arc that gains flow; such a multiplier might represent the conversion rate from dollars to yen, or the rate of interest for a savings account. If a flow multiplier is smaller than 1 in magnitude, it generally corresponds to an arc that loses flow; such a multiplier could represent transmission loss in power lines, or evaporation in irrigation canals.) A column that has just one nonzero element corresponds to a *root-arc*, an arc that is incident to just one node. Applications of generalized network flow problems are discussed in [Glover, et al, 73] and [Glover, et al, 84]. As with the pure network flow problem, which we designate as NP, the simplex algorithm for GP can be executed on a graph. One difference between NP and GP is that the graph of any basis for NP consists of one rooted tree, while the graph of a basis for GP is a forest of *quasi-trees*. A quasi-tree is a tree with one additional arc (making it either a rooted tree or a tree with exactly one cycle). Figure 1.1 shows a forest of quasi-trees.

A specialization of the primal simplex algorithm for GP is described in [Jensen and Barnes 80] and [Kennington and Helgason 80]. This algorithm utilizes the quasi-tree structure to do the ratio test, the computation of duals, and the flow update by a modified backsubstitution. A number of implementations now exist. GRNET, a sequential version of the primal simplex algorithm for GP is discussed in [Engquist and Chang 85]. The data structures of GRNET are based on those of [Adolphson 82] and [Barr, Glover and Klingman 79]. [Engquist and Chang 85] establishes that on a CYBER 170/750, GRNET is about 50 times faster than MINOS [Murtagh and Saunders 78], a standard LP code, on problems of the form GP. Another efficient code for GP called GENNET is discussed in [Brown and McBride 84]. GENNET, in addition to using state of the art data structures, tries to use integer (rather than floating point) arithmetic wherever possible. Sequential and parallel algorithms for convex network flow problems are discussed in [Zenios 86], and in [Zenios and Mulvey 85].

Any parallel simplex-based algorithm for GP developed must ensure that only one processor updates a quasi-tree at any given time. Otherwise, the tree functions may not be updated correctly. To ensure this mutual exclusion, we consider several algorithmic devices, including locking quasi-trees as needed. We have developed two distributed versions of the GRNET algorithm. The first algorithm, PGRNET, does multiple pivots in parallel. Each

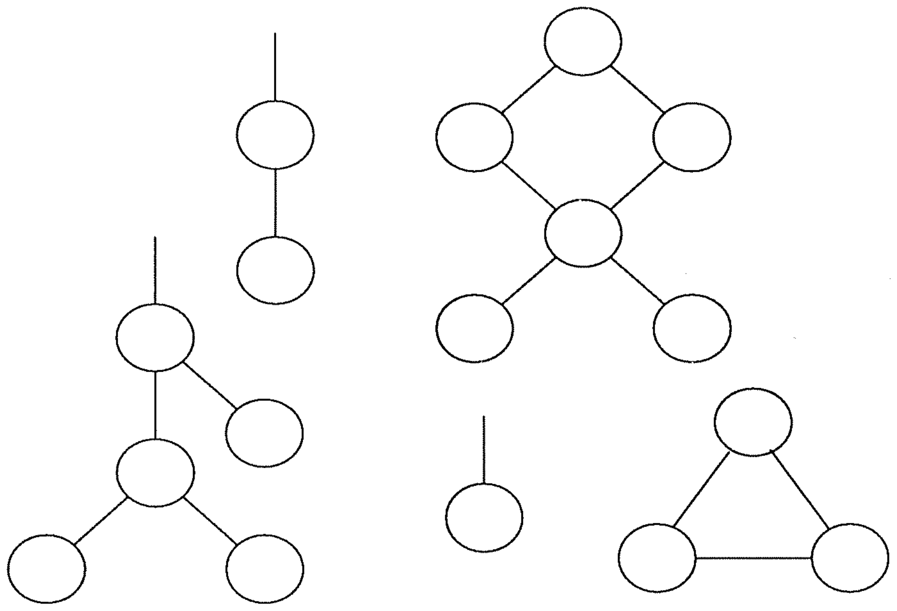

Figure 1.1 A Forest of Quasi-Trees

processor receives a subset of the arcs and computes the reduced costs of arcs in its set and executes pivots on the corresponding locked trees. All processors thus execute the same algorithmic tasks in parallel. (An earlier version of this parallel pivoting algorithm was described in [Clark and Meyer 87].) Our second algorithm, TPGRNET (for Task Parallel GRNET), *overlaps* the pricing and pivoting tasks. (In the implementation of TPGRNET, almost all of the computing time is spent in "Stage 1" (to be described below), in which most of the processors compute reduced costs, one processor selects pivot arcs, and one processor executes pivots without locking quasi-trees.) The computational results given below show that the TPGRNET algorithm outperforms PGRNET for large-grained problems, i.e. problems that have only a few quasi-trees in the optimal basis.

Our test problems are randomly generated generalized transportation problems. Problem sizes range from 10,000 nodes to 30,000 nodes and 100,000 arcs to 1,000,000 arcs. Speedups of up to 11 on 19 processors were obtained on the Sequent Symmetry S81 with these test problems.

2 The parallel algorithms PGRNET and TPGRNET.

2.1 Introduction

In section 2.2 we describe GRNET2, a modification to the sequential program GRNET of Chang and Engquist. The principle algorithmic difference between these codes is that GRNET2 uses a gradual penalty method [Grigoriadis 84] rather than the big M method. In section 2.4 we describe PGRNET, a "parallel pivot" version of the GRNET2 algorithm, and in section 2.5 we will describe a "parallel pricing, overlapped pivot" algorithm TPGRNET that was developed to solve large-grained problems not handled well by PGRNET.

2.2 GRNET2

We first give a summary of GRNET2, a sequential algorithm. Figure 2.1 gives a flow chart. In the figure, "l.t." designates *list_threshold*, a candidate list parameter.

INITIALIZATION
Set initial flows and penalty on the artificial arcs. Partition the set of arcs into roughly equal sized segments for pricing during the next stage.

STAGE 1 (*sequential pivoting with candidate lists*)
Develop a separate candidate list for each segment of the arc list. If the number of arcs stored in a candidate list is greater than some number *list_threshold*, select a pivot eligible arc (if there are any) from this list, execute the pivot and go to the next candidate list. If the number of arcs is less than or equal to *list_threshold*, scan the corresponding segment of the arc list to make a new candidate list. If it is not possible to make a candidate list from that segment with more than *list_threshold* entries and the penalty on the artificial arcs has reached its maximum value go to STAGE 2. Otherwise increase the penalty on the artificial arcs, recompute duals, and create new candidate lists and continue STAGE 1.

STAGE 2 (*verification of optimality*)
Scan the arc list for pivot eligible arcs. If a pivot eligible arc is found, then the pivot is immediately executed (no candidate list is constructed). If a complete sweep through the entire arc list can be made without finding any pivot eligible arcs, then optimality has been reached.

GRNET in its original form uses an artificial starting basis discussed in [Glover, et al, 74]. An artificial (root) arc with *big M* penalty is attached to each node, providing an initial basis of m quasi-trees with just one node and one arc per quasi-tree. Each artificial arc is given a flow that satisfies the constraint corresponding to the node. In GRNET2 we use a starting procedure motivated by the gradual penalty method (GPM) discussed

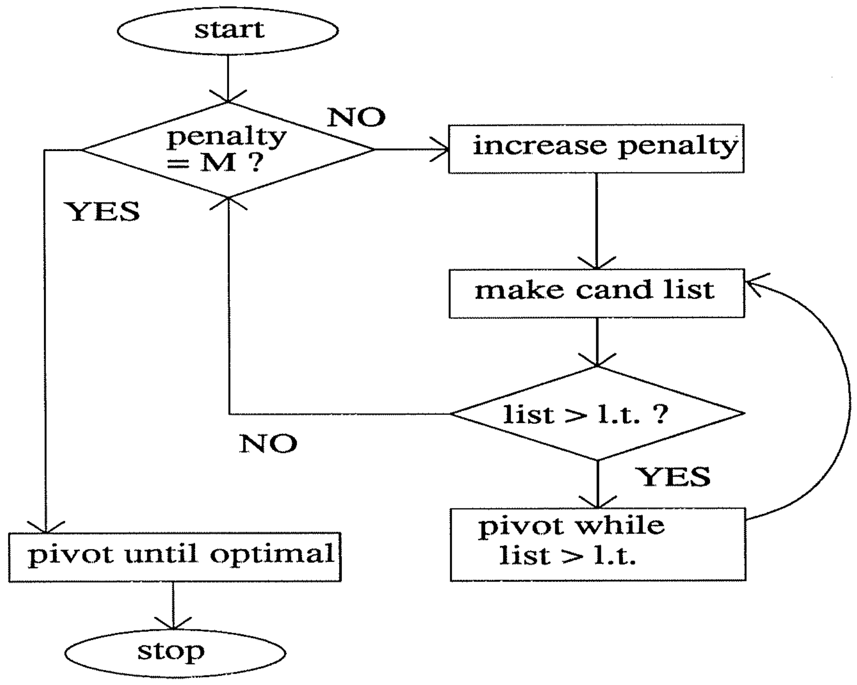

Figure 2.1 Flow Chart for Sequential Algorithm GRNET2

in [Grigoriadis 1984]. Again, a root arc with an appropriate flow is attached to each node, but the gradual penalty method gives a moderate initial penalty to the artificial arcs and then gradually increases the penalty. (As discussed below, this results in a dramatic improvement in performance in certain problem classes.) GRNET2 uses a candidate list strategy similar to that of GRNET, but computational experience motivated by parallel algorithms has shown that large-scale problems are most efficiently solved if the set of arcs is partitioned, and a separate candidate list is maintained for each subset of arcs. GRNET2 partitions the arc set into segments, and successive pivot arcs are selected from successive segments. Roughly speaking, a pivot arc is selected from a candidate list if the number of arcs remaining in the candidate list is greater than *list_threshold.* If the number of arcs in the list is less than or equal to *list_threshold*, the list is purged and pricing is done in the segment until a new candidate list is filled. If a sweep through the segment yields *list _threshold* or fewer pivot eligible arcs, the penalty is increased by popping a new value off of a stack, dual variables are recomputed, and pivots are executed until there are again *list_threshold* or fewer pivot eligible arcs in some segment. This gradual penalty method is continued until the penalty has reached its maximum (*big* M) value.

The usual motivation for using gradual penalty method is that the procedure can utilize original costs more effectively than the *big* M method. In the case of generalized network flow problems, the gradual penalty method has the added advantage that pivot arcs are initially less likely to be chosen in such a way that the outgoing arc will be an artificial arc. Under the *big* M method, the initial penalty on the artificial arcs is so large that pivots tend to be chosen essentially to reduce the flow on the artificial arcs, and therefore pivots tend to cause artificial arcs (which are root arcs) to leave the basis quickly. This tends to reduce rapidly the number of quasi-trees in the basis. Under the gradual penalty method, the penalty on the artificial arcs is more moderate, and artificial arcs are less likely to leave the basis immediately. This has the result that root-arcs remain in the basis for a longer time, and quasi-trees are more numerous and smaller than they would be under the *big M* method. Since quasi-trees are smaller, there is less work involved in executing pivots. Moreover, since quasi-trees are more numerous, algorithms like PGRNET that rely on the disjoint nature of the basis for parallelism can run more efficiently. Finally, since pivots are not chosen merely to achieve feasibility, more pivots result in advancement toward optimality, and the overall number of pivots is significantly reduced. The computational experience described in [Grigoriadis 1984] shows that the gradual penalty method can give a 15% improvement in CPU time for pure network flow problems. Our computational experience shows that our version of the gradual penalty method reduces CPU time by a factor of 13 to 29 (not 13% to 29%) for generalized network flow problems with heavy capacitation. The reduction in computing time decreases as the number of quasi-trees in the optimal basis is increased, but almost all problems can be solved more quickly by the gradual penalty method. The multiple candidate list strategy for GRNET2 was motivated by the excellent performance of PGRNET on large-scale problems. This resulted in an additional reduction, relative to the single candidate list strategy of GRNET, in sequential CPU time of up to 45% with the biggest improvement occurring for problems with more than 300,000 arcs.

