Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

Parallel Flow-Constrained Multicast Algorithms

Longsong Lin, Lih-Chyau Wuu, Yin-Fu Huang

Department of Electronic and Information Engineering,
National Yunlin Institute of Technology, Toulin, Taiwan, R.O.C.

Abstract

Multicast is an important operation in communication networks and its attainment relies on the performance of the underlying multicast routing algorithm. The multicast routing problem is well known to be NP-Hard and many heuristic algorithms have been developed. Most heuristic algorithms take time to compute the MST(Minimum Spanning Tree) of some specified nodes. Thus, the heuristic performance can be improved by exploiting parallelism within the algorithms when executed in a network node. In this article we investigate efficient parallel implementations for four MST algorithms – the Esau-Williams's, Prim's, Kruskal's, and Sollin's algorithms. The optimization goal of these algorithms is not only to construct a minimum cost spanning tree for a designated group of nodes, but also to satisfy the maximum flow constraint on each link. The corresponding parallel programs are implemented with message-passing mechanism in the Parallel Virtual Machine (PVM) environment.

1. Introduction

Multicasting is the simultaneous distribution of data to multiple destinations. To support multicast, many communication algorithms have been extensively studied to facilitate the delivery of data to a group of destinations [2][9][11][12]. These algorithms are intended to solve this concurrent data transmission problem by constructing a spanning tree for the destination nodes in the given network [7][10]. A basic performance guideline on designing the type of algorithm is to deliver the data within a specified time – with bounded delay for each destination [3][12]. Furthermore, the cost of the spanning tree is another objective to consider when the network has only limited bandwidth. The cost of a spanning tree is defined as the sum of individual link cost, which can be the operation cost, the capacity of the link, type of connection and line leased, or distance between the nodes, etc. Developing a spanning tree by simultaneously optimizing cost and time inevitably involves trade-off between network cost and performance for the underlying application. This thus entails careful consideration of the constraints imposed by the limited network bandwidth and quality requirement of the underlying application [8][12].

The optimization goals and constraints imposed by an application determine different levels of computational complexity in the tree-construction algorithm. For instance, if the link delay is considered to be minimized, a minimum delay tree for n nodes can be constructed in $O(n^2)$ time using Dijkstra *shortest path algorithm* [4][7]. If the cost of the tree is the optimization goal, then the tree is a minimum cost spanning tree called *Steiner tree* [10]. The problem of finding such a Steiner tree is not tractable in polynomial time. Heuristics for approximate Steiner trees have been proposed that produce near-optimal trees quickly [2][5][9][10][12]. The algorithms take polynomial time, ranging from $O(n^3)$ to $O(n^4)$. Nevertheless, the cost produced by these algorithms are almost twice the cost of the optimal solution.

For most multicast problems, although time delay is a common goal to be optimized, the time delay is nevertheless difficult to quantify. Instead we shall focus on a flow constraint: the flow in any particular link shall not exceed a specified maximum value. This of course implies a maximum link time delay on each link in the network. Given such a network, we aim at devising the multicast algorithm that constructs a minimum cost spanning tree with the flow on each link not exceeding the specified flow constraint. The cost of a link between any pair of nodes is given by the a symmetrical cost matrix C. Specifically, the traffic generated per unit time at each node is represented by λ/μ where λ is in unit of message/sec and $1/\mu$ is in bits/message. The constraint of link flow is assumed to be Λ(in bits/sec). Hence the algorithm must to ensure that $\lambda/\mu<\Lambda$ while minimizing the $\sum_{(i,j)\in T} C_{ij}$ where the C_{ij} is the cost of link (i, j) on the spanning tree T.

We shall introduce four algorithms to solve this problem[6][13][14][16]. These multicast-tree constructing algorithms require a multitude of computations especially when the network scale is extensive. In general, these algorithms requires four major steps of computing in common: 1) initialize the values of a trade-off function, 2) find the minimum from these values, 3) check constraints, and 4) union links as well as update trade-off values. We thus conceive that the effort is mainly on computing the trade-off values and finding the minimum among them. If the multicast tree spanning n nodes, then there are potentially $n(n-1)/2$ values to be examined. Apparently, for large n these steps can be very time consuming. To reduce the computational work in the algorithms, more efficient techniques are demanding. Parallelizing the algorithm is a viable approach to shorten the execution time by exploiting the parallelism within a sequential algorithm. We shall focus on the implementations of the parallel counterparts for the algorithms in this work.

