A PARALLEL ALGORITHM FOR GENERALIZED NETWORKS

by

Michael D. Chang
Michael Engquist
Raphael Finkel
Robert R. Meyer

Computer Sciences Technical Report #642
March 1987

A PARALLEL ALGORITHM FOR GENERALIZED NETWORKS*

by

Michael D. Chang
Department of Computer Science and Operations Research
North Dakota State University
Fargo, ND 58105

Michael Engquist
Cleveland Consultant Associates
Austin, TX 78731

Raphael Finkel
Robert R. Meyer
Department of Computer Sciences
The University of Wisconsin
Madison, Wisconsin 53706

*This research was supported in part by NSF grants DCR-8503148 and CCR-8709952 and by AFOSR grant AFOSR-86-0194.

A PARALLEL ALGORITHM FOR GENERALIZED NETWORKS

by

Michael D. Chang
Michael Engquist
Raphael Finkel
Robert R. Meyer

ABSTRACT

This paper presents an application of parallel computing techniques to the solution of an important class of planning problems known as generalized networks. Three parallel primal simplex variants for solving generalized network problems are presented. Data structures used in a sequential generalized network code are briefly discussed and their extension to a parallel implementation of one of the primal simplex variants is given. Computational testing of the sequential and parallel codes, both written in Fortran, was done on the CRYSTAL multicomputer at the University of Wisconsin, and the computational results are presented. Maximum efficiency occurred for multiperiod generalized network problems where a speedup approximately linear in the number of processors was achieved.

Generalized network problems form a special class of linear programming (LP) problems. A wide variety of important planning problems can be modeled as generalized networks. Glover et al. [19] discuss a number of these applications.

The method most widely used in practice for solving LP problems is the primal simplex algorithm, see Charnes and Cooper [7] and Dantzig [8]. This algorithm proceeds from one extreme point of the feasible region to another until optimality is achieved. Karmarkar [21] proposed a method for the solution of LP problems which does not utilize extreme points, and he notes that his algorithm lends itself to parallel computation but gives no details. (However, the results of the recent computational study [17] cast doubt on Karmarkar's claim that his algorithm will supercede the simplex algorithm.) The simplex algorithm for general LP problems does not readily yield to parallelization, although certain steps of this algorithm can benefit from vector processing. (Such steps include pricing–the calculation of reduced costs–and the so-called BTRAN and FTRAN operations when the product form of the inverse basis matrix is used. These operations involve the calculation of dual variables and the updated entering column, respectively.) However, since much of the work incurred by the simplex algorithm involves the application of Gaussian elimination to factor the basis matrix, which is typically large and sparse, parallelization of this algorithm depends heavily on developments in this area. (Vector processing has been applied to such operations for dense matrices [11].) Factorization for the basis matrices of generalized networks is not a problem because of their special structure as discussed in Section 2.1. The partitioning methods we describe for generalized networks can be extended to LP problems with staircase structure [15], [16].

Theoretical work in parallel optimization algorithms for problems with *network* structure, including the minimum spanning tree problem, the shortest path problem, and the traveling salesman problem, is presented by Quinn and Deo [29]. A parallel algorithm for a class of nonlinear multicommodity network flow problems known as traffic equilibrium problems is described by Feijoo and Meyer [14]. This algorithm uses a decomposition approach and includes computational experience on the CRYSTAL multicomputer [9] at the University of Wisconsin. Parallel computing applied to a multiple-cost-row linear programming method is proposed by Phillips and Rosen [27] in connection with solving linear complementarity problems. Their study was done using the CRAY XMP/48. Re-

cent mathematical programming studies which utilized vector processing include those of Plummer, Lasdon, and Ahmed [28] and Zenios and Mulvey [31].

The generalized network problem (GN) has the form

$$\text{minimize} \quad cx \tag{1.1}$$

$$\text{subject to} \quad Ax = b \tag{1.2}$$

$$0 \leq x \leq u \tag{1.3}$$

where A is an $m \times n$ matrix; c, x and u are n-vectors; and b is an m-vector. The matrix A must satisfy the condition that each column contains at most two nonzero values. The upper bounds u are also known as capacities. By scaling and/or complementing a variable relative to its capacity, GN can be transformed so that each column has at least one entry which is $+1$. We assume that such a transformation has been applied to GN. A representation of GN as a directed network is then obtained as follows. With each row of A we associate a node of the directed network. Each column of A is associated with an arc directed away from the node corresponding to the $+1$ column entry (arcs corresponding to columns containing two $+1$ column entries can be directed either way). (Columns with a single nonzero entry are associated with arcs incident on only one node, and such arcs are called self-loops.) Thus, a column with nonzero components of $+1$ and $-\theta$ in row i and j, respectively, corresponds to the generalized arc (i, j) of Figure 1. The value θ, which is shown in the triangle, is called the *multiplier* of the arc.

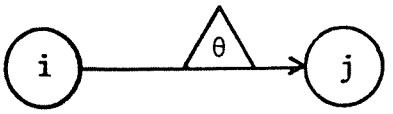


Figure 1. A generalized arc.

The variable f associated with the arc of Figure 1 can be interpreted as the amount of flow leaving node i. These f units of flow are transformed across the arc to θf units of flow entering node j. Since self-loops with large costs can be adjoined to the problem if necessary, it is assumed that A in (1.2) has rank m.

Sequential implementations of the primal simplex algorithm for GN have been described by Elam, Glover, Hultz, Klingman, and Stutz [12], [18], [19], [20], by Brown and McBride [5], and by Engquist and Chang [13]. These implementations utilize network data structures to achieve efficiency. Extensive computational results in [13] demonstrate that GRNET, a generalized network code, runs about 60 times faster than the general purpose LP code MINOS [26].