2.3 Parallel algorithms

In developing a distributed version of GRNET2, we tried a number of different strategies. One strategy, a *tree allocation* strategy, involves partitioning the set of quasi-trees, and executing pivots on *local arcs* in parallel. We say that a *local arc* is an arc that is incident to two nodes belonging to the same subset of the partition. This scheme was used on the University of Wisconsin crystal multicomputer, and is discussed in [Chang, et al, 1987]. It eliminates the need for locking quasi-trees but the set of *local arcs* may be a very small fraction of the whole arc set. This means that the set of potential pivot arcs is a small fraction of the whole arc set. The scheme does not scale well because the set of *local arcs* diminishes as the number of subsets increases, but the greatest disadvantage to the scheme is that the set of quasi-trees must be periodically repartitioned so that pivot eligible *cross arcs* become *local arcs.* This is a time consuming task, and it is difficult to quickly partition the quasi-trees in such a way that each subset contains roughly the same number of nodes and each subset contains a substantial number of pivot eligible *local arcs.* Another strategy, an *arc allocation* strategy, partitions the set of arcs, rather than the quasi-trees. Pivot arcs are selected from different subsets (segments) of the partition in parallel, and all arcs are potential pivot arcs. From a locking standpoint, the tree allocation strategy is analogous to assigning "long-term" locks (thereby reducing communication costs in a loosely-coupled system) on node subsets to processors, whereas the arc allocation strategy utilizes short-term locks (in effect only for the time required to perform a single pivot) on the appropriate nodes corresponding to the selected pivot arc. Since the values of all duals must be available to all processors in order to permit the parallel pricing of all arcs (not just *local arcs*), the shared memory multiprocessor is an ideal if not a necessary architecture for the implementation of *arc allocation* algorithms.

Section 2.4 contains a brief summary of PGRNET, an implementation of a strategy that works nicely when there are many quasi-trees in the optimal basis. PGRNET does parallel pricing to find pivot arcs, locks the quasi-trees at the ends of the pivot arcs to ensure exclusive access to quasi-trees, and executes pivots in parallel. The greatest disadvantage to PGRNET is that processors must temporarily lock one or two quasi-trees before executing a pivot involving those quasi-trees. If there are only a few quasi-trees in the basis, this means that only a few pivots can be executed in parallel. PGRNET rejects pivot eligible arcs that are found to connect quasi-trees that are locked (i.e. currently in the process of being modified), so the work invested in finding these arcs is wasted. Quasi-trees are more likely to be locked if there are just a few of them. Pricing can always be done in parallel, regardless of the number of quasi-trees in a given basis, but pricing is more likely to be done with old dual values if there are few quasi-trees and they are changing in parallel. The lack of current dual information and the frequent loss of good pivot eligible arcs can greatly reduce the efficiency of PGRNET if the optimal basis has only a few quasi-trees. Another *arc allocation* strategy that attempts to solve these problems gives to one processor the task of executing all pivots, and gives to the other processors the task of selecting the pivot arcs. TPGRNET, an implementation of this strategy, will be described in more detail in section 2.5, and results for implementations of PGRNET and TPGRNET on the Sequent Symmetry S81 will be compared.

2.4 PGRNET (Parallel GRNET)

The (parallel) PGRNET algorithm can be summarized:

INITIALIZATION
In parallel, generate the initial flows on the artificial arcs. Divide the problem arcs into roughly equal-sized segments for pricing in the next stage.

STAGE 1 (*parallel pivoting with candidate lists*)
Asynchronously and in parallel scan the segments of the arc set to develop multiple candidate lists. Pivot arcs are chosen from the candidate lists, and quasi-trees are locked before pivots are made. When, for a particular segment of the arc set, it is not possible to develop a candidate list with more than *list_threshold* entries, check the penalty on the artificial arcs. If this penalty has reached its maximum value go to STAGE 2. Otherwise assign a new value to the penalty of the artificial arcs, and update the duals in parallel and continue asynchronous pivoting.

STAGE 2 (*parallel verification of optimality*)
Scan the segments of the arc list in parallel to locate pivot eligible arcs. If a pivot eligible arc is found, lock the associated quasi-trees, and execute the pivot (if the quasi-trees were successfully locked). If an entire sweep through the segments can be made without finding any pivot eligible arcs, optimality has been reached.

The name PGRNET designates parallel GRNET. Figure 2.2 gives a flow chart for this algorithm. As in figure 2.1, "l.t." designates *list_threshold*, a candidate list parameter. Arcs are divided evenly between segments. If there are n arcs and P segments, then segment 1 has arcs (1) through (n/P), processor 2 gets arcs $(n/P+1)$ through $(2n/P)$ and so forth. (A more sophisticated allocation of the non-artificial arcs might try to guess the topology of the optimal solution, and thereby assign arcs to specific partitions. If the optimal topology is known, lock contention can be reduced significantly by collecting in the same subset of the partition, all of the arcs that connect nodes in a given quasi-tree or group of quasi-trees. This idea could be used to solve perturbed problems efficiently. Given the optimal quasi-tree structure of some solved problem, subsets of the arc set could contain arcs that are local to certain collections of the optimal quasi-trees. A small perturbation of the data would hopefully change the optimal topology by very little, and therefore the initial arc allocation might improve the solution time significantly.) The dual variables of all the nodes, the predecessor threads, the successor threads and all other tree functions required by the generalized network simplex method are assumed to be stored in shared memory and are available to all processors. It is important to emphasize that only the acquisition of problem data (i.e. generating data or reading data) is done sequentially, and the solution process is entirely parallel. The number of partitions is equal to the number of processors, and all processors execute the same set of tasks. Each processor creates candidate lists, selects pivot arcs, locks quasi-trees to prevent corruption of tree structures, and executes pivots. We say that PGRNET is an example of "uniform parallelism" for this reason. The results below show that uniform parallelism is the best solution strategy for generalized

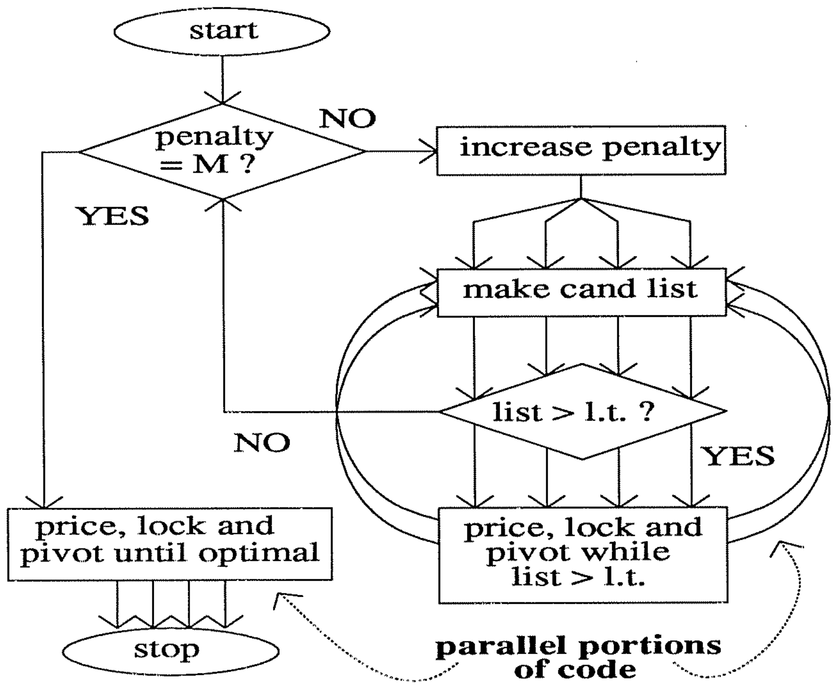
start
penalty = M ?
NO
increase penalty
YES
make cand list
list > l.t. ?
NO
YES
price, lock and pivot until optimal
price, lock and pivot while list > l.t.
stop
parallel portions of code

Figure 2.2 Flow Chart for Parallel Algorithm PGRNET

network flow problems, as long as the number of quasi-trees in the optimal solution is not too small.

The program that each processor executes is almost identical to GRNET2. During a *parallel pivoting* stage, Stage 1, each processor makes its own candidate list of pivot eligible arcs. The candidate lists are made in the same way that candidate lists are made in GRNET2. Each processor p choses its next pivot arc from its candidate list by selecting the pivot eligible arc with the greatest reduced cost in absolute value. If the quasi-trees at the ends of the arc have not been locked by another processor, p locks the quasi-trees to keep other processors from interfering with the tree update, performs the pivot and removes the arc from the candidate list. If the quasi-trees are already locked, processor p removes the arc from the candidate list and chooses another arc. The dual update part of the pivot operation has also been parallelized and this parallelization is described below. When the candidate list belonging to p has not more than *list_threshold* arcs, p develops a new candidate list. If the new candidate list also has not more than *list_threshold* entries, processor p sets a flag in shared memory to indicate that it is having difficulty finding pivot eligible arcs. This flag is checked frequently by all processors, and when it is set, processor 1 checks to see if the penalty on the artificial arcs is *big M*. If the penalty is equal to *big M*, all processors enter Stage 2. If the penalty is smaller, then the processors increment the penalty on the artificial arcs and cooperate in recomputing the dual variables. Then all processors develop new candidate lists.

Stage 2 of PGRNET corresponds to the *verification of optimality* stage in GRNET2. The verification of optimality is done in parallel, and all processors execute the same tasks.

Optimality is achieved by performing any remaining pivots. Processors sweep through their segments looking for pivot eligible arcs, but no candidate lists are developed. If processor p finds a pivot eligible arc, it locks the quasi-trees at either end of the arc, executes the pivot, and indicates to the other processors that they must restart their sweep (by setting flags in a shared array). This restart mechanism is needed because a pivot executed by processor p might cause an arc owned by another processor to become pivot eligible. If processor p finds that one of the trees at the ends of a pivot eligible arc is locked, it sets the other processors restart flags and restarts its own sweep. Each processor checks its restart flag frequently during Stage 2, and when a processor finds that its flag has been set, it marks the arc in its segment that was last priced out, and continues pricing. If the processor prices out all of its arcs up to the marked arc without finding any to be pivot eligible and without finding its restart flag to be set, that processor informs the others that none of its arcs are pivot eligible. Optimality is reached when all processors make a sweep through their arcs without finding their restart flags set, and without finding any arcs that are pivot eligible.