The parallel algorithms are implemented in a cluster of SunSparc 10 computers that are interconnected through a 100-BaseVG Ethernet. The cluster of workstations is installed a PVM (Parallel Virtual Machine) [1] library that uses explicitly message passing mechanism to communicate between processes. PVM was originally implemented for the design of task parallel programs in distributed memory parallel architectures. These multicast algorithms are parallelized by explicitly inserting PVM's message-passing procedure calls whereupon synchronization and communication are needed. Although in this cluster computing system the inter-process communication is relatively "costly" than the overhead incurred in a tightly-coupled processor system such as shared-memory parallel computing, we expect that the advance of high-speed network has overcome this gap. The rationality to employ PVM message passing method to simulate the inter-process communication in our research is for fitting the algorithms into the distributed network applications in the future.

Primarily, there are four classes of heuristics for computing non-constrained minimum spanning tree of a weighted graph in a serial computing environment. There are Prim's algorithm [14] and Esau-Williams's [6] algorithms based on the nearest neighbor method; the Kruskal's algorithm [13] based on the minimum weighted edge first method; and the Sollin's algorithm [16] based on the minimum weighted edge from each node method. Many works have concentrated on the complexity analysis by assuming certain types of computing model such as EREW, CREW, or TSC etc [15]. Our work instead focuses on the implementation techniques and quantifies the overhead. Our goal is to envisage the significance of parallel implementation [15] for the multicast algorithms and to provide an insight to the suitability of the algorithms to parallelize.

This article is organized as follows. After this introductory section is a formal presentation of the multicast algorithms to be discussed in the paper. In Section 3, parallelized versions for the four algorithms are proposed and the corresponding complexity results are examined. In Section 4, the results of simulating the parallel algorithms on the PVM architecture are described. Finally in Section 5 a conclusion is drawn for this research.

2. Flow-Constrained Multicast Algorithms

The problem considered in this paper is the generation of a sink-rooted multicast tree. Given a graph G=(V, E), |V|=n, |E|=m, a sink node $s \in V$, two functions

(1)Flow function F: V → N gives the traffic or average number of data units per unit time generated at each node $v \in V$,

(2)Cost function C: E → N gives the cost of each link $e \in E$,

a sink-rooted multicast tree is a minimum cost spanning tree T=(V, E'), E'⊆E, and the flow on each link does not exceed the specified flow constaint.

In this section we shall examine four types of multicast algorithms. The description of each algorithm is depicted in Figure 1. It shows the pseudo code for each the of sequential algorithms. We brief the property for each algorithm as follows.

1. Esau-Williams algorithm searches out the nodes that are furthest from the source and connects them to neighboring nodes that provide the greatest cost benefit.
2. Prim algorithm does the reverse: initially it select the node closest to the source, then connects in those nodes that are closest to those already in the network.
3. Kruskal algorithm simply connects the least-cost links, one at a time.
4. Sollin algorithm initially treats each node as a forest. Then each forest connects to its neighbor forest that provide the best cost benefit regardless the distance of the link to the source. The procedure repeats until all the nodes are connected in the multicast tree.

Each of the four algorithms typify a type of multicast algorithm in the way the multicast tree is growing. From the view of a source node, the Esau-Williams algorithm grows the tree outside-in until the source node is connected lastly whereas the Prim algorithm grows inside-out until the furthest nodes are connected. The Kruskal algorithm connects links in increasing order of their costs; while Sollin algorithm connects nodes without considering any order on link cost.

Observing the algorithms, it is found that four common operations are involved in the algorithm: INIT initializes node weight and trade-off functions, MIN finds minimum of the trade-off functions and checks the constraint, UNION unions nodes and links into present multicast tree, and UPDATE updates node weights and trade-off functions. Remember the outer loop is a while loop that executes theses four steps in at most *log n* times for the Sollin algorithm and *n* times for the Esau-Williams and Prim algorithms. As for the Kruskal, the outer loop executes m=$O(n^2)$ times at the worst case. The time complexity of inner loop in each algorithm is presented in Table 1. It is noted that the Kruskal algorithm needs m=$O(n^2)$ steps to execute MIN operation owing to using a sequential search method to find a minimum cost edge among m edges.

