World conf. on Systematics, Cybernetics and Information
Venezuela, 1997

NETWORKED COMPUTING FOR THE MAX-FLOW PROBLEM

C.N.Venkateswaran, K. Thulasiraman and A.Das
School of Computer Science
University of Oklahoma
Norman, Oklahoma.
{ thulasi@cs.ou.edu }

Abstract: The max-flow problem is a very fundamental one in network optimization. In view of the wide range of its applications, this problem has been studied extensively. Recently there has been considerable interest in designing efficient parallel algorithms for the max-flow problem. In this paper, we discuss an implementation of Goldberg and Tarjan's pre-flow push algorithm on a cluster of workstations using PVM.

I. Introduction

In the last few years or so, there has been an intensive interest in networked computing. This can be attributed to the fact that rapid advances in parallel computing technology have made available a number of parallel programming tools for implementing applications on a heterogeneous cluster of workstations at a low cost. By interconnecting computers in an organization and using parallel/distributed algorithms, one can solve efficiently several classes of problems whose complexity is beyond the ability of any one computer in terms of memory and computing requirements. In addition, effective utilization of computing hardware can be achieved. Parallel Virtual Machine (PVM) is one of the tools used in building a number of parallel applications. It is a software system that permits a network of heterogeneous computers to be used as a single large parallel computer and hence the term Networked Computing.

Significant progress has been achieved in the application of parallel computing for numerical problems. Such is not the case for problems such as network optimization problems which involve non-numerical computations. This can be mainly attributed to the fact that until recently, most computationally intensive applications have all been mainly of the numerical nature.

Network optimization, a branch of operations research, refers to the class of optimization problems defined on graphs or networks. These problems include the minimum cost flow problem, the max-flow problem, covering problems, the optimal assignment problems etc. These network optimization problems occur routinely as building blocks in designing efficient algorithms for more complex problems. The need for parallel algorithms for the max-flow problem and its variants is apparent when one sees the range of large engineering applications that can be modeled as a max-flow problem [1]. Though parallel algorithms have been developed for the max-flow problem, none of them have been implemented on the existing parallel tools like the Parallel Virtual Machine.

Motivated by the above considerations, this paper addresses issues in design and efficient implementation of parallel algorithms for the max-flow problem.

II. Parallel Network Optimization and the max-flow problem

In the recent past there has been an interest in designing and implementing parallel algorithms for network optimization and their applications [15], [16], [17], [2]. Banerjee discusses parallel algorithms for problems which arise in VLSI physical design [3].

Of special interest to us is the max-flow problem, which is a very fundamental problem in network optimization. It provides a link between operations research and graph theory. Many network and graph optimization problems can be formulated in terms of the max-flow problem. As a result, there has been considerable interest in designing efficient algorithms for this problem [14].

Ford and Fulkerson's labelling algorithm [8] provides an elegant approach to solving the max-flow problem. Edmonds and Karp [7] suggested an efficient implementation of this algorithm using shortest augmenting paths. Further improvements by Dinic [6] and Malhotra, Pramod Kumar and Maheshwari [13] led to an $O(n^3)$ algorithm for the max-flow problem, where n is the number of nodes in the network.

Most of the algorithms for the max-flow problem are variations of Ford-Fulkerson's original approach based on augmenting paths. Goldberg and Tarjan [9] proposed a new approach leading to the preflow-push algorithm. This algorithm is also of complexity $O(n^3)$. In addition it is amenable for a parallel/distributed implementation. References to other implementations of this algorithm leading to improved bounds may be found in [14].

In view of its importance, recently Anderson and Setubal [2] proposed a shared memory implementation of the preflow-push algorithm. They propose a modification where they use what is called a global relabeling technique. In this thesis, we consider implementations of this algorithm on a networked cluster of workstations.

2.1 The Max-Flow Problem

A transport network represents a model for transportation of a commodity from its production center to its market through communication routes. A *flow network* consists of a connected directed graph N=(V, E) with no self loops or parallel edges. N has to satisfy the following conditions.

