Parallel Algorithms for the Maximum Flow

by

Zair Abdelouahab

A Thesis Submitted in Fulfillment of the Requirements for the Degree of Master of Science in the Department of Computing Science at the University of Glasgow

October 1988

ACKNOWLEDGEMENTS

I am greatly indebted to my supervisor Professor A. C. ALLISON. He was a continuous source of inspiration and advice. His keen interest and his thoughtful comments brought this thesis in its present shape. Professor D. C. GILLES deserves credit for accepting me to study in this department. I would like to express my deep gratitude and all my respect to the Algerian Government for their financial support and the opportunity he gave me to study for this postgraduate degree.

Finally, I wish to thank my family and all my friends for their help, assistance and encouragement.

Summary

The problem of finding the maximal flow through a given network has been intensively studied over the years. The classic algorithm for this problem given by Ford and Fulkerson has been developed and improved by a number of authors including Edmonds and Karp. With the advent of parallel computers, it is of great interest to see whether more efficient algorithms can be designed and implemented.

The networks which we will consider will be both capacitated and bounded. Compared with a capacitated network, the problem of finding a flow through a bounded network is much more complicated in that a transformation into an auxiliary network is required before a feasible flow can be found.

In this thesis, we review the algorithms of Ford and Fulkerson and Edmonds and Karp and implement them in a standard sequential way. We also implement the transformation required to handle the case of a bounded network.

We then develop two parallel algorithms, the first being a parallel version of the Edmonds and Karp algorithm while the second applies the Breadth-First search technique to extract as much parallelism as possible from the problem. Both these algorithms have been written in the Occam programming language and implemented on a transputer system consisting of an IBM PC host, a B004 single transputer board and a network of four transputers contained on a B003 board supplied by Inmos Ltd. This is an example of a multiprocessor machine with independent memory.

The relative efficiency of the algorithms has been studied and we present tables of the execution times taken over a variety of test networks.

The transformation of the original network into an auxiliary network has also been implemented using parallel techniques and the problems encountered in the development of the algorithm are described.

We have also investigated in detail one of the few parallel algorithms for this problem described in the literature due to Shiloach and Vishkin. This algorithm is described in the thesis. It has not been possible to implement this algorithm because it is specifically designed to run on a multiprocessor machine with shared memory.

Preface

I declare that the work reported herein and the writing of the thesis is my own work. The design and implementation of the algorithms as described in chapter 6 is my own original work.

Z. ABDELOUAHAB

TABLE OF CONTENTS

1. Introduction

A problem of continuing interest is the determination of the amount of flow that can take place through various kind of networks. This kind of problem is an example of a beautiful subject that has many important applications such as telephone, urban and interurban road networks and electrical transmission lines. Depending upon supply and demand , flow can be diverted along various paths within networks . The fact that all networks whose edges have a physical limitation have a maximum flow , has stimulated much interest in determining this quantity.

The term "network" is frequently used instead of "graph" especially when quantitative characteristics are imparted to the nodes and lines. It consists of a set of vertices V and a set of edges E. It is denoted by N(V,E). It is important to note also that the edges are directed. Vertices or nodes are used to indicate and to represent the objects , while the edges are used to represent the direct connection between objects .

The first algorithm of a maximum flow problem for a network was given L.R Ford and D.R Fulkerson [18] . They used their algorithm not only to solve the problem , but also to prove theorems about flow networks. It turned out that their algorithm depends on the edge capacities in the network as well as the number of vertices and the number of edges. Edmonds and Karp [12] gave their first algorithm for the problem whose speed is bounded by a polynomial function of a number of vertices and a number of edges. In fact, their algorithm runs in time $O(E^2V)$. In 1970, E.A Dinic [11] came with a new algorithm whose speed is $O(EV^2)$. From the figure which is displayed below (see [44]), all the algorithms except the two first follow the Dinic approach. The success of his approach is that he partitioned the network into layers or levels . A layer consists of a set of a vertices which are connected to the other vertices of the next layer by the edges. For example, the first layer contains the first vertex, while the last layer contains the last vertex. From the complexities of the algorithms of Fig.1 , we can see that there is a lot of improvements since the Ford and Fulkerson method , but it is still unknown how close we are to the ultimate algorithm.