Table 1. Complexities of inner loop for the sequential multicast algorithms

Operation	INIT	MIN	UNION	UPDATE
EW	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n)$
Prim	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Kruskal	$O(1)$	$O(n^2)$	$O(1)$	$O(n)$
Sollin	$O(n)$	$O(n^2)$	$O(n)$	$O(n)$

It is therefore not difficult to check the complexity for each algorithm:

1. Esau-Williams: $O(n^3)$
2. Prim: $O(n^2)$
3. Kruskal: $O(n^4)$
4. Sollin: $O(n^2 \log n)$

3. Parallel Multicast Algorithms

Figure 2 delineates the parallel version for each algorithm. The source of parallelism lay in the trade-off function portion of the code. The corresponding complexity for each of the parallel algorithms is presented in Table 2 by assuming there are p processes involved in solving the problem.

Table 2. Complexities of inner loop for the parallel multicast algorithms

Operation	INIT	MIN	UNION	UPDATE
EW	$O(n^2)/p + f(p)$	$O(n^2)/p + f(p)$	$O(1)/p + f(p)$	$O(n)/p+f(p)$
Prim	$O(n)/p+ f(p)$	$O(n)/p + f(p)$	$O(1)/p + f(p)$	$O(n)/p+f(p)$
Kruskal	$O(1)/p+ f(p)$	$O(n^2)/p + f(p)$	$O(1)/p + f(p)$	$O(n)/p+f(p)$
Sollin	$O(n)/p+ f(p)$	$O(n^2)/p + f(p)$	$O(n)+ f(p)$	$O(n)/p+f(p)$

Yoo [19] has found that the minimum-cost spanning tree of a weighted, undirected, connected graph with m edges can be found in time $O(m)$ on a UMA (Uniform Memory Access) multiprocessor with $\log m$ processors. For n nodes, there is at most $n(n-1)/2$ edges in the graph, the complexity is thus $O(n^2)$ for n processors. In our implementations, the architecture is however a multicomputer; each processor has its own private memory and process interaction occurs through message passing.

Two points should be emphasized here. The function $f(p)$ is the overhead incurred in the message passing between p processes. The function is varied in according to the number of processors (computers) in the PVM setup for a run. PVM can specify a number of processes to be in a processor in which communication is "cheaper" than that of processes between processors. The factor p in the UNION operation accounts for the time, in the worst cast, a processes wanting to lock a particular tree A have to wait for every other processor to lock and unlock tree A. This is a special scenario in the Sollin parallel algorithm.

Consider the situation depicted in Figure3. Suppose one process is attempting to connect tree A with closest neighbor B, while another processes is attempting to connect tree B with its closest neighbor. A variable *edge[A]* contains edge $\{vA, wA\}$ with length k, Variable *edge[B]* contains edge $\{vA, wB\}$ also with length k. If both processes perform the test function FIND for finding the tree for node v and w before either process performs the UNION operation, then both edges will be added to the existing connected tree T. Therefore, if the code is to be made parallel, trees FIND(v) and FIND(w) must be locked by one of the process and unlocked by that process so as to let the other processes to access this critical section.

From the table we can summarize the complexities of the parallel algorithms as follows. We found that the Sollin algorithm seems to be the most suitable for parallelization. We will verify this in later section.

1. EW: $O(n(n^2/p + n/p + f(p)))$
2. Prim: $O(n(n/p + f(p)))$
3. Kruskal: $O(n^2(n^2/p + n/p + f(p)))$
4. Sollin: $O(\log n\,(n^2/p + n/p + n + f(p)))$

4. Simulation Results

The greedy heuristics described in this article are evaluated by simulation. The performance is evaluated using random graphs as the network model. We attained qualitatively similar results for different random graphs. For a given number of nodes, the results presented here for a fixed random graph as the network. The graphs used were designed to be sparse with the average degree being less than five to capture the flavor of realistic network topology such as NSF backbone. Group members are picked uniformly from the set of nodes in the graph.

The random graphs used in the simulation are constructed using the method proposed in [18]. The n nodes of a graph are randomly distributed on a Cartesian coordinate grid with unit spacing. The (x,y) coordinates of each node was selected uniformly from integers in $[0, n]$. Considering all possible pairs of node, edges are placed connecting nodes with the probability

$$P(u,v) = \beta \exp\left(\frac{-d(u.v)}{\alpha L}\right)$$

where $d(u, v)$ is the Manhattan distance between nodes u and v, and L is the maximum possible distance between two nodes. The parameters α and β are in the range $(0, 1)$ and can be selected to obtain desired characteristics in the graph. For example, a large β gives nodes with a high average degree, and a small α gives long connections. It has been observed that with appropriate parameters, this method gives networks that resemble "real networks". The parameters α and β are varied to obtain appropriately sparse networks. According to our experiments, for $\alpha = 0.25$ and $\beta = 0.2$, the random graph generated will have average degree of node less than or equal to five, which is the same result obtained in the work [5].