- There is only one node with zero indegree; this is designated as the *source* and is denoted as s.
- There is only one node with zero outdegree; this is designated as the *sink* and is denoted as t.
- Every directed edge $e = (i,j)$ in N is assigned a non-negative real number c_{ij}, the capacity of (i,j). $c_{ij}=0$ if there is no edge directed from i to j.
- Every directed edge $e=(i,j)$ in N is also assigned a non-negative real number called the flow f_{ij} on (i,j).

A flow f through a transport network N is an assignment of non-negative real numbers f_{ij} to the edges (i,j) such that the following conditions are satisfied:

- *capacity constraint:* $0 \leq f_{ij} \leq c_{ij}$, for $(i,j) \in E$.
- *conservation constraint:* For each node i, except the source s and the sink t, the flow transported into i is equal to the flow transported out of i.
- The value val(f) of a flow f is defined as

$$val(f) = \sum_i f(s,i) = \sum_i f(i,t)$$

A flow f^* in a transport network N is said to be maximum if there is no flow f in N such that val (f) > val (f^*). The *maximum flow (in short, the max-flow) problem* is to find a maximum flow in a transport network.

2.2 The Push-Relabel Algorithm

We now present Goldberg and Tarjan's max-flow algorithm.

Let N=(V,E) be a network with each edge assigned a non-negative real capacity. Without loss of generality assume that N has no multiple edges. If there is an edge from a node i to a node j, this edge is unique by the assumption and is denoted by (i,j). A *psuedoflow* is a function f: E $\rightarrow$ R that satisfies the following constraints:

$f_{ij} \leq c_{ij}$, $\forall$ $(i,j) \in$ E (capacity constraint)

$f_{ji} = -f_{ij}$, $\forall$ $(i,j) \in$ E (antisymmetry constraint)

We let $c_{ij} = 0$ if $(i,j) \in E$. Given a pseudoflow f, the excess function e_f: $V \rightarrow R$ is defined by,

$$e_f(i) = \sum_{k \in V} f_{ki}$$

Thus $e_f(i)$ is the net flow into i. A node i has excess if $e_f(i)$ is positive. This indicates that some amount of flow can be pushed out from node i. A node i has *deficit* if $e_f(i)$ is negative.

A pseudo-flow f is a *preflow* if $e_f(i) \geq 0$ for every node other than s and t. Given a preflow f, let $N_f = (V, E_f)$ denote the residual graph with respect to f. Each edge $(i,j) \in E$ induces an edge $(i,j) \in E_f$, if $f_{ij} < c_{ij}$, and an edge $(j,i) \in E_f$ if $f_{ij} > 0$. Edges of N_f are all called *residual edges*. In the former case $(i,j) \in E_f$ is called the *forward edge* and in the latter case $(j,i) \in E_f$ is called a *backward edge*.

$$f_f(i,j) = c_{ij} - f_{ij} \quad \text{if } f_{ij} < c_{ij}$$
$$c_f(i,j) = f_{ij} \quad \text{if } f_{ij} > 0$$

The push-relabel algorithm of Goldberg and Tarjan starts with a preflow and a distance labeling, and uses two operations, *pushing* and *relabeling*, to update the preflow and the labeling, repeating them until a maximum flow is found. For a given preflow f, a valid distance labeling is a function d from the nodes to the non-negative integers such that $d(s) = n$, $d(t) = 0$ and $d(i) \leq d(j) + 1$ for all the residual edges (i,j).

A node i is said to be *active* if $i \notin \{s,t\}$ and $e_f(i) > 0$. An edge (i,j) is *admissible* if $c_f(i,j) > 0$ and $d(i) = d(j) + 1$.

The push-relabel algorithm begins with an initialization phase. The flow on each edge leaving the source is set equal to the edge capacity, and all other edges not incident on the source have zero flow. For each node j, the excess $e_f(j)$ is calculated. It is clear that since some flow is pushed from the source, there exists at least one node with positive excess. So there exists at least one active node. Each node $j \in V - \{s\}$ is assigned an initial labeling $d(j) = 0$. For node s, $d(s) = n$. Then an update operation is selected and applied to an active node. This process continues until there are no more active nodes at which point the algorithm terminates, with a preflow f with no active nodes. f is a maximum flow at termination.