The dual update is performed recursively and in parallel. A processor p that is updating the dual variables in a subtree S inspects a tree function $t(\cdot)$ that specifies how many nodes are in the subtree. It then sets a shared variable *size_lim* so that $size\_lim = t(root)/nproc$ where *nproc* is the number of processors and *root* is the root node of S. (The root node of S is the usual root node if S is a rooted tree. If S is not a rooted tree, then S has a unique cycle, and *root* will be one of the nodes in the cycle.) Next, processor p traverses S and updates the dual variables in all subtrees of S that have fewer than some small number *tree_threshold* of nodes. The root nodes of the larger subtrees are put onto a shared queue. All processors that are computing reduced costs check frequently to see if there are any nodes on this queue. When a node appears on the queue, some processor q takes it off and checks to see if the associated subtree is smaller than *size_lim*. If so, then q updates all of the dual variables in that subtree. If not, then q traverses the subtree (without recomputing *size_lim*) and puts the root nodes of subtrees with *tree_threshold* or more nodes on the queue. The efficiency of this scheme relies on the subtrees of S being broad rather than deep. Ideally, p will chop S into *nproc* pieces, all having fewer than *size_lim* nodes. If this happens, then all processors will work in parallel updating the duals, and little CPU time will be spent putting root nodes onto the queue. If, however, the original subtree is badly skewed, then subtrees that are put on the queue may have to be chopped up by other processors. Despite some potentially inefficient aspects of the parallel dual update, computational experience has shown that it improves the overall efficiency of PGRNET by about 15% for certain problem classes. The parallel dual update is also used by TPGRNET, and the effects there are much more significant.

2.5 TPGRNET (Task-parallel GRNET)

We will now describe a class of problems that PGRNET handles poorly, and describe an algorithm, TPGRNET that solves these problems more efficiently. The most significant factor in the efficiency of PGRNET is the number of quasi-trees in the optimal basis. If this number is small, we say the problem is large-grained (because quasi-trees tend to be large), and if this number is large, we say the problem is small-grained (because quasi-trees tend to be small). If a problem being solved by PGRNET has large granularity, it is more likely that a processor p will find the quasi-trees at the ends of a pivot arc to be locked. If they are locked, p must reject the arc and look for another in its candidate list. By the time the quasi-trees are freed, often all of the best arcs from the candidate list have been rejected and a pivot is executed on an arc with a relatively small reduced cost. This tends to increase the total number of pivots required to reach optimality. Candidate lists are exhausted quickly because many arcs that are chosen turn out to connect locked quasi-trees. This increases the total amount of time that is spent developing candidate lists. For these reasons, large-grained problems are solved inefficiently by PGRNET.

Some alternatives to rejecting arcs with an end in a locked quasi-tree are to store them in a temporary stack, or simply to wait until the quasi-tree becomes available. Our experience indicates that these alternatives are less efficient than the algorithm we are using for PGRNET, which simply discards these arcs.

We have found that a better approach to solving large-grained problems is to eliminate the need for the locking operation by parallelizing candidate list development and prioritization, but doing pivots sequentially (and concurrently with the development of candidate lists). Since different processors perform different tasks and execute different code, we say that TPGRNET is an example of "specialized parallelism." The algorithm is divided into two main stages. During Stage 1, the tasks are distributed as described above, and Stage 2 is exactly the same as Stage 2 of PGRNET. Figure 2.3 illustrates the flow of information during Stage 1, and figure 2.4 gives a flow chart for TPGRNET. (In both of these figures, dotted arrows indicate the direction of the flow of information.)

The (parallel) TPGRNET algorithm is:

INITIALIZATION

In parallel, generate the initial flows on the artificial arcs. Divide the problem arcs into roughly equal-sized segments for pricing during the next stage. (Same as the INITIALIZATION stage of PGRNET)

STAGE 1 (*parallel candidate list development overlapped with sequential pivoting*)

A set of candidate lists is developed and prioritized in parallel. This process is continued during the pivot, which concurrently modifies some of the duals being used in candidate list development. When the pivot is completed, the next arc to enter the basis is selected by using the "best" arc from the candidate list (if this arc has a sufficiently good reduced cost) or a different arc if this is not possible. The latter case occurs very infrequently, and under conditions to be described below, may trigger an increase in the penalty cost or an exit to stage 2.

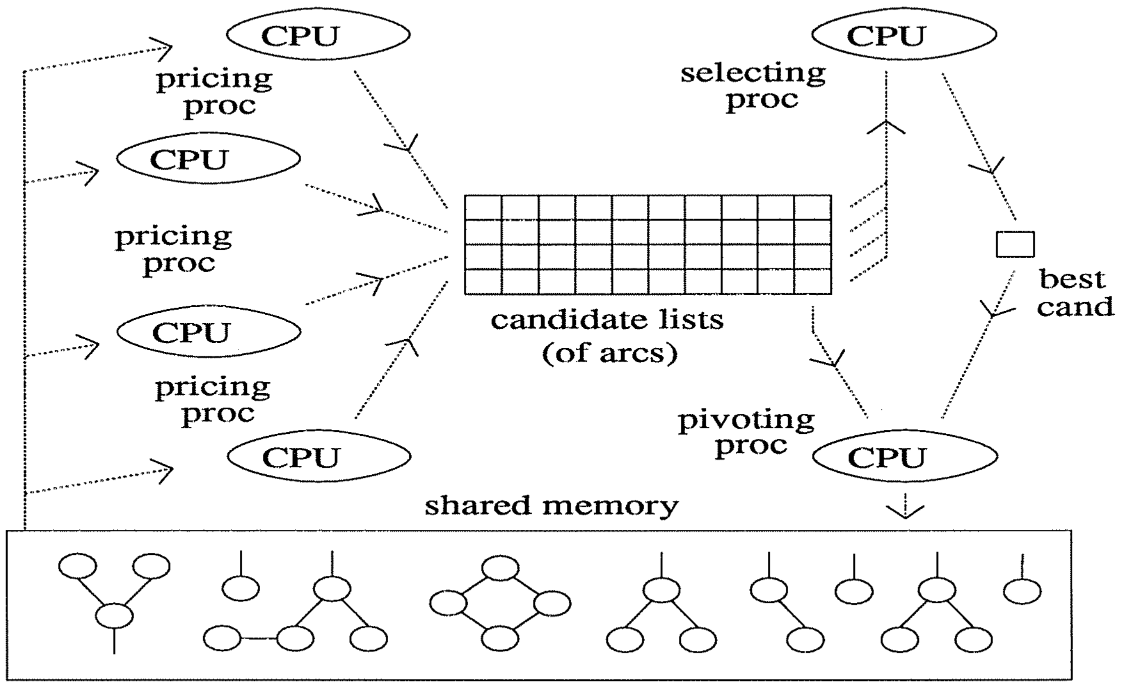

Figure 2.3 Flow of Information in TPGRNET, Stage 1

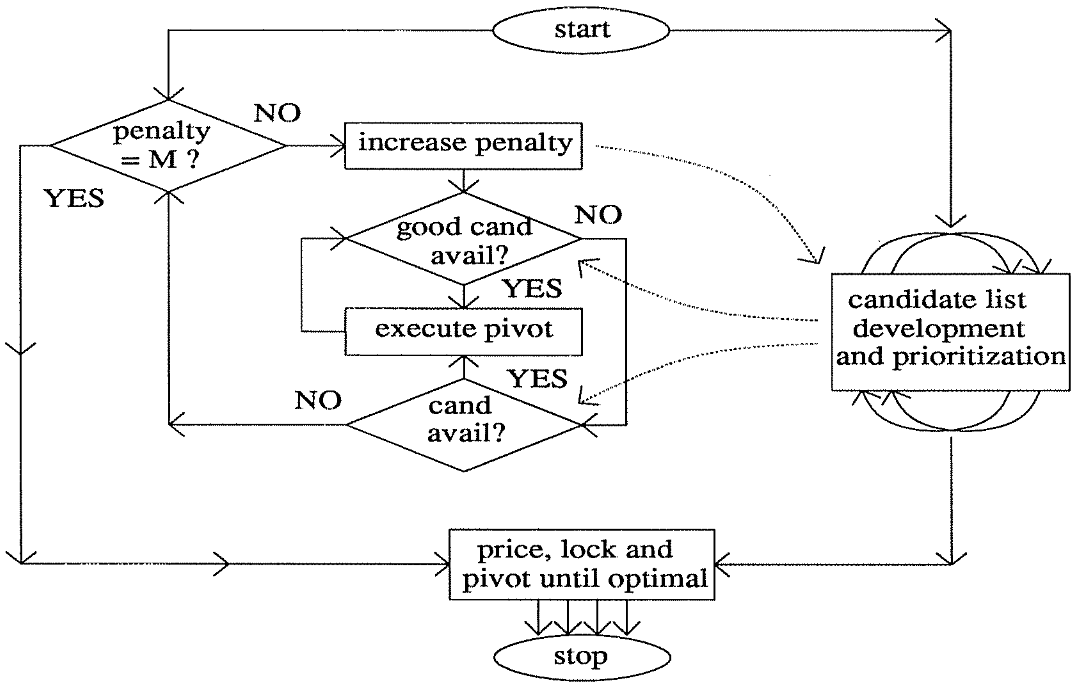

Figure 2.4 Flow Chart for Parallel Algorithm TPGRNET

STAGE 2 (*parallel verification of optimality*)

Scan the segments of the arc list in parallel to locate pivot eligible arcs. If a pivot eligible arc is found, lock the associated quasi-trees, and execute the pivot (if the quasi-trees were successfully locked). If an entire sweep through the segments can be made without finding any pivot eligible arcs, optimality has been reached. (Same as STAGE 2 of PGRNET)

We will now describe in detail the tasks performed by the individual processors during Stage 1 of TPGRNET. The pricing processors have the task of computing reduced costs and storing pivot eligible arcs (in shared candidate lists of length 10). When processor p finds a pivot eligible arc, it recomputes the reduced cost of the first element in its candidate list to see if the new arc has a larger reduced cost in absolute value. If it does, the new arc number gets stored in the first element of the array, and the previous entry is overwritten. Experience has shown that saving the previous entry yields no improvement in efficiency. If the new arc has a smaller reduced cost than the first arc in the candidate list, the new arc gets stored at a random location in the list. The pricing processors stay in a loop that includes three operations. First, there is the pricing operation. This uses most of the processor's CPU time. Second, there is a check to see if the pivoting processor has put a subtree on the dual-update queue (because of space limits this is not shown in the figures). Third, there is a check to see if Stage 1 of the algorithm has finished.

The pivot selecting processor has the task of scanning the collection of candidate lists of the pricing processors to locate the pivot eligible arc with the largest reduced cost and storing that arc in a single shared variable called *best_cand*. This processor stays in a loop that has three operations. First, the processor checks to see if *best_cand* is empty. If so, the processor looks in the first entry of some candidate list to find an arc to put in *best_cand*. Second, the processor traverses the candidate lists to see if there is a pivot eligible arc that has a reduced cost larger than the arc in *best_cand*. Third, there is a check to see if Stage 1 of the algorithm has finished.

The pivoting processor stays in a loop in which it selects pivot arcs for itself (as described below), executes pivots, and directs the increases in the penalty on the artificial arcs. Whenever possible, the pivoting processor selects its pivot arc from *best_cand*, but before adopting an arc from *best_cand*, a check is made to see that the arc is pivot eligible and to see if the reduced cost is sufficiently large in absolute value. If the arc in *best_cand* has a small reduced cost, or if there is no arc in *best_cand*, the pivoting processor looks at the first entry of each of the candidate lists to find a pivot eligible arc. If a pivot eligible arc is found, the pivot is executed. If no pivot eligible arc is found, then either the penalty on the artificial arcs is increased, or Stage 2 is begun (if the penalty has reached *big M* and cannot be increased). The pivoting processor has the task of directing the parallel update of dual variables during the execution of pivots and after the penalty on the artificial variables has been increased. During both of these operations, the pivoting processor can put the root node of subtrees onto the dual update queue, and the pricing processors will then adopt the task of updating the duals on those subtrees.