In this paper we develop parallel variants of the primal simplex algorithm for GN. One of these variants was implemented on CRYSTAL, and a computational study comparing this implementation with GRNET is presented.

2. A Sequential Primal Simplex Algorithm for GN

This section provides background on network data structures and their use in implementing the primal simplex algorithm for GN. The data structures we describe were used in GRNET. We limit ourselves to an overview here because the details are available in [5], [12], and [13].

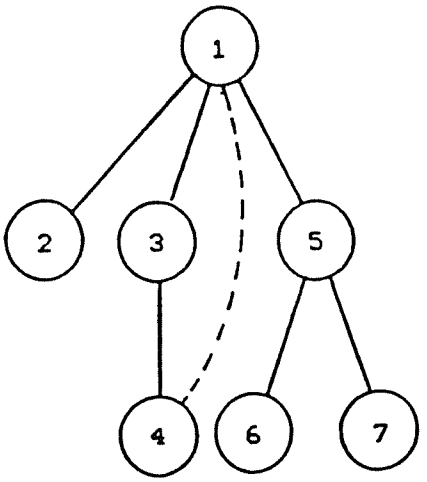
2.1 Basis Structure

It is known [2], [22] that a basis for GN can be represented graphically as a collection of quasi-trees (also called one-trees) where a quasi-tree is a tree to which a single arc has been adjoined. Thus a quasi-tree contains a single closed path, and this closed path is called a loop. It is possible for this loop to be incident to just a single node. The latter case, as mentioned previously, is known as a self-loop.

Next we discuss the node functions used to represent and maintain basis quasi-trees. Suppose that a quasi-tree Q is given. We select one of the arcs of the loop of Q and designate it as the *special arc* of Q. When the special arc is removed from Q, a tree T results. One of the end nodes of the special arc is designated as the *root* of T. Then node functions which were originally used for basis trees of pure networks in [3] can be used.

These node functions are defined as follows. If i and j are nodes incident to an arc of T such that i is closer to the root than j, then i is called the predecessor of j and node j is called an immediate successor of node i. The subtree of T that contains a node x and all of its successors will be denoted as $T(x)$. For nodes other than the root node, $p(x)$ denotes the predecessor of x. If x is the root node, then $p(x)$ is the node at the

opposite end of the special arc. The *preorder successor* of node x is denoted as $s(x)$. Once a preorder is given, the *thread* is defined as the function which traces the tree nodes in preorder. The value of the thread for the last node in the preorder is defined to be the root. The function whose value at node x is the *number of nodes* in $T(x)$ is denoted by $t(x)$. The *last node* of $T(x)$ in the preorder defining s is denoted by $f(x)$. For nodes on the loop the predecessors are flagged by means of a negative index in order to facilitate detection of loop nodes during the simplex ratio test. An example showing these node functions is given in Figure 2. Since arc direction does not affect node function values, arcs are shown as undirected in the figure.



	Functions			
Node	p	s	f	t
1	-4	2	7	7
2	1	3	2	1
3	-1	4	4	2
4	-3	5	4	1
5	1	6	7	3
6	5	7	6	1
7	5	1	7	1

Figure 2. Example quasi-tree (special arc dotted).

To eliminate a certain search in updating. GRNET also keeps the *reverse thread* function r, which is simply the inverse of s. For, GRNET, GN problem data is stored using arc length arrays for the head node, cost, capacity, and multiplier. This arc data is organized with all arcs emanating from a node (the forward star) in contiguous positions. A node length pointer array gives direct access to each forward star. Another node length array lk is required to link the basis representation to the arc data. For a node i in a quasi-tree, $lk(i)$ is the arc number of the arc between node i and its predecessor.

2.2 Primal Simplex Steps

1. Initialization

   The initial basis consists of self-loops with very large (Big-M) costs for each problem node. This is called an artificial start. Initial dual variables (node potentials) are set.

2. Pricing

   The reduced cost of an arc is calculated using the cost and multiplier of the arc and the node potentials of the end nodes of the arc. The entering arc is chosen either as a nonbasic arc with zero flow and negative reduced cost or as a nonbasic arc with flow at capacity and positive reduced cost. If an entering arc cannot be found, GN is solved. In GRNET, a candidate list strategy [25] is used in which pivot eligible arcs are placed on a list. Entering arcs are selected from the list if possible. GRNET uses a candidate list with a maximum length of 30 and up to 15 pivots are made from each list before it is updated.

3. Determine leaving arc and update flows

   Predecessor paths are traced back from the end nodes of the entering arc to the loop(s) of the quasi-tree(s) involved in order to accomplish this step. The leaving arc is found by carrying out the ratio test as this trace is made. Another trace of these paths is needed to update flows.

4. Basis exchange

   The entering arc becomes basic, the leaving arc becomes nonbasic, and the node functions used to represent the basis quasi-trees are updated as needed.

5. Additional updates

   Node potentials and possibly loop factors change for the new basis. (This update can be integrated with the basis exchange step.) Go to Step 2.

GRNET uses an array for saving loop factors and the updated entering column representation. Also, GRNET includes a step which corresponds to a basis reinversion every 300 pivots. That is, by tracing basis quasi-trees appropriately, GRNET recalculates node potentials and flow directly from GN problem data.

3. Parallel Primal Simplex Variants for GN

In this section, three parallel primal simplex variants for GN are considered. These approaches vary in the way in which they handle pricing and interprocessor communication. All three variants have tasks that are large-grained and are intended for a multiprocessor architecture involving relatively few processors.

The three variants all utilize a subset of the processors called SP. The members of SP (*simplex processors*) perform the sequential simplex algorithm on certain subproblems. These subproblems, which we call *local problems*, are created by partitioning the nodes of GN among the members of SP. The *local arcs* of a local problem are those arcs that connect the nodes of the local problem. The remaining arcs, which may be thought of as going between processors, are called *cross arcs*. A local problem is a generalized network problem in its own right with the problem data it inherits from GN. Because of the artificial start, all local problems are feasible.