Author(s)	Year	Complexity
Ford , Fulkerson	1956	-------------
Edmonds , Karp	1969	$O(E^2 V)$
Dinic	1970	$O(EV^2)$
Karzanov	1973	$O(V^3)$
Cherkassky	1976	$O(E^{1/2} V^2)$
Malhorta , et al	1978	$O(V^3)$
Galil	1978	$O(V^{5/3} E^{2/3})$
Galil , Naamad	1979	$O(EV\log^2 V)$
Sleator , Tarjan	1980	$O(EV\log V)$
Goldberg	1985	$O(EV\log V^2/E)$

Fig. 1

The second chapter of this thesis will contain the fundamental definitions, terms and symbols required to describe and classify the problem . In fact, all major concepts and generalities concerning the maximum flow problem through a network are presented to get a relatively broad idea of the problem . We will look also at different types of networks for which the problem is posed. These are capacitated networks and bounded networks. A capacitated network is the one in which the flow of an edge is bounded by a certain value from above (i.e the flow cannot exceed this value). However, in a bounded network the flow of an edge is contained in an interval (i.e the flow is bounded from above and below). A brief method for getting the maximal flow is presented for capacitated networks. A method for transforming a bounded network to a capacitated one is also described . At the end of the second chapter, among the algorithms which we will describe is the original version of Ford and Fulkerson [18] , this is because of its importance and its simplicity, if not for its speed.

The third chapter is devoted to the total description of two sequential algorithms and gives the full details concerning them. The first algorithm which will be presented is the original method of Edmonds and Karp which chooses a path leading to an increase of flow by the largest possible augmentation. The second algorithm is based also on the Edmonds and Karp approach using the breadth-first search method for traversing a network. A brief description of this method is also given. As an illustration of each algorithm presented, some test cases are added, showing how the algorithms work in order to obtain the maximum flow in any network.

Recently, the concept of parallelism has been used to improve the effectiveness of algorithm design using parallel computers. During this period a wide variety of parallel architecture have been proposed and a fair number have been implemented at least in an experimental form. More precisely, four different organisational classes of computers have been defined by Flynn [17] . These may be classified into :

- SISD : (single instruction stream, single data stream)
- SIMD : (single instruction stream, multiple data stream)
- MISD : (multiple instruction stream, single data stream)
- MIMD : (multiple instruction stream, multiple data stream)

The SIMD and MIMD machines are parallel computers.
The parallel algorithms for a maximum flow problem which we will present are designed for a network of transputers. A network of transputers is an example of an MIMD structure which is a class of multiple processors. Each transputer is considered as an SISD machine with own processor , memory and links. In chapter 4, more details are given about all the above architectures. It is also concerned mainly with the transputer and the Occam language. It illustrates the close association between the language and the device. The language which is designed for concurrent programming and which requires a modularity structure. The last part of the chapter, introduces the description of the network of

transputers used for implementing our algorithms.

So far, there exist good parallel algorithms for problems like finding the maximum of a list of numbers by Shiloach & Vishkin [38], Valiant [43], and merging two ordered sets by Gavril [20] , Shiloach & Vishkin [38], and sorting a set of elements by Hirschberg [22], Preparata [33], Shiloach & Vishkin [38], and for elementary graph problems such as computing connected components by Hirschberg et al [23], Shiloach & Vishkin [39], finding the minimum spanning tree , performing breadth-first search as well as depth first-search by Eckstein [13] on graph and so on. However, there are difficulties in designing a good parallel algorithm for the maximum flow problem , this is because of the complexity structure of the problem and also for the its apparent sequential nature. The most efficient sequential algorithm does not have a straight forward parallel implementation.

The general idea behind parallel computation is that programs using p processors should run p times faster than otherwise identical programs using only one processor, although theory and experience show that the actual speed up is smaller. It is important to note that the main objective of parallelism is to reduce the total cpu time that is used to obtain the solution of the problem. Among the parallel algorithms for a maximum flow problem which are known to us and presented in chapter 5 , is the algorithm of Shiloach and Vishkin [37]. It is considered as one of the few successful attempt to parallelise this kind of problem.

Their algorithm follows the E.A Dinic approach and it is specifically designed for MIMD machine with shared memory. This method is illustrated by an example for a better understanding.

Most of the algorithms for the maximum flow problem have been designed for a serial computer so they might not expose all the parallelism inherent in the problem. A typical simulation approach would divide the system into subsystems, each described by a mathematical model. At any stage within an algorithm, parallelism is defined as the number of steps that are independent and therefore can be performed concurrently . The chapter 6 introduces the parallel implementations of the two algorithms already presented in chapter 3. We will notice how each implementation is divided into processes which the transputers and Occam require. The full details are given in that chapter. A comparison between the two parallel methods is carried out in order to show the best. This is illustrated by some results found when the methods are applied to some examples.

Concluding remarks are contained in chapter 7.

2. Basics and generalities

2.1. Concepts

All networks of our interest are finite, directed, have no loops and no parallel edges .

In a given network N =(V,E) , where

V : Set of vertices or nodes

E : Set of edges.