The cost of each edge was set to the Manhattan distance between its endpoints plus one. By adding one to the Manhattan distance, the uninteresting case of zero edge is eliminated. Our experiments were run on to the maximum of 150-node graph.

Figure 4 depicts the algorithm for generating the random graph. Initially the cost matrix *graph[][]* is assigned a maximum cost *MAXINT* for each link. Then for each link (x, y), a probability function *probfunc(x, y)* which is the equation mentioned above produce a value to be test for connectivity. The cost of a link is computed by a distance function *costof(x, y)*. This value is assigned to the link if and only if the value produced by the random function *random(100)* falls in the range generated by the function *probfunc(x, y)*. For example, if the *probfunc(1, 4) * 100* is 50 and the *random(100)*100* is 30, then the cost *costof(1, 4)* which is 14 is assigned to link *(1, 4)*, i.e., *graph[1][4]*=14.

Figure 5 shows the cost found by each parallel algorithm. In the beginning of the graph wherein the number of nodes is below 40, the tree found is a minimum spanning. This is because that the constraint $\Lambda = 70$ is high with respect to the small number of network nodes. As the number of node increases, the cost increases accordingly. The figure shows that the Esau-Williams parallel algorithm obtains the lowest cost while the Kruskal algorithm has the highest cost among them. The Kruskal algorithm has highest cost in that the link costs in the random graph have large variation.

Figure 6 shows the problem salability for each algorithm. It is found that the only algorithm that scales well is the Sollin algorithm. As the number of nodes in the random graph increases, the Sollin algorithm maintains the total execution time under certain level. On the other hand, the time by using the Kruskal algorithm increases tremendously for the increase of problem size. The Esau-Williams and Prim are about average among them, with the Prim algorithm slightly better than the Esau-Williams algorithm.

In Figure 7, the scalibility with respect to the number of processes spawned is depicted. The figure shows that except for the Sollin algorithm the other three algorithms produce minimum total execution when the number of processes in under 10. This is because using PVM incurs costly inter-process communication. Even the number of process increases, the speedup gained is still overwhelmed by the communication overhead between these processes. The Sollin algorithm produces optimal execution time when the number of processes is around 20. The curve for execution time is flat out in that the communication overhead almost counteracts the speedup gained by the increase of processes.

5.Conclusion

We have shown the suitability of parallelization for four algorithms on the flow-constrained multicast problem. Our experiments with the four parallel multicast algorithms show that the Sollin algorithm is best candidate for parallel implementation. The Sollin parallel algorithm scales best with respect to the problem size and the number of processes. The Esau-Williams and Prim parallel algorithms on the other hand are able to find smallest cost spanning tree for a constrained multicast problem providing that a feasible solution can be found. The Kruskal algorithm has the worst time and cost among the algorithms for parallel implementations with longest total execution time and cost.

As we pointed our earlier in this article, the cluster computing system incurs larger inter-process communication overhead than that of a tightly-coupled processor system such as shared-memory parallel computing. Although the speedup gain is relatively limited by virtue of the message passing implementations, we expect the performance can be further improved if using high speed networks as the test bed. At any rate, this study should provide valuable insight for the implementations of these network algorithms in the "real world" message-passing distributed environment.

References

[1] Beguelin, B., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V., PVM 3 USER'S GUIDE AND REFERENCE MANUAL, Engineering Physics and Mathematical Science Section, May 1993.

[2] Chow, C.H., *On Multicast Path Finding Algorithms.* in Proc. IEEE INFOCOM '91 New York, NY, pp. 1274-1283, 1991.

[3] Deering, S. and Cheriton, D., *Multicast Routing in Internetworks and Extended Lans.* ACM Trans. on Comp. Sys., vol. 8, pp. 85-110, May 1990

[4] Dijkstra, E., *Two Problems in Connection with Graphs.* Numer. Math. 1 (1959), pp. 269-271

[5] Doar, M. and Leslie, I., *How bad is Naïve Multicast Routing.* IEEE University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, U.K 1993.