The first primal simplex variant is termed the *Pure SP Method* because all of its processors are in SP. As the local problems reach optimality, they are modified by the exchange of nodes in basis quasi-trees to form new local problems. It is necessary for each processor to maintain a list which keeps the initial allocation of nodes to members of SP. This initial location of a node is called its home. Further, the pricing of cross arcs is the responsibility of the processor that currently owns the tail node of the arc. (Cross arcs are nonbasic and this processor will also maintain their status–at zero or capacity.) When

trying to get information about node v, the head node of a cross arc, a processor first sends the request to v's home. If v's home does not currently own v, it forwards the request to v's current location. To mitigate the fact that the location of a quasi-tree in transit is not known to the home yet, each processor caches forwarding addresses for recently sent nodes.

When a processor in SP solves a local problem to optimality, it requests node potentials for the head nodes of its cross arcs and prices these cross arcs. When a cross arc is chosen to enter the basis, the processor requests that the quasi-tree at the opposite end of the cross arc be transferred to augment its local problem. A quasi-tree involved in a cross arc pricing operation is prevented from entering into further pivots until it is known whether this quasi-tree is to be transferred. In this way, correct node potentials are used in pricing and optimality for GN occurs when all local problems are optimal and no pivot eligible cross arcs can be found.

For the *second* primal simplex variant, in addition to SP, a disjoint set of *pricing processors* PP is used. One member of PP is chosen as the *master processor* (*mp*). It is the function of *mp* to direct the pricing activities of the other members of PP and to maintain the list of cross arcs. This variant is known as the *PP/SP Method*. The master processor prices cross arcs or instructs other members of PP to do so, and it keeps the most promising cross arcs on a candidate list. Since the members of PP use possibly obsolete node potentials for pricing, the arcs on the candidate list are not guaranteed to be pivot eligible. When the members of SP have all reached optimality for their local problems, quasi-trees incident to arcs from the candidate list are transferred from one member of SP to another. The number and size of the quasi-trees moved is monitored and controlled by *mp* for load balancing purposes. After the transfer of quasi-trees, the members of SP send current node potentials to *mp* and another parallel iteration begins. Optimality for GN is reached when all local problems are optimal and *mp* cannot find any candidate arcs.

The *third* variant, which we term the *Hybrid Method*, combines features of the two previous methods. The Hybrid Method retains both sets of processors PP and SP but it allows a member sp_1 of SP to interrupt *mp* and ask for promising cross arcs incident to nodes of its local problem as soon as it has reached local optimality. Once a cross arc is chosen, the processor sp_2 at the opposite end of this arc from sp_1 must be interrupted

and instructed to transfer the quasi-tree involved to sp_1. After this operation, sp_1 passes current node potentials to mp. The check for GN optimality is the same as for the PP/SP Method.

The Pure SP Method treats all processors equally and, in this sense, it is more parallel than the other two proposed methods. However, this method may be somewhat myopic in its search for good cross arcs. The PP/SP method has a step in which all processors exchange information, and for a distributed-memory system this may result in excessive communication. However, the use of mp should produce better cross arcs to enter the basis since it selects candidate arcs by searching all possible cross arcs. It is hoped that the Hybrid Method will capitalize on the good features of the other two methods.

An important aspect of all three methods that we have discussed is the procedure used to determine the initial partition of nodes among the members of SP. A good partition is one which results in local problems for which a large number of low cost local arcs exist. Details of the partitioning procedures that we used will be given in the following sections.

4. The CRYSTAL Multicomputer

CRYSTAL [9] is a set of VAX 11/750 computers (currently there are 20) with 2 megabytes of memory each, connected by a 80 megabit/sec Proteon ProNet token ring and accessed via VAX 11/780 "host" machines. This multicomputer can be used simultaneously by multiple research projects by partitioning the available processors according to the requirements of each project. Partitioning is done via software, and, once a user has acquired a "partition" or subset of processors, the user then has exclusive access to the node machines of that partition. CRYSTAL software is largely written in a local extension to Modula, but also includes communications routines (the simple-application package) callable from Fortran, Pascal, and C. Development, debugging, and execution of projects takes place remotely through any of several VAX 11/780 hosts runing Berkeley Unix 4.2. Acquiring a partition of node machines, resetting each node of the partition, and then loading an application onto each node may be performed interactively from any host machine. CRYSTAL has been used for research in a variety of areas, including distributed operating systems, programming languages for distributed systems, tools for debugging distributed systems, multiprocessor database machines, protocols for high performance local network communications, numerical methods, and recursive search. The latter work is of par-

ticular interest in terms of providing preliminary combinatorial optimization experience with CRYSTAL as well as communications tools useful for studying branch-and-bound algorithms for global optimization.

5. Implementation

In this section a Fortran code for the PP/SP parallel simplex algorithm is described. Of the three algorithms presented in Section 4, we started with PP/SP because it is the easiest to implement. This implementation was done on CRYSTAL and the resulting code is known as GRNET-K.

GRNET-K utilizes K processors where K is a value in the range from 3 to 20. It is limited to a single member in PP, and for simplicity, it departs slightly from PP/SP in that it uses two stages. In Stage 1, the K processors are distributed as follows:

$$PP = \{ \quad processor\ K \quad \}$$
$$SP = \{ \quad processor\ 1, ..., processor\ (K-1) \quad \}$$

where processor K is designated as the master processor (mp). Stage 1 termination criteria are based upon candidate list size and are discussed in Section 5.3 below. In the transition from Stage 1 to Stage 2, all processors i, $1 < i < K$, send their basis quasi-trees to processor 1. In Stage 2, processors i, $1 < i \leq K$, are idle while processor 1 executes the sequential simplex algorithm. This approach is motivated by the time-consuming nature of a message-passing validation of optimality.