2.7 Comparison with other work.

NETPAR is a parallel network simplex code for the pure network flow problem discussed in [Peters 1988a]. The code uses a task partitioning algorithm in which one processor executes all pivots, and all other processors compute reduced costs and store pivot eligible arcs in shared a queue. This scheme is modified in [Peters 1988b] to give to one processor the task of executing pivots, to another processor the task of selecting arcs, and to all other processors the task of computing reduced costs. The modified code, PARNET, compares favorably with NETFLO [Kennington and Helgason, 1980], a standard sequential code for the regular network flow problem. PARNET, when run on three processors, is 10 to 20 times faster than NETFLO for a collection of problems generated by NETGEN [Klingman, et al, 74]. PARNET yields linear speedups for a variety of large scale problems generated by NETGEN.

PARNET and TPGRNET have a similar distribution of tasks. Both programs have one pivoting processor, one pivot selecting processor, and both programs give the task of pricing (i.e. computing reduced costs) to the other processors. However, the two programs have slightly different pivot selection strategies. Under the PARNET strategy, the pricing processors develop queues of pivot eligible arcs. When a pricing processor finds a pivot eligible arc, it puts it at the head of the queue. The pivot selecting processor looks at the first three elements in a queue when selecting an arc for the pivoting processor. Under the TPGRNET strategy, a pricing processor compares the reduced costs of the newly found pivot eligible arc with the reduced cost of the first arc on its candidate list. If the new arc has a larger reduced cost it is put at the beginning of the list, and the arc that was previously at the beginning of the list is overwritten. Otherwise, the newly found arc is stored at a random location elsewhere in the list. This strategy has been found to be somewhat faster, for generalized network flow problems, than the PARNET strategy.

Since the distribution of tasks is very similar in PARNET and TPGRNET, and since PARNET yields a linear speedup for almost all test problems, one would hope that the speedups from TPGRNET would also be linear. The results given in section 3 show that TPGRNET does not yield linear speedup for all test problems. The difference in the performance of the two algorithms is possibly due to differences in the nature of the two algorithms. PARNET does the ratio test and the flow update with integer operations, while TPGRNET does these operations in double precision. This means that pivots are, on the average, more expensive for TPGRNET than they are for PARNET. PARNET computes reduced costs with integer arithmetic while TPGRNET computes reduced costs in double precision. Experience has shown that the dual variables in large grained generalized network problems change rather frequently. A change in the dual variables after the execution of a pivot can make the arcs stored in a candidate list no longer pivot eligible, and the CPU time that was spent finding the arcs is wasted. It's possible that TPGRNET spends more time doing pricing with incorrect duals than PARNET, although further studies are needed to verify this possibility.

GENFLO [Kennington and Muthukrishnan 88] is a parallel generalized network program written for the Sequent Balance 21000 and the Sequent Symmetry S81. GENFLO does not scale to force one element 1 in each column. A sequential version of GENFLO is competitive with NETFLO [Kennington and Helgason 80] and GENNET [Brown and McBride 84] for pure network problems, and it is competitive with GENNET for generalized network flow problems. The parallel version is similar to PGRNET in its distribution of tasks. It was tested with a group of problems generated by GNETGEN, a modification of NETGEN [Klingman, et al, 1974]. GENFLO speedups ranged from about 2 to 3 (on 8 processors) for these problems. In a forthcoming joint paper [Clark, et al, 89] we provide a comparison of the performances of these approaches on the GNETGEN test problems as well as the MAGEN test problems described below.

3 Computational results

3.1 Code parameters.

We will now give some more details of GRNET2, our modified version of GRNET, and we will give the code parameters used in our runs. After the starting basis has been generated, GRNET2 solves the problem in two main stages. In STAGE 1, the processor generates *num_pricers* candidate lists of pivot eligible arcs of length *listsize*. In our implementation, *num_pricers* = 19 (this value appeared best for transportation problems; a smaller value was better for transshipment problems) and *listsize* = 60. A candidate list is created by sweeping through a segment of the arc list and adding pivot eligible arcs to the candidate list as they are found. When the sweep is completed or the list has *listsize* entries, the processor can begin pivoting. Successive pivot arcs are selected from different candidate lists. GRNET2 cycles through its collection of *num_pricers* lists and chooses a pivot arc from a different list each time. The arc that is chosen from a list is the pivot eligible arc that has the largest reduced cost in absolute value. The processor executes the pivot, removes the arc from the list and looks for another pivot eligible arc in the next candidate list. If there are no pivot eligible arcs in a list, or if the list has *list_threshold* or fewer entries (where *list_threshold* = 30), then a new list is made. If, in the process of making a new candidate list, *list_threshold* or fewer pivot eligible arcs can be found, then the penalty on the artificial arcs is checked. If the penalty on the artificial arcs is smaller than *big M*, then the penalty is increased, and duals are recomputed. If the penalty on the artificial arcs is equal to *big M*, then STAGE 2 is begun. In STAGE 2, optimality is achieved by sweeping through the entire list of arcs and pivoting on any that are pivot eligible. Optimality is verified when the processor can sweep through all arcs, finding none that are pivot eligible. The code parameters *listsize* and *list_threshold* are the same for PGRNET as for GRNET. Both PGRNET and TPGRNET use the same value for *tree_threshold* (used in parallelizing the dual update), namely 5. Assuming that the arc costs are in the range [1,100], the initial penalty on the artificial arcs is set at 20, and the initial increment between penalties is 5. Later, the increment increases to 10 and then to 20. The last increment between penalties changes the penalty from 200 to *big M*, and *big M* in our implementation is 9,999,999.

3.2 The problem generator.

Our test problems are generated randomly with a generator similar to GTGEN [Chang et. al. 1986]. The problems are bipartite transportation problems with multipliers ranging from (0.90) to (0.98). The generator has the feature that the user may specify, roughly, the number of quasi-trees in the optimal basis. This is accomplished with a technique discussed in [Chang et. al. 1986]. A processor first selects a source node and, in two steps, generates all of the arcs incident to that node. In the first step, the processor generates all of the arcs that will begin at that node and end at one of the sink nodes. As these arcs are generated, the divergences at the source and sink nodes of the arcs are adjusted so that a feasible flow exists. In the second step, the processor determines whether or not a generalized root arc will be generated for the source node. To do this, it checks a randomly generated number, and if this number is smaller than a user specified parameter $\alpha \leq 1$, two things are done. First, a generalized root arc for the source node is generated. Second, the divergence at the node is adjusted so that it equals the sum of the capacities of the outgoing arcs. This tends to force the generalized root arc into the optimal basis, creating a quasi-tree. The number of quasi-trees in the optimal basis will be roughly $\alpha \cdot (num\_sources)$. By adjusting α, one can specify, approximately, the number of quasi-trees in the optimal basis.

We measure the parallelism of our algorithms by calculating the speedup for various problems. For a given problem, we define the speedup for P processors in the following way:

$$\text{speedup} = \frac{\text{CPU time required by GRNET2}}{\text{CPU time required by } P \text{ processors}}$$

In our work, we use the Sequent Symmetry S81, a tightly coupled system. The configuration that we are using has 20 processors, each with a Weitek 1167 floating-point accelerator. The time sharing system allows the user to request up to 19 processors for the execution of a program. Sequent provides a parallel programming library that includes commands for forking processes, locking shared variables and killing processes. The DYNIX operating system on the Sequent Symmetry S81 provides the user time and the system time used in the execution of a program. Also, the system time reported can depend on the number of users on the system. For these reasons, we report only the user time as the CPU time.

3.3 Problem organization and problem sizes.

Our problems are divided into four groups. Groups 1 through 3 have 10,000 nodes, and Group 4 contains problems with 30,000 nodes. The problems in Group 1 have about 100,000 arcs, the problems in Group 2 have about 200,000 arcs, and the problems in Group 3 have about 400,000 arcs. All problems in Group 4 have 30,000 nodes and more than 300,000 arcs. All problems were solved by both PGRNET and TPGRNET, and each group contains problems with varying numbers (ranging from 1 to 7376) of quasi-trees in the optimal basis.

3.4 Computational Results.

Figure 3.1 shows a collection of speedup graphs for the Group 1 problems listed in Table 3.1. All problems have 10,000 nodes and more than 100,000 arcs. Each column in the table corresponds to a different problem, and the problem number is given at the top of the column. The last two digits of the problem number are $\alpha \times 100$, where α is the quasi-tree parameter for the generator described above. Since α is different for each of these problems, each problem has a different number of quasi-trees in its optimal basis. The graphs show speedup as a function of the number of processors for the algorithm PGRNET. According to the table, problem 1.50 is the one with the greatest number of quasi-trees in this group of problems, and both the table and the graph show that this is the problem for which PGRNET yields the best speedup (even though the sequential time for this problem is the smallest of the group). The great amount of pivots and CPU time required to solve problem 1.00 and the fact that PGRNET yields a maximum speedup of only 3.2 for this problem indicate the need to develop an alternate algorithm (like TPGRNET) for large grained problems. However, the speedups for all other problems are greater than 5.3 (on 19 processors). This means that PGRNET yields a very substantial reduction in wall clock time for a fairly wide range of problems. Also notice that, except for problem 1.00, the graphs in figure 3.1 have a slope indicating that the speedup has not reached its maximum at 19 processors. This means that the wall clock time might be reduced further by increasing the number of processors beyond 19.

Using 19 processors, PGRNET solved each one of the problems in Group 1 with more pivots than GRNET2. This may be due to the fact that some of the best pivot arcs from candidate list are rejected, since an arc is removed from a candidate list in PGRNET whenever it is found to have an end in a locked quasi-tree. The result is that PGRNET makes poorer choices for pivot arcs.