We next present the update operations. The push operation modifies the preflow f and the relabel operation modifies the valid distance labeling d.

```
Push (i,j)
Applicability
        i is active, $c_f(i,j) > 0$ and
      $d(i) = d(j) + 1$.
Action
        send $\delta = \min(e_f(i), c_f(i,j))$
      units of flow from i to j;
        $f_{ij} \leftarrow f_{ij} + \delta$;
      $f_{ji} \leftarrow f_{ji} - \delta$;
      $e_f(i) \leftarrow e_f(i) - \delta$;
      $e_f(j) \leftarrow e_f(j) + \delta$;
```

```
Relabel (i).
Applicability
        i is active and $\forall\ j \in V$,
      $c_f(i,j) > 0 \Rightarrow d(i) \leq d(j)$.
Action
      $d(i) \leftarrow \min_{c_f(i,j) > 0} \{d(j) + 1\}$
      ( If this minimum is over an
      empty set, $d(i) \leftarrow \infty$ ).
```

An efficient implementation of the push/relabel algorithm is discussed next. In this implementation, an unordered pair $\{i,j\}$ such that $(i,j) \in E$ or $(j,i) \in E$ is an *undirected edge* of N. Each undirected edge $\{i,j\}$ is associated with three values c_{ij},

```
Push/Relabel (i).
Applicability
      i is active.
Action
      Let {i,j} be the current edge
    of i.
      if push(i,j) is applicable then
    push(i,j)
      else
        if {i,j} is not the last edge
         on the edge list of i
       then replace {i,j} as the
           current edge of i by the
        next edge on the edge
        list of $i$;
    else begin
            make the first edge
        on the edge list of $i$
        the current edge;
            relabel(i);
    end.
```

c_{ji} *and* f_{ij} . Each node i has a list of the incident edges $\{i,j\}$, in fixed but arbitrary order. Thus each edge $\{i,j\}$ appears in exactly two lists, the one for i and the one for j. Each node i has a *current edge* $\{i,j\}$ which is the current candidate for a pushing operation from i. The max-flow algorithm repeats the *push/relabel* operation until there are no more active nodes. The push/relabel operation combines the basic push and relabel operations.

In a parallel implementation of the push-relabel max-flow algorithm, the nodes perform in parallel the following pulses or phases (sets of sequential instructions) in that order:

- Push/Update
- Relabel, if necessary.

These pulses are repeated until termination. It should be ensured that relabeling is done only after all the nodes have completed the push operations. For this reason, the push/relabel operation as described above is not appropriate for use in a distributed environment. Furthermore, on the completion of push operations by the nodes, each node should have information about the flows pushed into itself from the adjacent nodes. This is essential to update the edge flows and to compute the excess flow at each node. This means that during a push/update pulse, each node should consider all incident edges for a possible push operation and inform the adjacent nodes about the flows pushed. These issues are taken into consideration in the presentation of the distributed max-flow algorithm described in the next chapter.

As regards termination, the algorithm may be terminated when there are no active nodes at the end of a push pulse. An alternative termination detection scheme based on the following theorem is more elegant. It is this scheme we have used in our algorithm development.

Theorem 1

If at any pulse, the total flow out from the source is equal to the total flow into the sink, then at that pulse and all subsequent pulses there will be no active nodes.

III. Parallel Virtual Machine

The **PVM** (Parallel Virtual Machine) provides a uniform framework within which large parallel systems can be developed in a straightforward and efficient manner. It allows a collection of heterogeneous machines on a network to be viewed as a general purpose concurrent computation resource.

The **PVM** system [4] is composed of a suite of user-interface primitives and supporting software that together enable concurrent computing on loosely coupled networks of processing elements. Several design features distinguish **PVM** from other similar systems. Among these are the combination of heterogeneity, scalability, multilanguage support, provisions for fault tolerance, the use of multiprocessors and scalar machines.

The **PVM** system is composed of two parts. The first part is a daemon, called *pvmd*, that resides on all the computers making up the virtual machine. When a user wants to run a **PVM** application, he executes pvmd on one of the computers which in turn starts up a pvmd on each of the computers making up the user-defined virtual machine. The **PVM** application can then be started from a Unix prompt on any of these computers.