The host program for GRNET-K is shown in Figure 3. Each client processor receives a copy of the GN problem data. The processors in SP flag this problem data to identify their own local problems while mp prices cross arcs.

```
PROGRAM Host
        Read problem data
        Start client programs
        Send data to client processors
        Receive solution from client processors
        Write solution
END
```

Figure 3. Host program for GRNET-K.

5.1 Initial Partitioning Procedures

GRNET-K has two options available for setting up the initial partition of GN nodes among the processors of SP. For both options, once this partition is set up, the resulting local problems use an artificial starting basis. That is, each node of a local problem has a self-loop attached which carries a big-M cost. The second option requires access to a node's reverse star (the set of arcs entering the node). GRNET-K provides this access using arrays which are described in Section 5.2.1. The two options are:

(i) Set up a user-supplied partition.

(ii) Use the following procedure.

For each unlabeled node i,

1. Find an arc e in the union of the reverse and forward stars of i, ignoring self-loops, such that e has least cost.

2. If the end node of e opposite to i is labeled, then label i with the same processor number.

3. If the end node of e opposite to i is unlabeled, then label both this node and i with the same processor number. (The processor number used is based on load balancing considerations.)

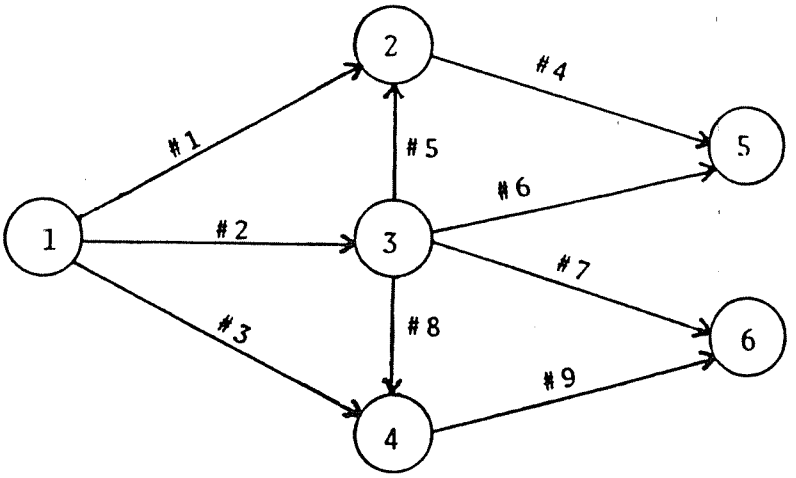
5.2 Quasi-tree Transfer and Related Updates

When a quasi-tree is transferred from one processor to another by GRNET-K, the local problems on the two processors involved are redefined. Of course, the set of cross arcs is also changed.

5.2.1 Data Structures

In addition to the arrays used by GRNET for maintaining the basis quasi-trees, GRNET-K employs an array pn on the master processor and arrays pb and $barc$ on the members of SP. The current partition of GN nodes in processors is kept by pn where $pn(i) = j$ in case node i is a node of the local problem on processor j.

The status of nonbasic arcs is determined by whether their flows are at zero or at capacity. It is necessary that the correct status of nonbasic local arcs be available to the processor which currently owns them and that the status of cross arcs be available to mp. The arrays pb and $barc$ are used to help maintain this information. The value $pb(i)$ is the arc number of the first arc entering node i. The value $barc(k)$ is the arc number of the arc entering the head node of k, immediately after k. Also, $pb(i) = Big - M$ in case there is no arc entering node i, and $barc(k) = Big - M$ in case k is the last arc in the reverse star of the head node of k. This structure is shown in Figure 4.



node	pb	arc	barc
1	999	1	5
2	1	2	999
3	2	3	8
4	3	4	6
5	4	5	999
6	7	6	999
		7	9
		8	999
		9	999

Figure 4. Access to reverse star.

These arrays are flagged to identify local arcs for pricing purposes. For *arc* k, $barc(k) < 0$ indicates that the tail node of k is in the local problem, and if i is the head node of k, $pb(i) < 0$ indicates that node i is in the local problem. When quasi-trees are transferred, these flags must be updated.

Capacities for GRNET-K are flagged negative, as they are for GRNET, to indicate an arc that is nonbasic at capacity. The capacity array for a member of SP must have the correct flags for its local problem arcs, while the capacity array for mp must have the correct flags for cross arcs. The pn array allows mp to identify cross arcs.

5.2.2 Quasi-tree Information Exchange

GRNET-K uses the simple-application package of CRYSTAL for communication between processors. The simple-application package provides auxiliary Fortran subroutines which allow certain arrays (buffers) to be sent and received as messages. In order for a quasi-tree owned by processor x to be transferred to processor y, processor x must formulate and send an appropriate message. Along with node i, both $lk(i)$ and $p(i)$ are sent in the message. We take advantage of the fact that the thread is an ordering on the nodes of a quasi-tree, and these nodes are put into the buffer in this order so that processor y can recreate the functions s as it unloads the message.

In order to avoid excessive communication times, node functions that are easily recalculated were not sent as messages. (This is known as "overlapped work".) These node functions are r, t, f, flow, and poten (the node potentials). The reverse thread r is easily set as the nodes are received in thread order. As node i is received, $t(i)$ is set to 1 and $f(i)$ is set to 0. To complete the recalculation of t and f, the quasi-tree is traversed in reverse thread order and for each node i, the following steps are executed:

1. If $t(i) = 1$, then set $f(i) = i$.
2. If $f(p(i)) = 0$, then set $f(p(i)) = f(i)$.
3. Set $t(p(i)) = t(p(i)) + t(i)$.

The recalculation of flow and poten are done in the same fashion as in a generalized network basis reinversion.