[6] Esau, L. R. and L. C. Williams, *A Method for Approximating the Optimal Network.* IBM Systems J., 5, no. 3, 1966, 142-47.

[7] Floyd, R.W., Algorithm 97: *Shortest Paths.* Commun. ACM, vol. 5, pp. 345, 1962.

[8] A. Frank, L.. Wittie, and A, Bernsstein, *Multicast Communication in Network Computers.* IEEE Software, vol. 2, no. 3, pp. 49-62, 1985

[9] Garey, M. and Johnson, D., *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York : W.H Freeman and Co.,1979

[10] Hakimi, S., *Steiner's Problem in Graphs and Its Implications.* Networks, vol. 1, pp. 113-133,1971

[11] Karp, R., *Reducibility among Combinatorial Problems.* in Complexity of Computer Computations, pp.85-103, New York: Plenum Press, 1972.

[12] Kompella, V., Pasquale, J. and Polyzos, G., *Two Techniques for Multicast Routing for Multimedia Networking.* Tech Report CSL-1005-91, Computer Systems Laboratory, University of California, San Diego, Dec 1991.

[13] Kruskal, J. B. Jr., *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem.* Proc. of the AMS, vol. 7, no. 1, pp. 48-50, 1956.

[14] Prim, R. C., *Shortest Connection Networks and Some Generalization.* Bell System Technical Journal , vol. 36, pp. 1389-1401, 1957.

[15] Quinn, M.J., *Parallel Computing Theory and Practice.* McGraw-Hill International Editions

[[6] Sollin, M. 1977 . *An Algorithm Attributed to Sollin. In S. E.* Goodman and S. T. Hedetniemi, eds. Introduction to the Design and Analysis of Algorithms, McGraw-Hill, New York, sec. 5.5.

[17] Wall, D., *Mechanisms for Broadcast and Selective Broadcast.* Ph.D thesis, Elect. Engin. Dept., Stanford University , Jun. 1980

[18] Waxman, B.M., *Routing of Multipoint Connections.* IEEE J. Selected Areas in Comm., Vol. 6 , No. 9 , December 1988.

[19] Yoo, Y. B. 1983. *Parallel Processing for Some Network Optimization Problems.* Ph.D. dissertation, Computer Science Dept., Washington State University, Pullman.

```
1.  BEGIN :
2.    E'←φ
3.    FOR i←1 TO n DO                        /* INIT */
4.       FOR j←1 TO n DO
5.          compute $t_{ij} = c_{ij} - c_{i1}$
6.    WHILE (|E'|<n-1) DO
7.      MinimumEdge←∞
8.      FOR i ←1 TO N DO                     /* MIN */
9.         FOR j←1 TO N DO
10            MinimumEdge←min(MinimumEdge, $t_{ij}$)
11.     IF add MinimumEdge has NoCycle and
12.           NoInfeasibleFlowConstraint then
13.        add MinimumEdge to tree        /*UNION*/
14.        update flow of tree           /* UPDATE*/
15.     ELSE abort this MinimumEdge
16.   ENDWHILE
17.  END
```

(a) Sequential Easu-Willams Algorithm

```
1. E' ← ∅
2. TC ← 0
3. TF ←0
4. N[1] ←0
5. BEGIN
6. FOR I=2 TO N DO                              /* INIT */
7.    NB[I] ←0
8.    D[I] ←C[1,I]
9. ENDFOR
10. WHILE |E'| < N-1 DO
11.   select j (D[j]=min{D[I] | NB[I]<>0}) AND   /* MIN */
            (TF + Flow[j] < F)
12. E' ← E' UNION {(j , NB[j])}         /*UNION*/
13. TC ←TC + D[j]                      /*UPDATE*/
14. TF ← TF + Flow[j]
15. FOR I=1 TO N DO
16.    IF(NB[I] <> 0) AND (C[I , j] < C[I , NB[I]])
17.      THEN {  N[I]←j
18.              D[I] ←C[I , j]   }
19     ENDIF
20.   ENDFOR
21.  ENDWHILE
22.  END
```

(b) Sequential Prim's Algorithm

```
1.  BEGIN
2.  E'←φ                                        /*INIT*/
3.  WHILE (|E'|<n-1) DO
4.   find the least cost edge e in E           /*Min*/
5.    using one-by-one edge comparing method
6.   IF add e has NoCycle and
7.   NoInfeasibleFlowConstraint then
8.     ADD e into E'; Delete e from E      /*UNION*/
9.     update flow of tree               /* UPDATE*/
10.  ENDWHILE
11.  END
```