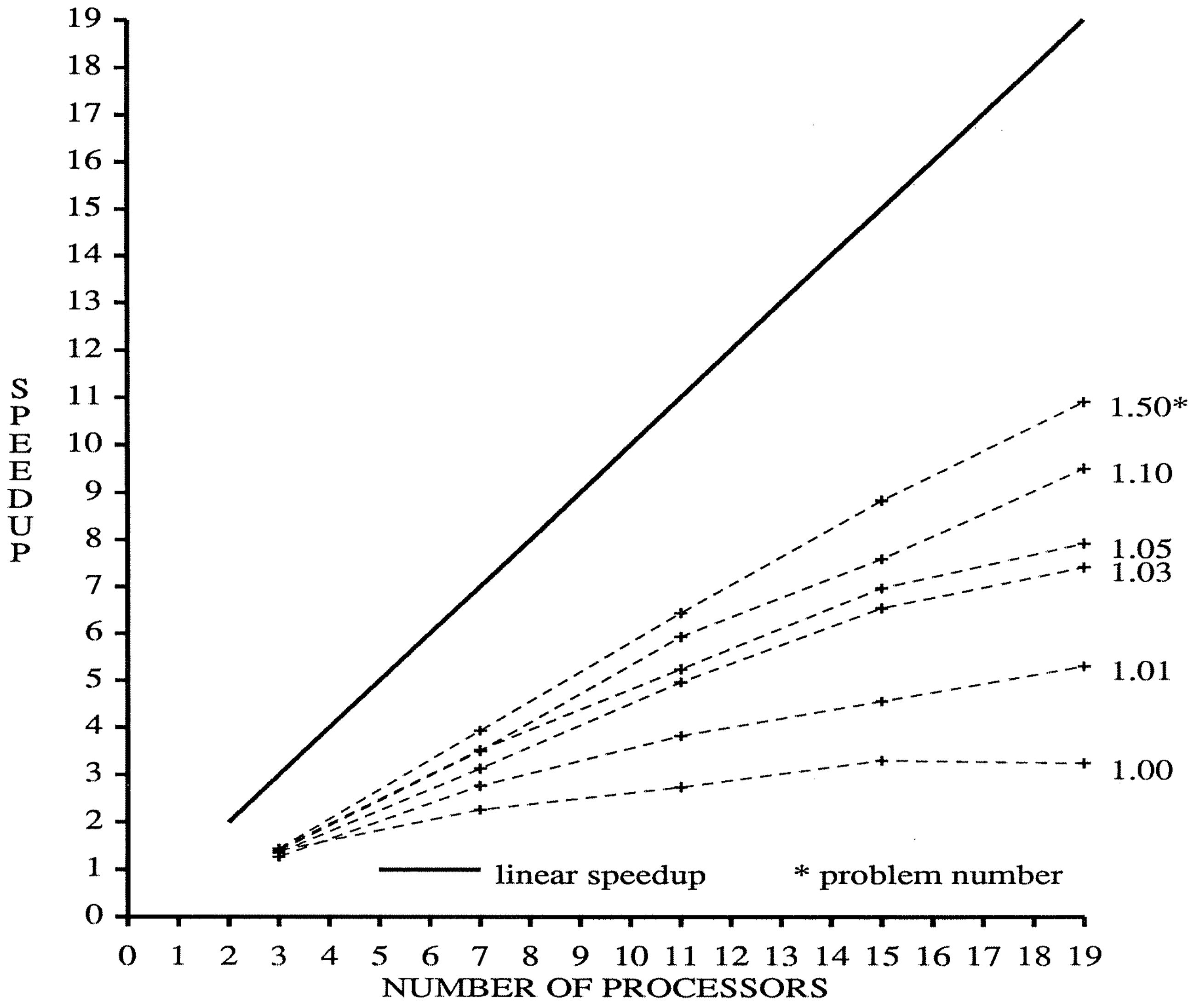

Figure 3.1 Speedups for PGRNET

Problem #	1.00	1.01	1.03	1.05	1.10	1.50
# qtrees at optimality	1	43	147	258	494	2,467
# nodes	10,000	10,000	10,000	10,000	10,000	10,000
# arcs	107,290	107,333	107,437	107,548	107,784	109,757
# pivots sequential	101,630	108,359	101,717	97,875	92,407	75,458
# pivots 19 procs	111,843	115,680	106,496	103,064	96,075	76,367
CPU secs. sequential	2,796	1,974	1,076	833	638	404
CPU secs. 19 procs	864	372	145	105	67	37
Speedup 19 procs	3.2	5.3	7.4	7.9	9.5	10.9

Table 3.1 Results for PGRNET for Group 1

Figure 3.2 and table 3.2 give results for TPGRNET for the Group 1 problems. Note that TPGRNET solves all problems with fewer pivots than both GRNET2 and PGRNET, and uses substantially fewer pivots to solve the large grained problems. Profiles of TPGRNET and GRNET2 have indicated that TPGRNET uses 68% of its CPU time doing pricing (i.e. computing reduced costs), while GRNET2 spends only 12% of its CPU time on pricing. The heavy emphasis that TPGRNET puts on pricing apparently results in better choices for pivot arcs and ultimately results in a smaller number of pivots needed for solution. Note that GRNET2 solves problem 1.00 in 2796 seconds and problem 1.50 in 404 seconds. This means that the solution time for problem 1.00 is about 6 times greater than the solution time for problem 1.50. However, the number of pivots needed for problem 1.00 is only about 4/3 times the number of pivots needed for problem 1.50. The difference in CPU time relative to the number of pivots is explained by the fact that the quasi-trees developed during the solution of problem 1.00 are larger than the quasi-trees for 1.50. Therefore, the pivots used to solve problem 1.00 are, on average, much more expensive than the pivots used to solve problem 1.50. Looking at the speedup graphs for these problems, one sees that the behavior of TPGRNET is fairly uniform for these problems. The speedup on 19 processors for TPGRNET ranges from a minimum of 3.5 to a maximum of 6.2, while the speedup for PGRNET ranges from a minimum of 3.2 to a maximum of 10.9. The shape of the speedup graphs for the problems in this group indicates that the maximum speedup for TPGRNET was reached when TPGRNET was run on about 15 processors.

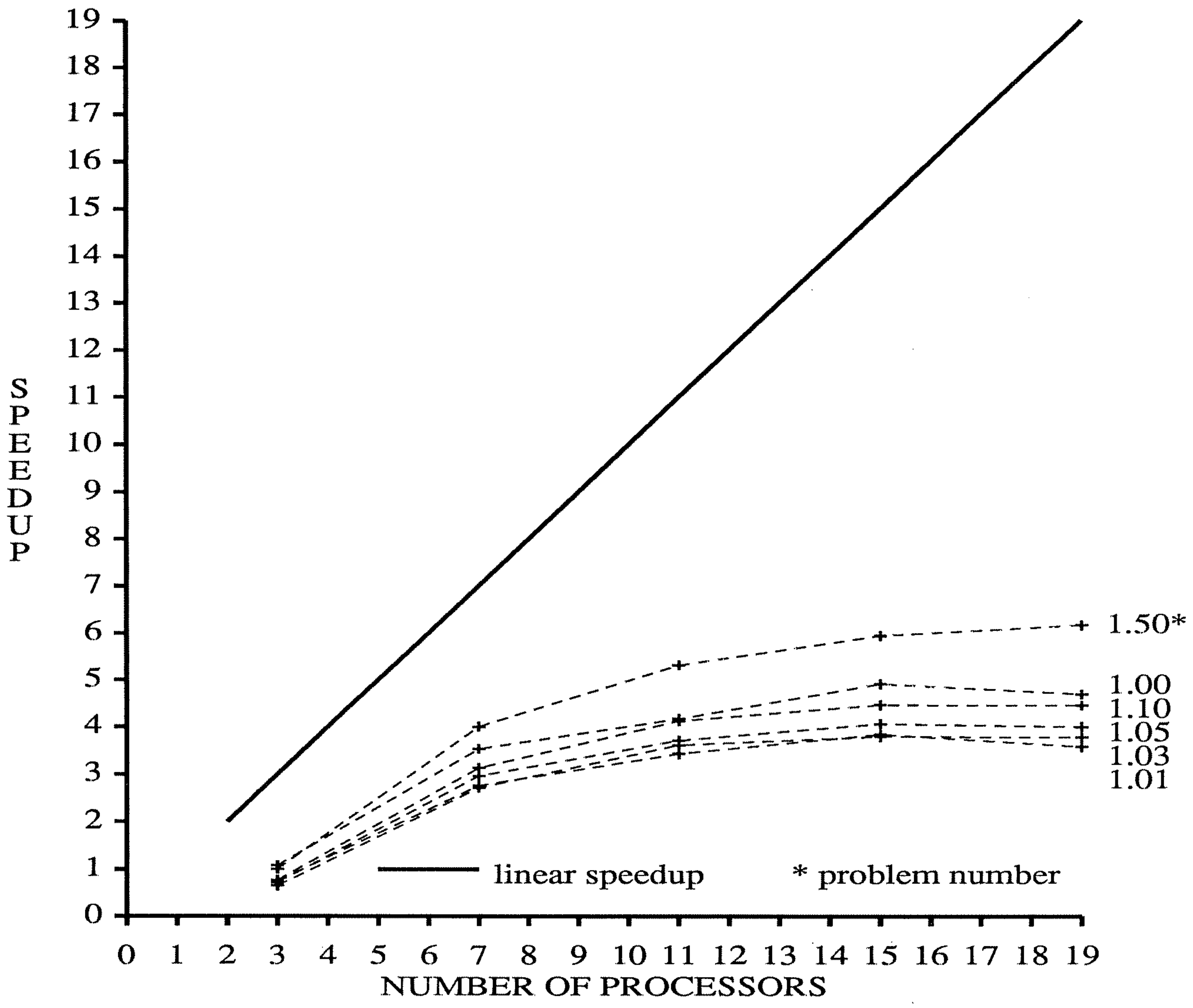

Figure 3.2 Speedups for TPGRNET

Problem #	1.00	1.01	1.03	1.05	1.10	1.50
# qtrees at optimality	1	43	147	258	494	2,467
# nodes	10,000	10,000	10,000	10,000	10,000	10,000
# arcs	107,290	107,333	107,437	107,548	107,784	109,757
# pivots sequential	101,630	108,359	101,717	97,875	92,407	75,458
# pivots 19 procs	88,144	95,798	92,026	90,292	86,534	73,775
CPU secs. sequential	2,796	1,974	1,076	833	638	404
CPU secs. 19 procs	595	551	284	208	143	65
Speedup 19 procs	4.6	3.5	3.7	4.0	4.4	6.2

Table 3.2 Results for TPGRNET for Group 1

Figures 3.3 - 3.4 and tables 3.3 - 3.4 present results for PGRNET and TPGRNET for Group 2, a set of problems having 10,000 nodes and more than 200,000 arcs and for Group 3, where the number of arcs is increased to more than 400,000. The speedup for PGRNET and TPGRNET improves slightly when the number of arcs is increased, but otherwise the results are similar to those of Group 1, except that for TPGRNET for Group 3 the maximum speedup doesn't seem to have been reached when 19 processors are used. This is due to the fact that the Group 3 problems have many more arcs than the problems in Groups 1 and 2, relative to the number of nodes. So more processors are "needed" to do pricing. (We note that each pricing processor in TPGRNET can each compute reduced costs for only about 9 arcs between pivots at the beginning of the run. This happens because the quasi-trees in the starting basis consist of just one node and one generalized root-arc, and therefore pivots are executed by the pivoting processor very quickly. Assuming that there are 17 pricing processors, the total number of arcs priced out between pivots in TPGRNET is initially only about 153, regardless of the number of arcs in the problem. This represents a very small fraction of all of the arcs, especially for the problems in Group 3. For small grained problems the pricing processors can compute only about 10 reduced costs between pivots at the end of the run. For the large grained problems, the pricing processors can each compute more reduced costs between pivots, but only rarely are 17 processors able to price out as many as 10% of all the arcs between pivots.)