5.3 Stage 1 Logic

Once the initial partition of GN nodes for the processors of SP is set up, these processors solve their local problems in parallel while *mp* generates a candidate list of cross arcs. When the processors of SP have all reached optimality for their local problems, all processors exchange information. Then another parallel step is taken, followed by another information exchange, and so on, until Stage 1 is completed. The client program for GRNET-K is given in Figure 5. (Parallel tasks are shown using a notation (parbegin, parend) introduced by Dijkstra [10].)

```
PROGRAM Client
    receive problem data from host
            stage := 1
            repeat
                  begin
                        parbegin
                        processor K pricing
                        processor K-1 pivoting
                        . . . . . .
                        processor 1   pivoting
                        parend
                        exchange information
                  end
            until ( stage = 2 )
            exchange information
            repeat
                  processor 1 pivoting
            until optimal
    send solution and timing to host
END
```

Figure 5. Client Program for GRNET-K

The candidate list generated by *mp* has a maximum length equal to the number of GN problem nodes. For each node, the best pivot eligible cross arc leaving that node, if any, is placed on the list. Of course, the determination of pivot eligibility is made with

reference to the possibly obsolete node potentials available to mp. A threshold reduced cost value is used by mp as a further requirement which must be met by a cross arc in order for it to be placed on the candidate list. That is, the absolute value of the reduced cost must be at least as great as the threshold value. It may be necessary to adjust the threshold value according to the cost range for GN in order to achieve good performance. However, a fixed threshold value of 1.0 was used for all the testing discussed in Section 6.

Next, we discuss the information exchange and its control. When the members of SP have reached local optimality, mp takes the next available cross arc, say (i, j), from its candidate list. It then tells the processor that currently owns node i to transfer the quasi-tree containing node i to the processor that currently owns node j. When $\lfloor 60/K \rfloor$ quasi-trees have been transferred in this manner or no arcs remain on the candidate list, mp notifies the members of SP that the current exchange of quasi-trees is complete. As a quasi-tree is transferred from processor x to processor y, information is exchanged as discussed in Section 5.2.2. In particular, information on the new location of quasi-tree nodes and on newly created cross arcs which are nonbasic at capacity is sent to mp. To complete an information exchange step, the members of SP send current node potentials to mp.

In Stage 1, and Stage 2 as well, candidate lists of local arcs are formed by processors which perform pivots. These candidate lists have a maximum length of 30, and up to 15 of the best arcs are selected from the list. Stage 1 is terminated when the candidate list formed by mp contains fewer than $\lfloor 60/K \rfloor$ cross arcs.

6. Computational Study

In this section, a computational comparison of the parallel code GRNET-K with the sequential code GRNET is made. In contrast to previous computational studies involving network optimization, e.g. [23], [30], the present study takes place on CRYSTAL, which is an experimental computing environment. For this reason, the goal of the present study is to determine the long-range implications of parallel technology on solution methods for generalized network problems, whereas previous computational studies have been mainly concerned with finding the best refinements of known optimization algorithms on relatively well understood sequential machines.

The results of the present study carry over to parallel computers that make multiple

processors available at a reasonable cost. We have assumed that such a computer is available and that the user's major concern is reducing the run times of his jobs. Thus, we have chosen to focus on speedup (the ratio of sequential solution time to parallel solution time) as our main performance measure. Production parallel systems are still quite rare, and it is difficult to say precisely how billing will be handled for such systems. CRYSTAL does not bill users for the use of its resources.

6.1 Problem Generators

For pure networks, the problem generator NETGEN developed by Klingman, et al. [24] has been widely used. NETGEN has been extended to generate generalized network problems and this extension is known as NETGENG [19]. In a previous computational study [13] conducted using GRNET, it was discovered that the number of quasi-trees in an optimal basis for a NETGENG problem is typically one or two. In view of the desirability for parallel processing of having a large number of quasi-trees, this behavior raised the question of whether this range of quasi-tree cardinality was a general phenomenon for large-scale generalized networks or whether a test problem generator arising from a very different foundation would yield a different quasi-tree count. The development of the test problem generators GTGEN and MPGEN established the latter alternative.

6.1.1 GTGEN

GTGEN is a Fortran code capable of randomly generating generalized transportation problems, and it was previously described in [6]. The underlying transportation network is generated so as to be feasible and connected. GTGEN also allows some control of the number of quasi-trees in an optimal basis for the problems it generates. As an arc is created by GTGEN, a flow on this arc is also created as part of a feasible flow pattern. This flow is set at μ times the arc capacity, where μ is a user-specified fractional value. For test results reported herein, μ had a value of 0.25. The supply for an origin node is set to the value obtained by summing these generated flows over the forward star of the node. Similarly, the generated flows, after being transformed by the appropriate multiplier, are summed over the reverse star of a destination node to determine the demand. After the supply and emanating arcs for an origin node are initially set, this problem data is modified with probability q, where q is another user-specified parameter. This modification includes (1)

creation of a slack arc for the node and (2) creation of a new supply which is $1/\mu$ times the previous supply. The number of origin nodes multiplied by q thus gives the expected number of modified origin nodes. Computational results included in [6] show that this expected value provides a good estimate of the number of quasi-trees in an optimal basis. This is plausible since, relative to the original feasible flow generated, each modified origin node has an increased supply that will likely force the associated slack into the basis, leading to the creation of a corresponding quasi-tree. (This is not guaranteed, however, since the excess supply could result in flows at capacity on all arcs emanating from this origin node. The slack arc would then likely be nonbasic at zero flow in the optimal solution, and a corresponding quasi-tree would not occur.)