(c)Sequential kruskal Algorithm:

```
1.  BEGIN
2.     FOR i←1 TO n DO                         /*INIT*/
3.        vertex i is initially in set i.
4.     T← ∅
5.     WHILE |T|<n-1 DO
6.       FOR every tree i DO   close[i] ←∞
7.       FOR every edge {v,w} DO              /*MIN*/
8.          IF FIND(v)<>FIND(w) THEN
9.            IF weight{[v,w]<close[FIND(v)] THEN
10.              close [FIND(v)] ←weight{[v,w]}
11.              edge[FIND(v)] ←{v,w}
12.           ENDIF
13.         ENDIF
14.      ENDFOR
15.      FOR every tree i DO               /*UNION*/
16.        (v,w) ←edge[i]
17.        IF FIND(v)<>FIND(w) then
18.          T←T∪{(v,w)}; UNION(v,w)
19.          update flow of tree          *UPDATE*/
20.        ENDIF
21.      ENDFOR
22.    ENDWHILE
23.  END
```

(d) Sequential Sollin Algorithm

Figure 1. Four multicast algorithms.

```
 1.  all Tij of node i.         Ti
 2.  number of processor        Nproc
 3.  tree                       T
 4.  Cij-Ci1                    Tij
 5.  BEGIN :
 6.  spawn (Nproc)
 7.  FOR i←1 TO N DO
 8.    FOR j←1 TO N DO
 9.         compute Tij
10.  WHILE (|T|<N-1) DO
11.     FOR every processor DO
12.        send (Ti)
13.     FOR every processor DO
14.        MiniEdgeOfNode←receive(minimum-edge)
15.     MinimumEdge←min(MiniEdgeOfNode)
16.     IF add MinimumEdge has NoCycle and
17.         NoInfeasibleFlowConstraint THEN
18.                  add MinimumEdge to tree
19.                  update flow of tree
20.     ELSE   abort this MinimumEdge
21.  ENDWHILE
22.  END
```

(a) Parallel Easu-Willams Algorithm

```
 1. BEGIN
 2. FOR j=1 TO Nproc DOPAR
 3.    FOR all nodes u ∈ Pj DO
 4.        IF u<> 1
 5.        THEN NB[u]  ← 1
 6.        ENDIF
 7.    ENDFOR
 8. ENDDOPAR
 9. FOR I=1 TO N-1 DO
10.   FOR j=1 TO Nproc DOPAR
11.       Pj finds x and y , x ∈ Pj such that
                               C[x,y]=min{C[p,q]|
12.       (p ∈ Pj but not yet in the tree) AND (N[p]=q)}
13.       D[j]=C[x , y] ; A[j]=x ; B[j]=y
14.   ENDDOPAR
15.   find t (D[t]=min{D[j] | j=1,2,...,Nproc}) AND
                                    (TF+Flow[A[t]]≤ F)
16.   E'  ← E' UNION (A[t] , B[t])
17.   broadcast A[t] to all processors Pi , 1≤i≤Nproc
18.   FOR j=1 TO Nproc DOPAR
19.       IF A[t] ∈ Pj
20.       THEN mark A[t] as a node already in the tree
21.       ENDIF
22.       FOR v ∈ Pj DO
23.           IF v is not in the tree
24.           THEN IF C[v , A[t]]<C[v , N[v]]
25.                   THEN   N[v]  ←  A[t]
26.                   ENDIF
27.           ENDIF
28.       ENDFOR
29.   ENDDOPAR
30.ENDDO
31.END
```

(b) Parallel Prim's Algorithm

```
1.  BEGIN
2.  E'← ∅
3.  WHILE |E| <> |N|-1
4.    find the least cost edge e in C among G
      // use client-server model
      first, divide graph into pnum pieces Cn' and
      send them to all children (servers actually), one
      child one pieces, i.e. C=C1'∪C2'....∪Cpnum'
      then, send commands to ask for least edge and
      set masks to prevent replicated edge searching.
5.    check whether cycle will happen after adding e
6.    IF no cycle Then add e into E
7.  ENDWHILE
8.  END
```

(c) Parallel kruskal Algorithm

```
 1.  tree          T
 2.  Processor     NProc
 3.  BEGIN
 4       i=1;j=1
 5.      WHILE (|T|<n-1) DO
 6.          spawn(SlavePro,NProc)
 7.          FOR every NProc DO
 8.              send(cost,forest[i])
 9.              send(flow,forest[i])
10.              i←i+1
11.          ENDFOR
12.          FOR every NProc DO
13.               receive(best-outgoing-edge,forest[j])
14.               j←j+1
15.               IF NoCycleCause then
16.                   Add best-outgoing-edge to Tree
17.               ELSE   abort this best-outing-edge
18.          ENDFOR
19.          update every forest's total flow
20.      ENDWHILE
21.  END
```

(d) Parallel Sollin Algorithm

Figure 2 The four parallel multicast algorithms.