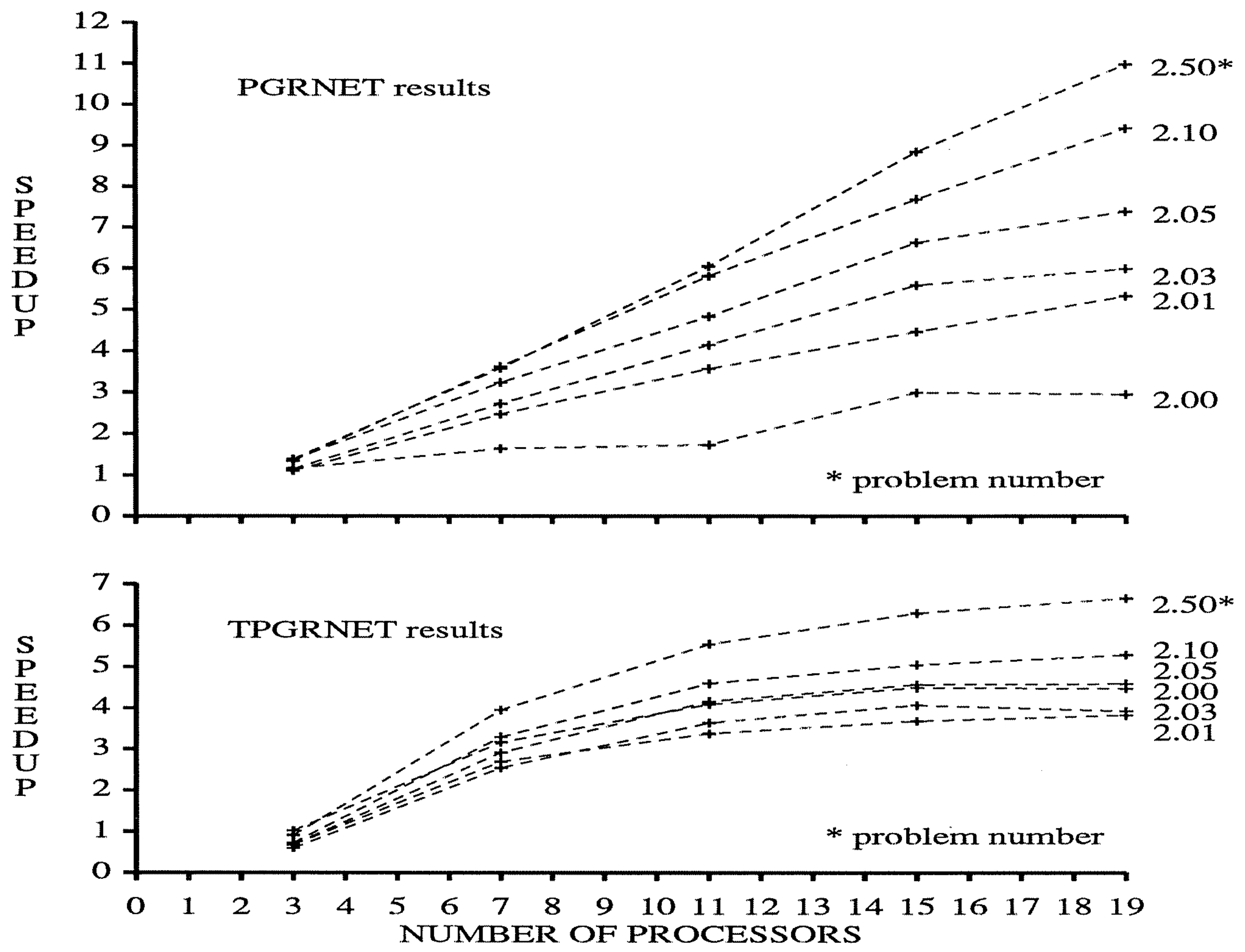

Figure 3.3 Speedups for PGRNET and TPGRNET

Problem #	2.00	2.01	2.03	2.05	2.10	2.50
# qtrees at optimality	1	46	133	233	508	2,490
# nodes	10,000	10,000	10,000	10,000	10,000	10,000
# arcs	212,627	212,673	212,760	212,860	213,135	215,117
# pivots sequential	185,105	195,430	184,746	180,115	166,392	138,635
# pivs 19 procs PGRNET	205,149	212,594	199,859	190,724	174,818	140,260
# pivs 19 procs TPGRNET	156,885	169,054	165,166	160,899	153,504	134,183
CPU secs. sequential	5,003	3,470	2,007	1,703	1,201	765
CPU: 19 procs PGRNET	1,700	652	336	230	127	69
CPU: 19 procs TPGRNET	1,118	909	512	371	227	115
Speedup PGRNET	2.9	5.3	5.9	7.4	9.4	11.0
Speedup TPGRNET	4.4	3.8	3.9	4.5	5.2	6.6

Table 3.3 Results for PGRNET and TPGRNET for Group 2

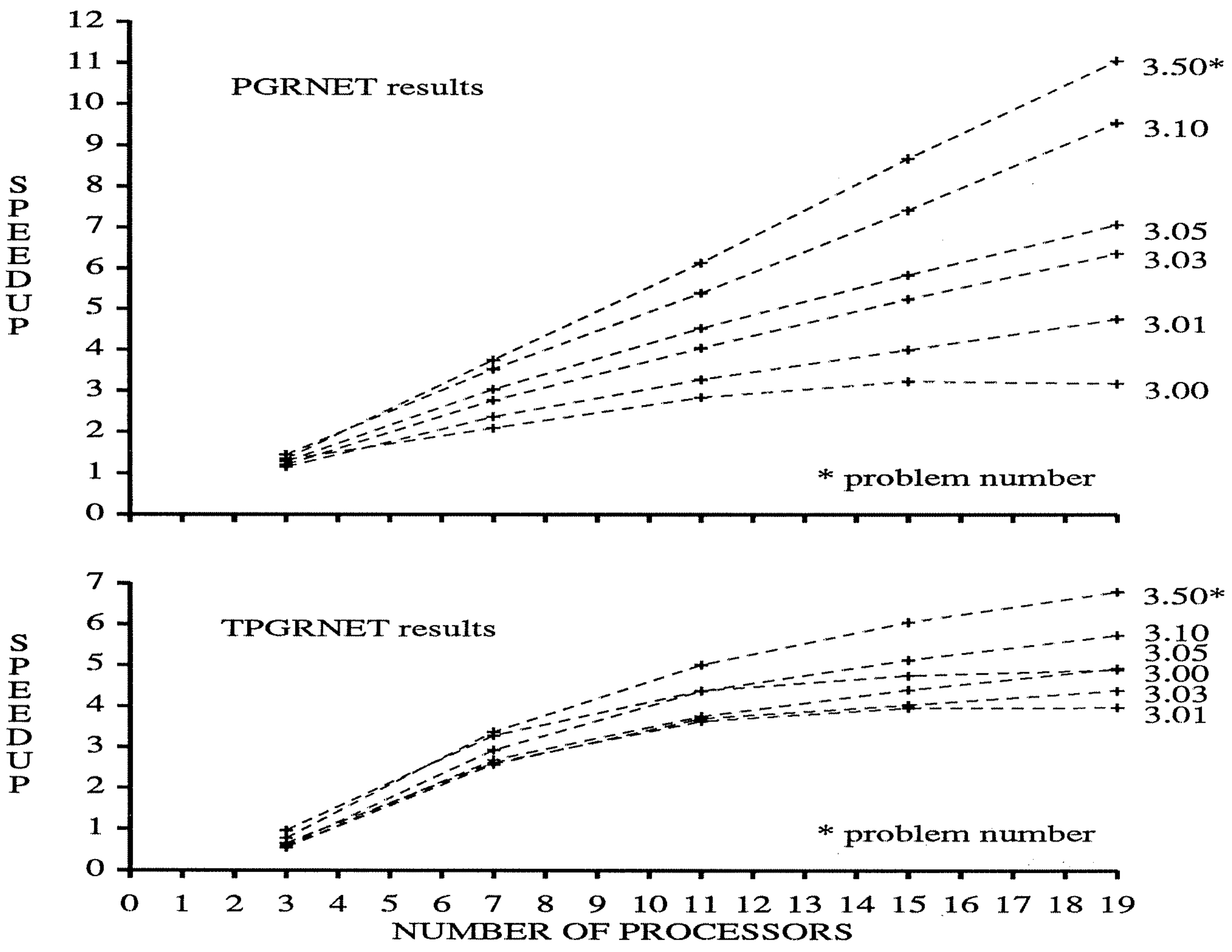

Figure 3.4 Speedups for PGRNET and TPGRNET

Problem #	3.00	3.01	3.03	3.05	3.10	3.50
# qtrees at optimality	1	41	142	235	487	2,560
# nodes	10,000	10,000	10,000	10,000	10,000	10,000
# arcs	417,397	417,438	417,539	417,632	417,884	419,957
# pivots sequential	340,153	373,375	347,419	333,389	312,852	260,266
# pivs 19 procs PGRNET	378,806	415,128	379,045	362,048	330,367	264,381
# pivs 19 procs TPGRNET	283,695	314,301	304,675	295,536	284,233	250,043
CPU secs. sequential	9,124	7,814	4,337	3,444	2,578	1,531
CPU: 19 procs PGRNET	2,877	1,649	682	488	270	138
CPU: 19 procs TPGRNET	1,867	1,966	989	700	450	225
Speedup PGRNET	3.1	7.7	6.3	7.0	9.5	11.0
Speedup TPGRNET	4.8	3.9	4.3	4.9	5.7	6.8

Table 3.4 Results for PGRNET and TPGRNET for Group 3

Figure 3.5 and table 3.5 present results for the two parallel algorithms for Group 4, a set of problems with 30,000 nodes and about 320,000 arcs. Most problems give a speedup of more than 6 for PGRNET. Only problem 4.00 has a speedup as low as 3.8. The ratio of the number of arcs to the number of nodes is roughly the same as for Group 2, but the speedups for Group 4 seem to be higher. This is probably due to the greater number of quasi-trees in the optimal bases of the Group 4 problems. Since the Group 4 problems have more quasi-trees, contention between processors is reduced during runtime. This suggests that the results might improve even further if the problems were made even larger. For problem 4.00, TPGRNET yields a speedup of 6.6 on 19 processors. This suggests that TPGRNET solves extremely large-grained problems much more efficiently than PGRNET.