6.1.2 MPGEN

Multiperiod networks incorporate replications of a fundamental network, and these replications represent the fundamental network in a succession of time periods. There is an extensive literature on multiperiod network problems which has been documented in the survey papers [1], [4]. Multiperiod network problems are promising for parallel solution because the links between fundamental networks are often sparse and also because these links exist only between nodes of adjacent periods. We created MPGEN, a Fortran multiperiod generalized transportation problem generator, which calls GTGEN to create the fundamental network. In MPGEN problems, backlogging arcs are included which connect origins in the fundamental network for period $i+1$ to destinations in period i. Flow on such a backlogging arc represents an amount demanded in period i that is not shipped until period $i+1$. In a K period problem, a user-specified parameter d gives the probability that an origin node in the fundamental network for period j, $2 \leq j \leq K$, has a backlogging arc emanating from it. MPGEN adjusts the supplies, demands, and capacities of the fundamental networks as backlogging arcs are created to insure that some positive flows on these arcs occur in any feasible solution.

6.2 Computational Testing

Testing on GTGEN problems was carried out with the option (ii) heuristic starting procedure of Section 5.2. The problem specifications of two typical GTGEN problems, A and B, are given in Table 1.

	Problem	
	A	B
No. Origin Nodes	600	600
No. Dest. Nodes	400	500
No. Arcs Per Origin	16–20	16–20
Cost Range	1–100	1–100
Capacity Range	1–1000	1–1000
Multiplier Range	.90–.98	.90–.98
q	.30	.30

Table 1. Problem specifications for GTGEN.

These problems were solved by both GRNET and GRNET-K with values of K varying from 3 to 6. The test results are found in Tables 2 and 3 for problems A and B, respectively. The fact that the specifications of these problems differ only in the number of destination nodes causes GTGEN to create the same number of arcs for both of them. However, the network topology of the two problems is quite different. All times are in seconds and are exclusive of I/O. Iterations for GRNET-K are the sum of the iterations (pivots) made by all members of SP.

For GRNET-K, T_1 and T_2 are defined by:

$$T_1 = \sum_{j \varepsilon \{parallel\ tasks\}} max\ \{CPU_{1j}, CPU_{2j}, ..., CPU_{kj}\}$$

where CPU_{ij} denotes the CPU time for processor i to complete parallel task j, and

$$T_2 = Stage\ 2\ CPU\ time.$$

These values are used to define

$$Parallel\ CPU\ time = T_1 + T_2\ .$$

It is noted that Parallel CPU time includes neither the communication time required for information exchange nor the recalculation time for overlapped work (see the discussion in Section 5.2.2). Parallel CPU time is regarded as a surrogate for the elapsed time of a job on a shared memory parallel computer. With this in mind, we define

$$Speedup = \frac{Sequential\ CPU\ time}{Parallel\ CPU\ time}$$

Elapsed time (ET) for the parallel code is the actual solution time for GRNET-K on CRYSTAL. Of course, it includes all communication and recalculation time. For GRNET, CPU time and elapsed time are the same. We define

$$ET\ \ Ratio = \frac{Sequential\ elapsed\ time}{Parallel\ elapsed\ time}$$

		Sequential (GRNET)		Parallel (GRNET-K)				GRNET/GRNET-K	
K	Arcs	Iterations	CPU Time	Iterations	CPU Time	Stage 2 Time	Elapsed Time	Speedup	ET Ratio
1	10668	19141	392.94						
3	10668			18588	241.98	10.84	671.07	1.62	.59
4	10668			19306	216.31	12.42	750.79	1.82	.52
5	10668			21334	234.34	10.98	853.68	1.68	.46
6	10668			22979	258.40	7.69	1029.67	1.52	.38

Table 2. Computational results for problem A

		Sequential (GRNET)		Parallel (GRNET-K)				GRNET/GRNET-K	
K	Arcs	Iterations	CPU Time	Iterations	CPU Time	Stage 2 Time	Elapsed Time	Speedup	ET Ratio
1	10668	18564	384.06						
3	10668			17782	211.46	10.65	626.30	1.82	.61
4	10668			18878	192.26	9.54	702.75	2.00	.55
5	10668			21127	209.83	7.86	832.01	1.83	.46
6	10668			22788	245.12	8.67	979.23	1.57	.39

Table 3. Computational results for problem B

It is interesting to note that for both problems A and B the best speedup occurs for $K = 4$. However, when efficiency is defined as Speedup/K, the best efficiency occurs for these problems when $K = 3$. The small values of the ET Ratio reflect the relatively large amount of communication and recalculation time used by GRNET-K. Based on preliminary testing, we believe that the majority of this time is for communication. This is not surprising given the PP/SP logic, which causes all processors to send messages simultaneously. When implemented, the other simplex variants of Section 3 should yield much smaller communication times on CRYSTAL. Implementations of analogous algorithms in a shared-memory environment should also lead to significant reductions in communication time.

In order to better exploit the parallel capabilities of GRNET-K, we tested it on the more structured multiperiod problems generated by MPGEN. Four groups of test problems (C,D,E, and F) were used and the specifications for the fundamental networks of these problems are given in Table 4.

	Group			
	C	D	E	F
No. Origin Nodes	100	100	200	100
No. Dest. Nodes	400	400	300	400
No. Arcs Per Origin	6–16	6–16	6–16	6–16
Cost Range	1–100	1–100	1–100	1–100
Capacity Range	1–1000	1–1000	1–1000	1–1000
Multiplier Range	.90–.98	.90–.98	.90–.98	.90–.98
d	.50	.90	.70	1.0
q	.40	.40	.40	.40

Table 4. Specifications of fundamental networks for MPGEN

For these groups, GRNET-K employs special starting procedures– option (i) of Section 5.1. In problem groups C,D, and E, the fundamental network is replicated for K periods and GRNET-K uses K processors to solve the resulting problems, initially allocating the fundamental network nodes of each period to separate processors. The test results for problem groups C,D, and E are found in Tables 5, 6 and 7, respectively.