The second part of the system is a library of **PVM** interface routines (libpvm.a). This library contains user callable routines for message passing, spawning processes, coordinating tasks, and modifying the virtual machine. Application programs must be linked with this library to use **PVM**.

IV Implementation

In this section, we describe the implementation details of the pre-flow push algorithm for the max-flow problem.

4.1 Graph generation and partitioning

Graph generation is the first step in our implementation of the max-flow algorithm. We have used two different graph generators in studying our implementations : NETGEN and Washington generators. NETGEN is a program which is used for generating a variety of network problems. This program was originally developed by a group of researchers at the University of Texas at Austin.

NETGEN [11] program can be divided into two main parts. The first part creates what is called the skeleton network and is concerned with obtaining the proper number of nodes of each type, insuring the correct total supply, and guaranteeing that the resulting problem will be connected and feasible. The second part completes the problem while insuring that the remaining specifications are met, such as total number of arcs, cost range, upper bound range, and percentage of arcs capacitated.

The other program which is used for generating graphs was developed at the University of Washington. This program can generate Mesh, Square Mesh, Random level graphs etc. Graph generators produce graphs in what is known as DIMAC's format.

Let $G = (V, E)$ be an undirected graph, where $V = \{ v_1, v_2, \ldots v_n \}$ is the set of n nodes, and $E \subseteq V \times V$ is the set of edges between the nodes. The graph partitioning problem is to divide the graph into disjoint subsets of nodes, such that the number of edges between the nodes in the different subsets is minimal, and the sizes of the subsets are nearly equal. The subsets are called *partitions*.

METIS is a package which provides high quality partitions and is extremely fast when compared to similar packages available [10] . METIS uses Multilevel recursive bisection algorithm to perform graph partitioning.

4.2 Slave/Slave Communication

For our implementation we use Slave/Slave communication model. In this model, there exists a process termed as the Master process. This Master process spawns a number of Slave processes based on the number of partitions of the given network. But, unlike the Master/Slave Communication model, in the Slave/Slave Communication model, the Slave processes communicate among themselves if they need any information instead of requesting the Master process. The Master process is used only for synchronization among the Slave processes. This reduces the excessive traffic between the Slave processes and the Master process. Figure Fig 4.1 below illustrates this model in more detail.

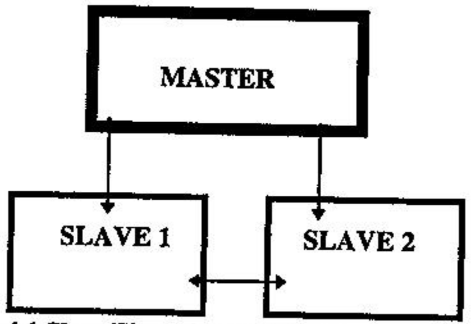

Fig 4.1 Slave/Slave Communication Model

4.3 Pseudocode

We next present the pseudocode for the parallel max-flow algorithm in the Slave/Slave communication model. The implementation has two different programs to deal with. One of them is called the Master program and the other the Slave program. Following the pseudocode, we have a brief description of what each program is intended to achieve.

```
              Algorithm Master
{ This code corresponds to the Slave/Slave
            Communication model }

Program Master()
```

```
{ main program }
    { Allocate memory for all the variables used }
    AllocateMemory()
    { Read the Graph }
    ReadGraph()
    { Convert the Graph from the DIMAC's format to the METIS format }
    NetToMetis()
    { Call the METIS package to partition the graph }
    PartitionGraph()
    { Get the partitioning information from METIS }
    GetPartition()
    { Spawn the Slave processes based on the number of partitions required }
    PvmSpawn()
    { Send the graph information to each slave process }
    SendGraphInfo()
    { This is the main loop of the Master program }
    While($totalinflow \neq totaloutflow$) do
        { Receive the totalinflow from the SINK node }
        RecTotInflow()
        { Receive the totaloutflow from the SOURCE node }
        RecTotOutflow()
        { Inform the Slaves to terminate once the totalinflow = totaloutflow }
        InformSlaves()
    End While
    { Get all the Timing information }
    GetTimerInfo()
    { Free the memory that has been allocated }
    FreeMemory()
    { Exit PVM and also the program }
    ExitAll()
End Program Master
```