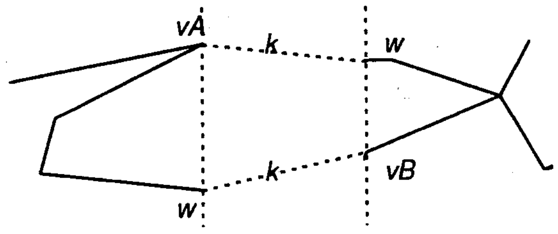
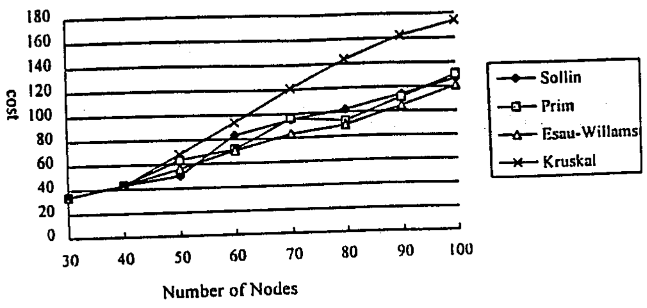


Figure 3, Locking critical Section

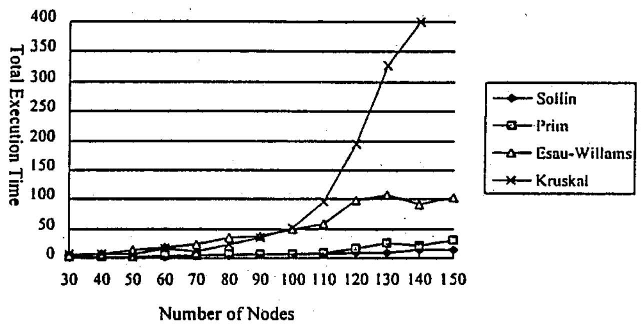


For Flow Bound=70
Figure 5. Cost of the multicast tree for each of the parallel algorithms.

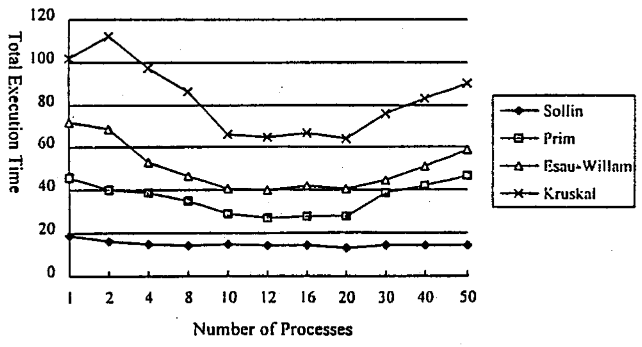
Random graph generator algorithm:

```
1    //begin of program
2    alpha = 0.25; beta = 0.2;
3    graph[][] ← MAXINT
4    //refresh graph array into disconnected value
5    for all nodes on coordinator (x,y) {
6        if (x==y) graph[x][y] = 0; // means no cost in
                                        self-loop
7          prob ← probfunc(x,y);
8.   //probfunc(x,y)=beta*exp((-1)*dist(x,y)/
                 (alpha*2* #(node)))
9        if (prob==CONNECTED)
         then graph[x][y] = costof(x,y);
10    //costof() measures the distance between (x,y)
11       if random(0..100) in [0..probfunc(x,y)*100 ]
         then prob will return CONNECTED
12       make sure each column & each row has
         at least one connected edge
13    //this can ensure G will be a connected graph
14    } // end of for-loop
15 }    // end of program
```

Figure 4. The algorithm for generating a connected random graph.



For Flow Bound=200
Figure 6. Show the scalibility to problem size for each algorithm.



For Number of Nodes=150 , Flow Bound=200
Figure 7. Scalibility to the number of processes for each algorithm