		Sequential (GRNET)		Parallel (GRNET-K)				GRNET/GRNET-K	
K	Arcs	Iterations	CPU Time	Iterations	CPU Time	Stage 2 Time	Elapsed Time	Speedup	ET Ratio
3	2231	2724	44.67	2706	22.67	0.37	37.02	1.97	1.21
4	3366	3764	60.17	3985	23.82	2.01	48.59	2.53	1.24
5	4501	5386	93.53	5335	27.20	3.04	54.47	3.44	1.72
7	6771	8408	160.01	7990	27.74	1.14	63.24	5.77	2.53
9	9041	10974	223.52	10698	29.09	1.52	73.57	7.68	3.04
11	11311	13944	306.69	13404	30.28	1.90	84.83	10.13	3.62
13	13581	16393	380.72	16042	31.52	2.29	96.43	12.08	3.95

Table 5. Computational results for group C.

		Sequential (GRNET)		Parallel (GRNET-K)				GRNET/GRNET-K	
K	Arcs	Iterations	CPU Time	Iterations	CPU Time	Stage 2 Time	Elapsed Time	Speedup	ET Ratio
3	2280	3597	110.05	3381	49.49	1.41	50.51	2.22	2.18
4	3569	3977	62.04	3628	21.90	2.80	42.87	2.83	1.45
5	4792	5370	87.01	4842	23.49	2.71	51.37	3.70	1.69
7	7238	8134	146.58	7370	26.78	4.28	65.21	5.47	2.25
9	9420	16452	880.29	13350	91.74	6.31	185.93	9.60	4.73
11	11800	20795	1275.29	16624	99.48	7.91	230.37	12.82	5.54
13	14180	24259	1434.96	20041	101.28	9.51	256.78	14.17	5.59

Table 6. Computational results for group D.

		Sequential (GRNET)		Parallel (GRNET-K)				GRNET/GRNET-K	
K	Arcs	Iterations	CPU Time	Iterations	CPU Time	Stage 2 Time	Elapsed Time	Speedup	ET Ratio
3	4473	6852	125.37	6362	63.21	3.06	78.74	1.98	1.59
4	6775	10204	200.24	9750	72.35	5.03	94.22	2.77	2.13
5	9078	13882	296.28	13080	78.80	9.04	109.22	3.76	2.71
7	13702	20778	484.23	19587	94.65	12.68	177.18	5.12	2.73
9	18315	28058	716.43	26143	108.05	13.56	255.95	6.63	2.80
11	22943	34759	945.25	32860	123.50	16.85	332.88	7.65	2.84
13	27561	42125	1255.22	39574	137.95	19.53	424.33	9.10	2.96

Table 7. Computational results for group E.

For these problem groups, the most interesting result is the linear increase of the speedup with the number of processors. The efficiency is about 0.7 for all K values for group E, but is roughly increasing with K for groups C and D. The ET ratios are also very encouraging for those groups. An unusual result occurs when $K = 9, 11$, and 13 for group D. Here, the efficiency is actually greater than 1. This is explained by the fact that GRNET-K takes fewer iterations (pivots) to reach optimality than does GRNET. Fewer iterations for GRNET-K have two possible explanations– a better starting procedure or a better rule (pivot rule) for selecting entering arcs. A comparison of pivot rules is difficult; however, it is plausible that GRNET-K is getting a better start than GRNET for these problems since it is provided with the node numbers of fundamental networks for each period. A better comparison with GRNET might be achieved if it also were provided these node numbers so that it could solve the fundamental networks for each period in sequence to get a starting basis. Such a procedure might cut GRNET iterations back to the number of iterations used by GRNET-K. However, using a proportional reduction in CPU time for GRNET, speedups would still remain impressive. On the other hand, experimentation with pivot rules for GRNET-K might further improve its performance.

To shed light on the performance of GRNET-K on multiperiod problems in the case in which the number of periods exceeded the number of processors, a single problem with 12 periods was created using the fundamental network for group F. This single problem, which we refer to as problem F, was solved once with GRNET and five times with GRNET-K using different starting procedures and a different number of processors each time. The test results are shown in Table 8.

		Sequential (GRNET)		Parallel (GRNET-K)				GRNET/GRNET-K	
K	Arcs	Iterations	CPU Time	Iterations	CPU Time	Stage 2 Time	Elapsed Time	Speedup	ET Ratio
1	14334	15692	339.68						
3	14334			15605	154.09	2.41	190.24	2.20	1.79
4	14334			15481	106.09	9.00	149.35	3.20	2.27
5	14334			15545	80.90	2.41	126.49	4.20	2.69
7	14334			15220	56.01	8.53	114.45	6.06	2.97
13	14334			14752	41.18	15.40	127.65	8.25	2.66

Table 8. Computational results for problem F.

When K processors are used, GRNET-K is given an initial allocation of $12/(K-1)$ consecutive fundamental networks per SP processor. In terms of elapsed time, 7 processors are best for the solution of this problem. Speedups increase with K, but the efficiency for $K = 13$ is less than for $K = 7$. For $K = 13$, the fact that Stage 2 time is 37% of Parallel CPU time helps explain the lower efficiency.

7. Conclusions

Three parallel primal simplex variants for solving generalized networks have been presented which differ in their degree of parallelism, their compatability with distributed computing, and their accommodation of primal simplex pricing mechanisms. The most straightforward of these variants has been implemented on the CRYSTAL multicomputer and tested against a sequential code on large-scale randomly generated test problems. For generalized transportation problems with a random sparsity pattern, it appears that the speedup afforded by this variant is limited to about 2, and this occurs when four processors are used. For multiperiod generalized network problems, however, in which a better initial approximation to the optimal decomposition is used, the results are much more encouraging with speedups achieved which are roughly linear in the number of processors used. Further implementation and testing is of interest. This would include work with all three of our simplex variants on distributed systems and also on shared memory machines. (Discussion of computational experience with these problem classes on the Sequent Balance 21000 multiprocessor is the subject of a forthcoming technical report. On multiperiod problems, the speedups that we obtained on the Balance 21000 are comparable to the results on Crystal that do not count communication time, and on single-period problems nearly linear speedups have been obtained for problems whose optimal solutions contain large numbers of quasi-trees.) Extension of our parallel methods to linear programming problems which exhibit staircase structure is another promising direction for future work.