Algorithm Slave

{ This code corresponds to the Slave/Slave Communication model }

```
Program Slave()
{ main program }
    { Allocate memory for the variables used }
    AllocateMemory()
    { Register with PVM }
    RegisterPVM()
    { Receive the graph information from the Master }
    ReceiveGraphInfo()
    { Initialize all the variables used }
    Initialize()
    { Start the Max-flow algorithm with the SOURCE node }
    StartWithSource()

    { Main loop of the Slave program }
    While (TRUE) do
        { Perform the Push operation for the nodes }
        Push()
        { Synchronize with the Master by waiting for all the Slaves to finish }
        Synchronize()
        { Send the information regarding Push to the Slaves }
        SendPushInfo()
        { Receive information regarding Push from other Slaves }
        ReceivePushInfo()
        { Update the flows }
        Update()
        { Synchronize with the Master by waiting for all the Slaves to finish }
        Synchronize()
        { Perform Relabel operation if required }
        Relabel()
        { Send the label information to the other Slaves }
        SendLabelInfo()
        { Receive the label information from the other Slaves }
        ReceiveLabelnfo()
        { If I am the SOURCE node, then send totaloutflow to the Master }
        SendTotalOutflow()
        { If I am the SINK node, then send the totalinflow to the MASTER }
        SendTotalInFlow()
```

```
                    { Wait for the signal from
the Master whether to terminate/proceed }
                    GetSignal()
                    { If signal is to quit then
compile the timing information and quit }
                    If ( signal = quit ) then
                            CompileTiming()
                            Quit()
                    End If
        End While
End Program Slave
```

4.4 Implementation Using Synchronizer

A synchronizer is a mechanism that helps simulate a synchronous communication network on an asynchronous one. The use of synchronizers for asynchronous communication networks was studied by Lakshmanan and Thulasiraman [12].

In the first implementation, we used the master process for synchronization. But, the master waits for the signal from all the slave processes before it allows them to proceed with the next phase. This problem is overcome in our second implementation wherein we use a synchronizer. We partition the original graph into a number of subgraphs and store these subgraphs in a processor. The slave program will be working on these subgraphs. For example, let us consider that we have 4 partitions. It may so happen that partition 1 does not have an edge connected to partition 2. In such a case, there is no need for the master process to wait for a signal from all the slave processes. It can go to the next pulse of its operation if it has computed its present pulse and all its neighbour slaves have completed their current pulse of operation. Thus the synchronizer helps some of the slave processes to go to the next phase instead of waiting for all the slave processes to start the next phase at the same time. We believe that such an implementation would be very efficient if we have a large number of partitions. The change we have made in our Master program to incorporate the synhronizer is indicated in Fig 4.2.

```
{ Synchronizer }
While( not received the signal from all the
slaves )
begin
        { Receive signal from any slave
processor }
        ReceiveSignal()
        { Check if all the adjacent slave
processes to a slave have sent a signal }
        CheckAdjacent()
        if ( Check is TRUE )
                { Send a signal to that slave
process to go ahead }
                StartWork()
        end if
end
```

Fig 4.2 The modified code to incorporate the Synchronizer in the Master Program

4.5 Implementation of Push and Relabel phases

In order to reduce the communication overhead we implemented the push and relabel phases as follows.

First, for each partition we first identify nodes which are connected to at least one node in another partition. These nodes will be called the border nodes. The other nodes will be called the internal nodes.