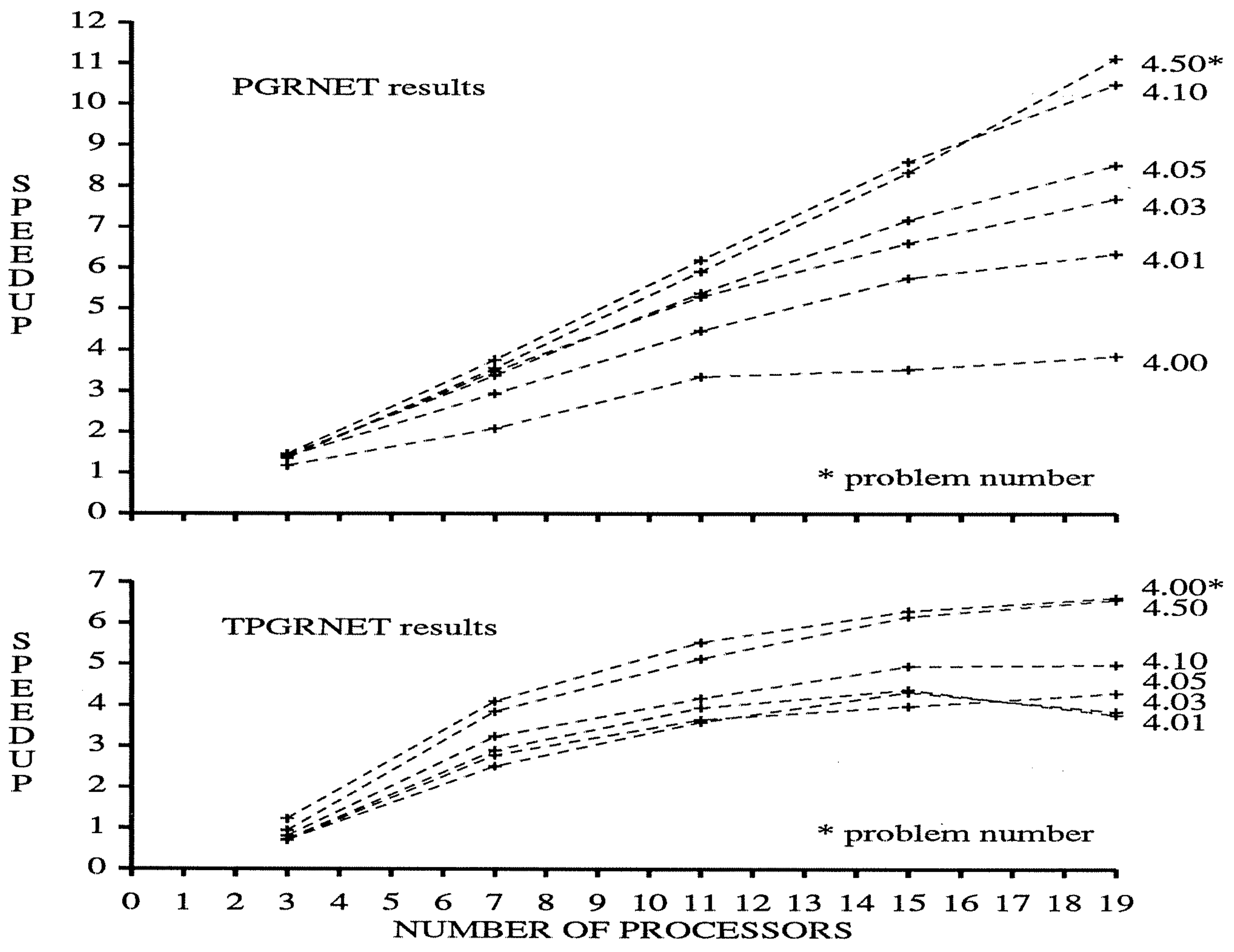

Figure 3.5 Speedups for PGRNET and TPGRNET

Problem #	4.00	4.01	4.03	4.05	4.10	4.50
# qtrees at optimality	1	139	459	776	1,490	7,376
# nodes	30,000	30,000	30,000	30,000	30,000	30,000
# arcs	322,289	322,428	322,748	323,065	323,779	329,665
# pivots sequential	328,711	347,420	320,937	308,284	288,856	228,819
# pivs 19 procs PGRNET	365,633	383,483	345,325	326,323	300,496	231,507
# pivs 19 procs TPGRNET	265,774	305,979	288,279	272,322	263,411	221,544
CPU secs. sequential	22,434	9,679	4,525	3,096	2,340	1,388
CPU: 19 procs PGRNET	5,829	1,527	589	364	223	124
CPU: 19 procs TPGRNET	3,390	2,571	1,177	721	470	211
Speedup PGRNET	3.8	6.3	7.6	5.2	10.4	11.1
Speedup TPGRNET	6.6	3.7	3.8	4.2	4.9	6.5

Table 3.5 Results for PGRNET and TPGRNET for Group 4.

3.4 Other measures of performance

Figure 3.6 and table 3.6 further help to explain the behavior of PGRNET. We say that a "collision" occurs when a processor tries to lock a quasi-tree that has already been locked by another processor. Table 3.6 gives the number of pivots that PGRNET uses to solve the problems in Group 1, along with the number of collisions that occur during the solution of each problem. The 19 processor version of PGRNET had 1,044,340 collisions for problem 1.00 and it solved this problem with 111,843 pivots. This means that there were more than 9 collisions per pivot for this problem, and this can explain the poor speedup results from PGRNET for this problem. The 3 processor version had only about 1/10th as many collisions, because there were fewer processors to compete with each other. For problems 1.03, 1.05 and 1.50, PGRNET had fewer collisions than pivots, and the number of collisions decreased as the granularity of the problem and the number of processors were decreased. The ratios of collisions to pivots for these problems are listed in table 3.9 and graphed in graph 3.9.

One way to measure the effectiveness of the TPGRNET algorithm is to count the number of times that pivot arcs are chosen from *best_cand* and the number of times that pivot arcs are chosen from the candidate lists. This gives a measure of the effectiveness of the TPGRNET pricing scheme, which overlaps pricing and pivoting, and dedicates all but two processors to pricing. Table 3.7 gives this data for the 3 and 19 processor versions of TPGRNET solving the problems in Group 1. Figure 3.7 gives the percent of pivot arcs that are taken from *best_cand* (the variable in shared memory in which the pivoting processor attempts to find its pivot arcs). The 3 processor version takes a slight majority of its pivot arcs from *best_cand.* The 19 processor version takes a vast majority of pivots from *best_cand.* This is because more pricing can be done by the 17 pricing processors, and there is thus a high probability that the pivoting processor will find a pivot eligible arc in *best_cand* almost every time that it looks there.

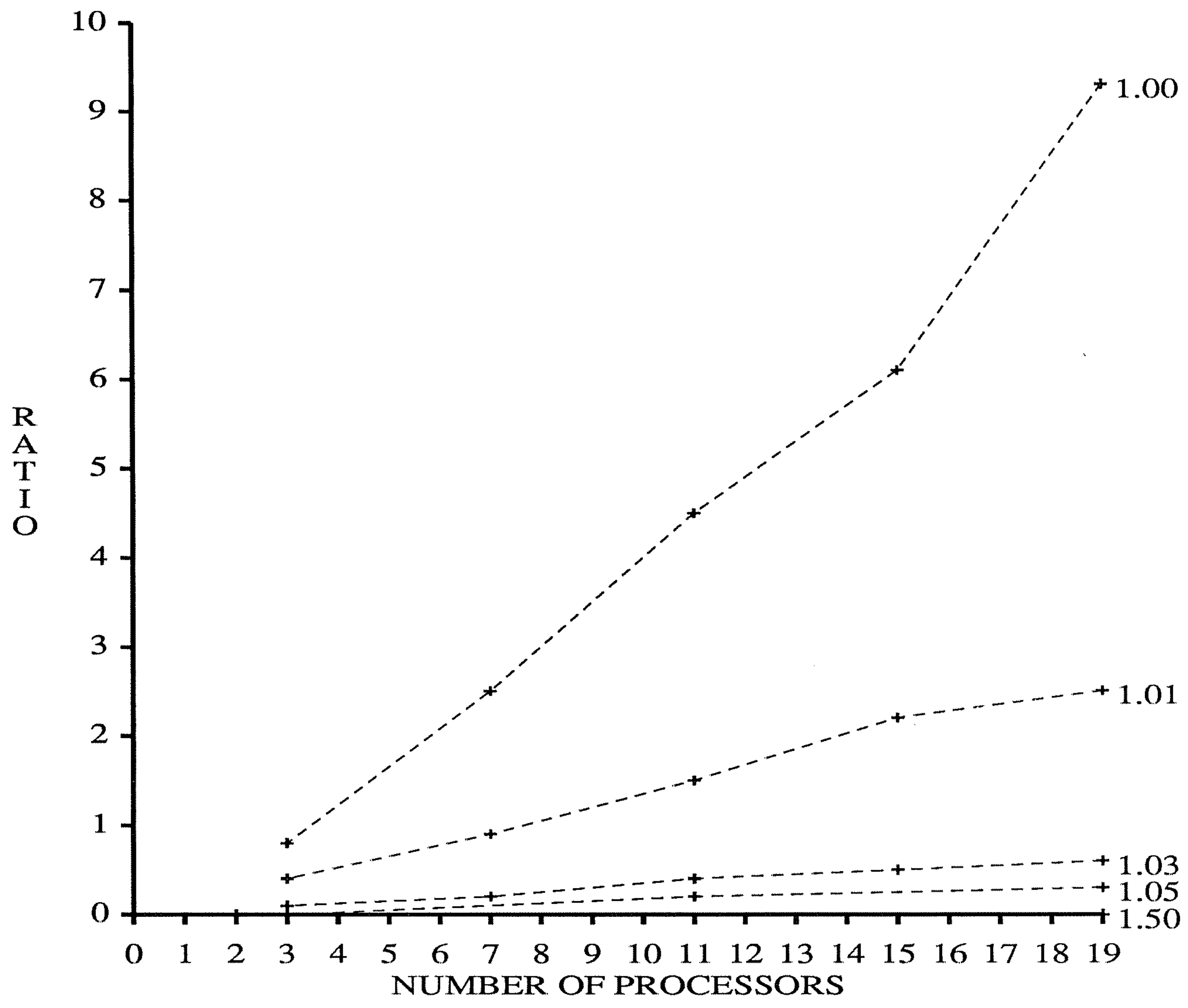

Figure 3.6 Ratio of collisions to # pivots for PGRNET

Problem #	1.00	1.01	1.03	1.05	1.50
collisions 19 procs	1,044,340	296,144	70,768	40,072	615
pivots 19 procs	111,843	115,680	106,496	103,064	76,367
ratio 19 procs	9.3	2.5	0.6	0.3	0.0
collisions 3 procs	94,875	56,714	15,084	5,964	31
pivots 3 procs	117,075	126,604	116,083	109,244	76,847
ratio 3 procs	0.8	0.4	0.1	0.0	0.0

Table 3.6 Collisions, pivots and collision/pivot ratio

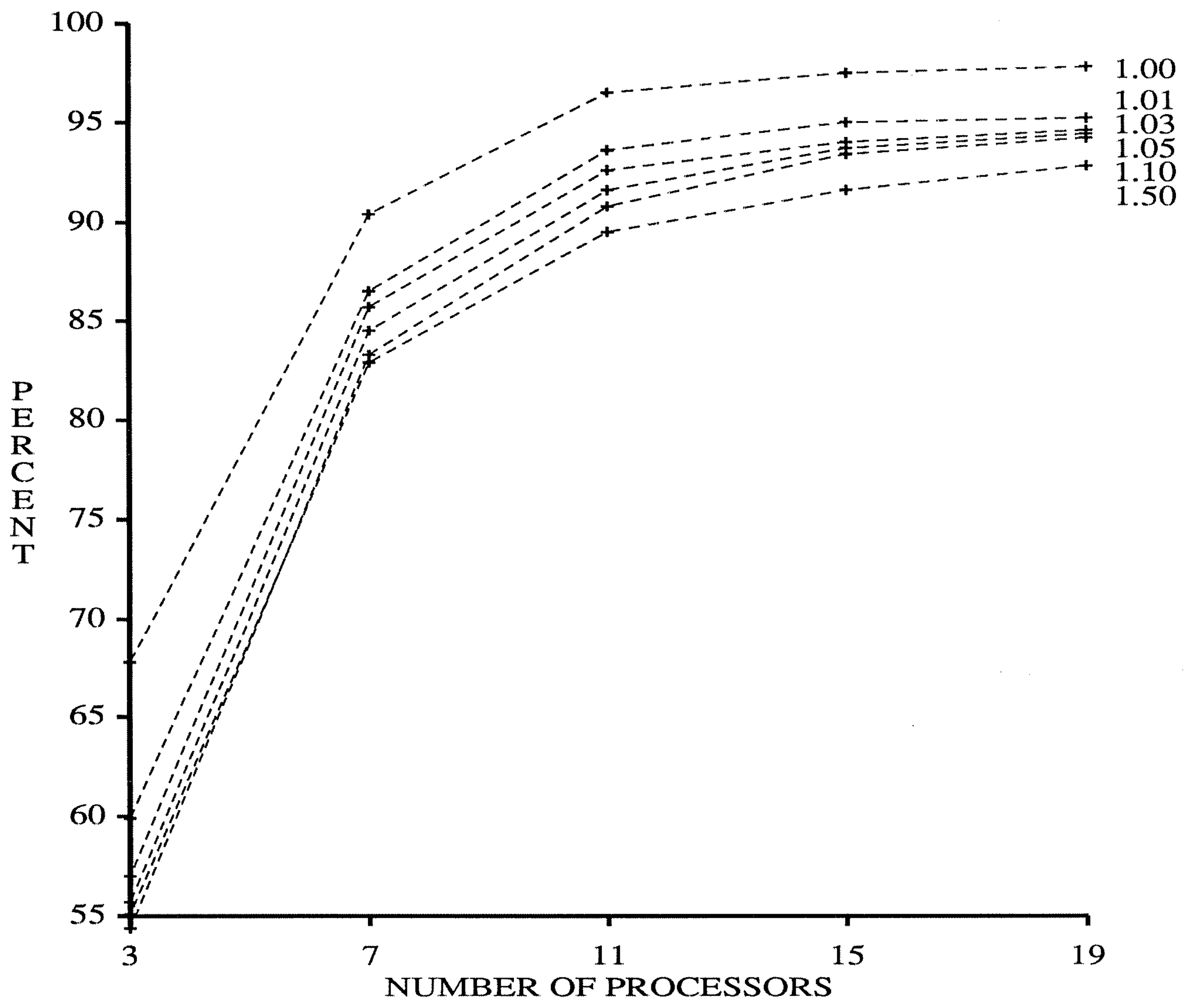

Figure 3.7 TPGRNET pivot arcs taken from *best_cand* (Group 1)