REFERENCES

1. Aronson, J., "Survey of Dynamic Network Flows," Technical Report 85-OR-12, Department of Operations Research and Engineering Management, Southern Methodist University, Dallas, 1985.

2. Balas, E. and P. Ivanescu (Hammer), "On the Generalized Transportation Problem," *Management Science* 11, 188-202 (1964).

3. Barr, R., F. Glover and D. Klingman, "Enhancements of Spanning Tree Labeling Procedures for Network Optimization," *INFOR* 17, 16-34 (1979).

4. Bookbinder, J. and S. Sethi, "The Dynamic Transportation Problem: A Survey," *Naval Research Logistics Quarterly* 27, 447-452 (1980).

5. Brown, G. and R. McBride, "Solving Generalized Networks," *Management Science* 30, 1497-1523 (1984).

6. Chang, M. and M. Engquist, "On the Number of Quasi-trees in an Optimal Generalized Network Basis," *MPS Committee on Algorithms Newsletter* 14, 5-9 (1986).

7. Charnes, A. and W. Cooper, *Management Models and Industrial Applications of Linear Programming*, Volumes I and II, John Wiley and Sons, New York, 1961.

8. Dantzig, G., *Linear Programming and Extensions*, Princeton University Press, Princeton, 1963.

9. DeWitt, D., R. Finkel and M. Solomon, "The CRYSTAL Multicomputer: Design and Implementation Experience," Technical Report 553, Department of Computer Sciences, University of Wisconsin, Madison, 1984.

10. Dijkstra, E., "Cooperating Sequential Processes," Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.

11. Dongarra, J., F. Gustavson and A. Karp, "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Review* 26, 91-112 (1984).

12. Elam, J., F. Glover and D. Klingman, "A Strongly Convergent Primal Simplex Algorithm for Generalized Networks," *Mathematics of Operations Research* 4, 39-59 (1979).

13. Engquist, M. and M. Chang, "New Labeling Procedures for the Basis Graph in Generalized Networks," *Operations Research Letters* 4, 151-155 (1985).

14. Feijoo, B. and R. Meyer, "Optimization of the CRYSTAL Multicomputer," Technical Report 562, Department of Computer Sciences, University of Wisconsin, Madison, 1984.

15. Fourer, R., "Solving Staircase Linear Programs by the Simplex Method, 1: Inversion," *Mathematical Programming* 23, 174-313 (1982).

16. Fourer, R., "Solving Staircase Linear Programs by the Simplex Method, 2: Pricing," *Mathematical Programming* 25, 251-292 (1983).

17. Gill, P., W. Murray, M. Saunders, J. Tomlin and M. Wright, "A Note on Interior-point Methods for Linear Programming," *MPS Committee on Algorithms Newsletter* 13, 13-18 (1985).

18. Glover, F. and D. Klingman, "A Note on Computational Simplifications in Solving Generalized Transportation Problems," Transportation Science 7, 351-361 (1973).

19. Glover, F., J. Hultz, D. Klingman and J. Stutz, "Generalized Networks: A Fundamental Computer Based Planning Tool," *Management Science* 24, 1209-1220 (1978).

20. Glover, F., D. Klingman and J. Stutz, "Extension of the Augmented Predecessor Index Method to Generalized Network Problems," *Transportation Science* 7, 377-384 (1973).

21. Karmarkar, N., "A New Polynomial Time Algorithm for Linear Programming," *Proceedings of the 16th Annual ACM Symposium on the Theory of Computing*, 302-311 (1984).

22. Kennington, J. and R. Helgason, *Algorithms for Network Programming*, John Wiley and Sons, New York, 1980.

23. Klingman, D., A. Napier and G. Ross, "A Computational Study of the Effects of Problem Dimensions on Solution Times for Transportation Problems," *Journal of ACM* 22, 413-424 (1975).

24. Klingman, D., A. Napier and J. Stulz, "NETGEN: A Program for Generating Large Scale Capacitated Assignment, Transportation, and Minimum Cost Flow Problems," *Mangement Science* 20, 814-821 (1974).

25. Mulvey, J., "Pivot Strategies for Primal-Simplex Network Codes," *Journal of ACM* 25, 266-270 (1978).

26. Murtagh, B. and M. Saunders, "Large Scale Linearly Constrained Optimization," *Mathematical Programming* 14, 41-72 (1978).

27. Phillips, A. and J. Rosen, "Multitasking Mathematical Programming Algorithms," Technical Report, Department of Computer Science, University of Minnesota, Minneapolis, 1986.

28. Plummer, J., L. Lasdon and M. Ahmed, "Solving a Large Nonlinear Programming Problem on a Vector Processing Computer," Technical Report 85/86-3-1, Department of General Business, University of Texas, Austin, 1985.

29. Quinn, M., and N. Deo, "Parallel Graph Algorithms," *Computing Surveys* 16, 319-348 (1984).

30. Srinivason, V. and G. Thompson, "Benefit-Cost Analysis of Coding Techniques for the Primal Transportation Algorithm," *Journal of ACM* 20, 194-213 (1973).

31. Zenios, S. and J. Mulvey, "Nonlinear Network Programming on Vector Supercomputers," Technical Report EES-85-13, Department of Civil Engineering, Princeton University, Princeton, 1986.