The push phase in each processor (or partition) is implemented as in [9] using a queue. During a push phase the border nodes are relabelled at most once. The push phase terminates when all the internal nodes become inactive. Only the information pertaining to the border nodes is sent to the other processors. It has been observed [2] that the preflow-push algorithm is slowed down by the excessively large number of relabel operations. In order to reduce this Setubal and Anderson [2] suggest what is called global relabelling. Since the labels of the nodes are estimates of distances to the sink or the source, they suggest using exact distance labels periodically and proceed with the algorithm using these exact labels. These exact labels are calculated by a backward BFS from the sink and assign the distances as the labels of the nodes from which the sink can be reached. Next another backward BFS is conducted to calculate the distances of those nodes which could not be labelled in the BFS from the sink. The labels of these nodes will be their distances plus n. In our implementation global relabeling is done after every n/2 or n/4 or n/8 push operations.

We have access to a network containing 40 workstations. Evaluating our implementation using this cluster is in progress. Preliminary results indicate good performance for large graphs.

REFERENCES

[1] Ahuja, R.K., T.L. Magnanti, and J.B. Orlin, "Network Flow: Theory, Algorithms, and Applications," *Prentice-Hall*, Englewood Cliffs, NJ, 1992.

[2] Anderson, R., and J.C. Stubal, "A Parallel Implementation of the Push-Relabel Algorithm for the Maximum Flow Problem," *Journal of Parallel and Distributed Computing*, Vol.29, 17-26, 1995.

[3] Banerjee, P., "Parallel Algorithms for VLSI Computer-Aided Design Applications," *Prentice-Hall, Inc.*, 1994.

[4] Beguelin, A., J.J. Dongarra, G.A. Geist, R. Manchek, and V.S. Sunderam, "A Users' Guide to PVM Parallel Virtual Machine," *Oak Ridge National Laboratory*, September, 1994

[5] Chvatal, V., "Linear Programming, " *Freeman Company*, Potomac, Maryland, 1983.

[6] Dinic, E.A., "Algorithm for the Solution of a Problem of Maximum Flow in a Network with Power Estimation," *Soviet Math., Dokl.*, Vol.11, 1277-1280, 1970.

[7] Edmonds, J., and R.M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. Assoc. Comput. Mach*, Vol. 19, 248-264, 1972.

[8] Ford, L.R., and D.R. Fulkerson, "Flows in Networks," *Princeton University Press*, Princeton, N.J., 1962.

[9] Goldberg, A.V., and R.E. Tarjan, "A New Approach to the Maximum Flow Problem," *Journal of the ACM*, Vol.35, 921-940, 1988.

[10] Karypis, G., and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Technical Report*, Department of Computer Science, University of Minnesota, 1995.

[11] Klingman, D., A. Napier and J. Stutz, "NETGEN: A Program for Generating Large Scale Capacitated Assignment, Transportation and Minimum Cost Flow Problems," *Management Science*, Vol.20, 814-821, 1974.

[12] Lakshmanan, K.B., and K. Thulasiraman, "On the Use of Synchronizers for Asynchronous Communication Networks," *Proc. II Intl. Workshop on Distributed Algorithms*, Amsterdam, July 1987.

[13] Malhotra, V.M., M. Pramodh Kumar, and S.N. Maheswari, "An O(v3) Algorithm for Maximum Flows in Networks," *Information Proc. Letters*, Vol.7, 277-278, 1978.

[14] Thulasiraman, K., and M.N.S. Swamy, "Graphs: Theory and Algorithms," *Wiley-Interscience*, New York, 1992.

[15] Thulasiraman, K., R.P. Chalasani, and M.A. Comeau, "Parallel Network Dual Simplex on a Shared Memory Multiprocessor," *Proc. 5th IEEE Symp. on Parallel and Distributed Processing*, Dallas, 408-415, 1993.

[16] Thulasiraman, K., R.P. Chalasani, P. Thulasiraman, and M.A. Comeau, "Parallel Network Primal-Dual Method on a Shared Memory Multiprocessor and A Unified Approach to VLSI Layout Compaction and Wire Balancing," *Proc. VLSI Design '93*, Bombay, India, 242-245, Jan. 1993.

[17] Thulasiraman, K., M.A. Comeau, R.P. Chalasani, A. Das, and J.W. Atwood, "On the Design of a Parallel Algorithm for VLSI Layout Compaction," *Proceedings of International Symposium on Circuits and Systems*, ISCAS 90, 352-355, 1990.