Problem #	1.00	1.01	1.03	1.05	1.10	1.50
best_cand 19 procs	86,252	91,282	87,131	85,275	81,567	68,530
cand lists 19 procs	1,877	4,512	4,879	4,989	4,944	5,194
STAGE 2 pivots 19 procs	15	4	16	28	23	51
best_cand 3 procs	81,862	82,767	75,812	70,810	64,388	46,884
cand lists 3 procs	38,837	55,267	57,028	56,174	53,876	38,131
STAGE 2 pivots 3 procs	9	7	9	2	0	2

Table 3.7 # pivot arcs taken from *best_cand* and cand lists

Figure 3.8 and tables 3.8 and 3.9 show results for two problems with more than a million variables. The problem reported in table 3.8 has the same sort of capacitation that groups 1 through 4 have, and the problem in table 3.9 has lighter capacitation. Both of these problems are small grained, so they can be solved quite efficiently by PGRNET. Due to the large number of quasi-trees in the optimal bases of these problems, one would expect the speedups on 19 processors to be somewhat higher than they are, 11.0 for the heavily capacitated problem and 7.0 for the lightly capacitated problem. One possible factor is that the system time for the major page faults might be included, to some extent, in the user time for these problems.

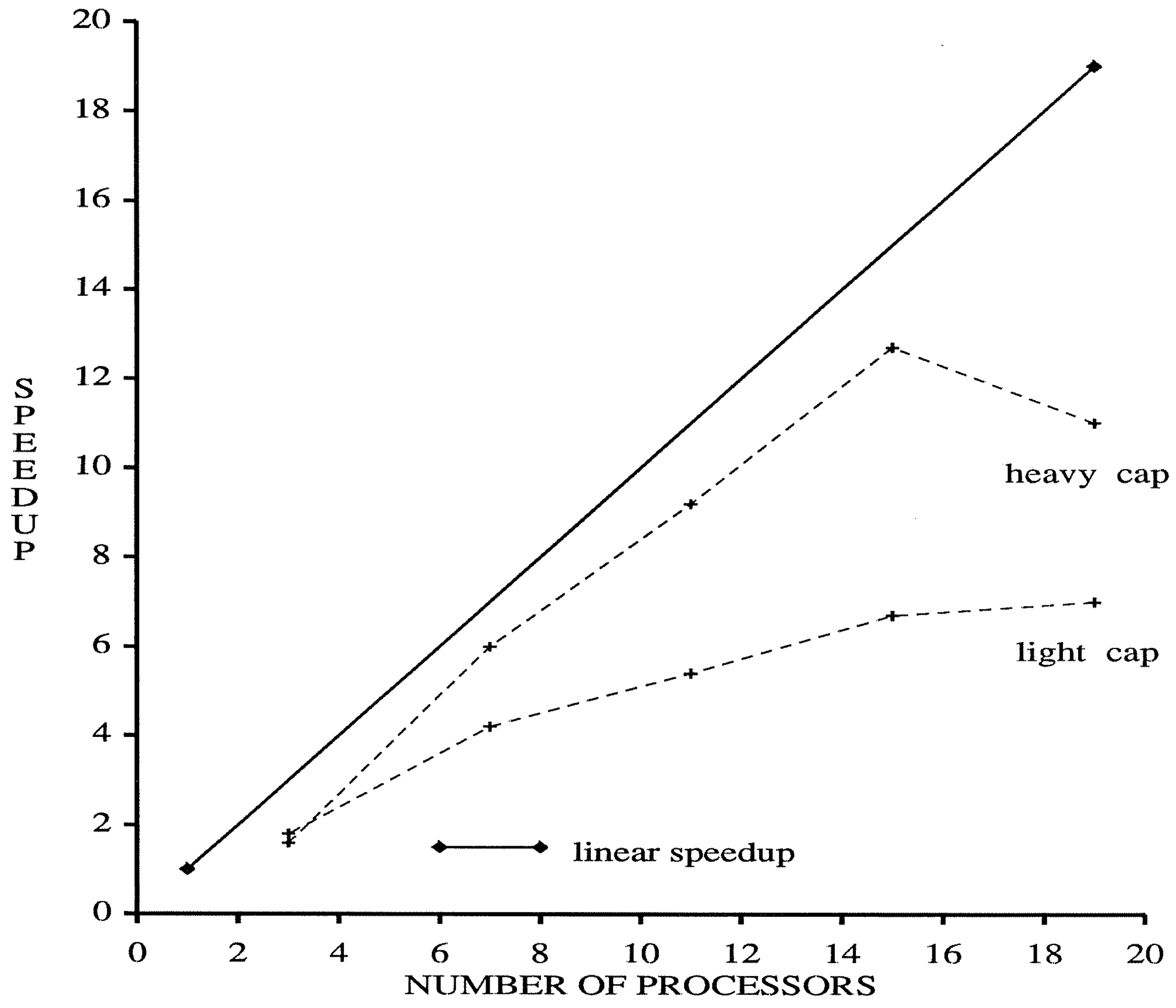

Figure 3.8 Speedups for PGRNET (# arcs > 1,000,000)

program	# arcs	# nodes	# qtrees	time	pivots	maj page swap
sequential	1,267,185	30,000	14,859	8,415	706,776	914,478
19 procs	1,267,185	30,000	14,859	760	716,218	100,614

Table 3.8 PGRNET results for problem with heavy capacitation

program	# arcs	# nodes	# qtrees	time	pivots	maj page swap
sequential	1,267,185	30,000	14,859	3,305	184,379	363,755
19 procs	1,267,185	30,000	14,859	470	185,522	51,146

Table 3.9 PGRNET results for problem with light capacitation

4 Future Research

We are currently investigating a hybrid algorithm that incorporates features from both PGRNET and TPGRNET. This algorithm starts by executing pivots in a PGRNET phase and switches into a TPGRNET phase if the number of quasi-trees in the basis drops below some threshold. Another generalization of the parallel pricing approach that we wish to study is the utilization of some of the processing power for the simplex "ratio tests". That is, given a sufficiently large number of processors (say, more than 15 in the case of generalized networks), it may be more advantageous to utilize some of the additional processors to do ratio tests for pivot eligible columns rather than to allocate them to do additional pricing. We are also testing our algorithms with problems generated by GNETGEN, a modification of the pure network generator, NETGEN [Klingman, et al, 1974]. Results will appear in a joint paper with J. Kennington and R. Muthukrishnan, who have independently been developing distributed algorithms for GP. Further into the future, we would like to develop a heuristic for PGRNET that would allocate arcs to processors using information from a solved problem. The optimal basis of the solved problem (the warm start) might resemble the optimal basis of a problem with similar data. If so, then the warm start would indicate how to allocate groups of arcs to processors in such a way that processors would rarely collide with each other when selecting pivot arcs. Finally, we would like to test variants of the TPGRNET strategy on general linear programs. The TPGRNET algorithm clearly extends in a straightforward way to general linear programs, but implementation details will play an important role in determining efficiency.

References

Adolphson, D. [1982]:
"Design of primal simplex generalized network codes using a preorder thread index", Working Paper, School of Management, Brigham Young University, Provo.

Barr, R., Glover, F. and Klingman, D. [1979]:
"Enhancements of spanning tree labeling procedures for network optimization", *INFOR* 17, 16-34.

Brown, G.G., McBride, R.D. [1984]:
"Solving generalized networks", *Management Science*, 30, 1497-1523.

Chang, M. and Engquist, M., Finkel, R. and Meyer, R. [1987]:
"A parallel algorithm for generalized networks", *Annals of Operations Research* 14(1988) 125-145.

Chang, M. and Engquist, M. [1986]:
"On the number of quasi-trees in an optimal generalized network basis", *COAL Newsletter* 14, 5-9.

Clark, R. and Meyer, R. [1987]:
"Multiprocessor algorithms for generalized network flows", Technical Report #739, Department of Computer Sciences, The University of Wisconsin-Madison

Clark, R., Kennington, J., Meyer, R. and Muthukrishnan, R. [1989]:
"Parallel algorithms for generalized networks: computational experience.", to appear in 1989

Engquist, M. and Chang, M. [1985]:
"New labeling procedures for the basis graph in generalized networks", *Operations Research Letters* Vol. 4, No. 4, 151-155.

Glover, F., Klingman, D. and Stutz, J. [1973]:
"Extension of the augmented predecessor index method to generalized network problems", *Transportation Science* 7, 377-384.

Glover, F., Karney, D., Klingman, D. and Napier, A. [1974]:
"A computation study on start procedures, basis change criteria, and solution algorithms for transportation problems", *Management Science* 20, 793-813.

Glover, F., Hultz, J., Klingman, D. and Stutz, J. [1984]:
"Generalized networks: a fundamental planning tool", *Management Science* 24, 1209-1220.

Grigoriadis, M. [1984]:
"An efficient implementation of the network simplex method", *Mathematical Programming Study* 26, 83-111

Jensen, P. and Barnes J. W. [1980]:
Network Flow Programming, John Wiley and Sons, New York.

Kennington, J. and Helgason, R. [1980]:
Algorithms for Network Flow Programming, John Wiley and Sons, New York.

Kennington, J. and Muthukrishnan R. [1988]:
"Solving generalized network problems on a shared memory multiprocessor," Technical Report 88-OR-21 of the Department of Operations Research and Engineering Management, Southern Methodist University, Dallas, Texas.

Klingman, D., Napier, A. and Stutz J. [1974]:
"NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow problems," *Management Science*, 20, 814-821

Murtagh, B. and Saunders, M. [1978]:
"Large-scale linearly constrained optimization", *Mathematical Programming* 14, 41-72.

Peters, J. [1988a]:
"A parallel algorithm for minimal cost network flow problems", Technical Report # 762, Department of Computer Sciences, The University of Wisconsin-Madison

Peters, J. [1988b]:
"The network simplex method on a multi-processor", June 1988, to appear in *Networks*

Zenios, S. [1986]:
"Sequential and parallel algorithms for convex generalized network problems and related applications", PhD thesis, Civil Engineering Department, Princetion University, Princeton, N. Jersey

Zenios, S. and Mulvey J. [1985]:
"A distributed algorithm for convex network optimization problems", Report EES-85-10, Engineering-Management Systems, Civil Engineering Department, Princeton University, Princeton, N. Jersey