Each such a network has two distinct vertices , which are the source and the sink. They are denoted by s and t respectively. Most of the material used here is related specifically to flow having a single source and a single sink. The source showed have no incoming edges and the sink showed also have no outgoing edges and although we allow these edges to exist, they are not of interest.

2.1.1. Flow

A flow f in a network is an integer valued function defined on the edges which satisfies the following conditions

(a) - For every edge e, $0 =< f(e) <= c(e)$, where c is a nonnegative integer which is assigned to every edge e.

(b) - Let INF(v), OUTF(v) be the total amount of flow incoming to a vertex v and outgoing from v, respectively. For every vertex v other than the source and the sink ($v \in V - \{s,t\}$)

$$INF(v) = OUTF(v)$$

The second condition is called the conservation rule.

2.1.2. Type of networks

In this thesis we will study two types of networks:

- Capacitated networks
- Bounded networks.

A network is said to be capacitated if for each edge, the flow is bounded from above by an upper bound which is denoted by ub. The upper bound is referring to a capacity (when we mention the upper bound, this means that c(e) = ub(e)). The flow f then satisfies

For $e \in E$, $0 =< f(e) =< ub(e)$.

In a capacitated network, it is assumed that the flow starts from zero, for every edge. In a bounded network, in addition to the upper bound (ub), the flow is also bounded from below by a lower bound denoted by (lb). The bound (lb) is a nonnegative integer . The flow f in this case satisfies the following condition

For each edge $e \in E$, $0=< lb(e) =< f(e) =< ub(e)$.

Note that we can find in a bounded network at least one edge e in which lb(e) <> 0. If not (i.e lb(e) = 0 for every e), the network is capacitated.

The main question which we ask is how to get a maximal flow. A part of the answer is to get first a feasible flow.

2.1.3. Feasibility

A flow f is said to be feasible if and only if

For each edge $e \in E$, $lb(e) <= f(e) <= c(e)$

From a previous definition of a capacitated network there is no problem in calculating a feasible flow. In fact, a starting feasible flow which is equal to zero is sufficient.

Once a feasible flow has been determined, we get an augmenting path from the source to the sink, then increase the flow along the edges of the path. This process is repeated until there is no augmenting path left in the network. This is the general method applied to get a maximum flow in a network.

The problem of finding a feasible flow in the case of a bounded network is more complicated than the capacitated one. Here the difficulty is that the initial feasible flow is not equal to zero since the flow must satisfy the condition

For every edge $e \in E$, $lb(e) <= f(e) <= c(e)$.

Thus our main problem here is whether or not this network possesses a legal (feasible) flow. We shall now describe a method which checks the existence of the flow.

2.1.4. Modification of a network

We modify our original network N = (V,E) into another one, called an auxiliary network. It is denoted by N' = (V',E') and has the following characteristics :

- The vertex set V' consist of the set $V = \{v_1, v_2, v_3, ..., v_n\}$ including two additional vertices v_0, v_{n+1} which are the source and the sink respectively of the new network N'.

- For every edge e = (v,w) , i.e v --> w, such that $v,w \in V$, N' has three corresponding edges :

$e' = (v,w)$

$e'' = (v_0,w)$

$e''' = (v,v_{n+1})$

- The assignment bounds are as follows :

$lb'(e') = lb'(e'') = lb'(e''') = 0$

$c'(e') = c(e) - lb(e)$

$c'(e'') = c'(e''') = lb(e)$

c' , lb' are the capacity and the lower bound of an edge belonging to the auxiliary network N', respectively.

- One final edge is added and has the following characteristics :

$b = (v_n,v_1)$

- We associate with this edge a very high upper bound ($c'(b) = ub'(b) = \infty$) and a lower bound $lb'(b) = 0$

There are two main reasons for transforming a network N into an auxiliary network N' :

- First, N' is a capacitated network and a starting feasible flow is always available with a value zero.

- Second, a maximal flow in N' can be easily transformed into a starting legal flow for the original network N. For more detail see

[8] [14] [15] [18] . In fact, one can show that if the flow f' saturates all the edges emanating from v_0, such a flow (if it exist) is necessarily maximal in N'. Then it will follow that the original network has a legal flow. Clearly, if all the edges which emanate from v_0 are saturated, then so are all the edges which enter v_{n+1}. This follow from the fact that each lb(e), of the original network, contributes its value to the capacity of one edge emanating from v_0 and to the capacity of one edge entering v_{n+1}. Thus the sum of capacities of edges emanating from v_0 is equal to the sum of capacities of edges entering v_{n+1}.

$$f'(e'') = c'(e'') \quad \text{for each edge } e''$$

$$f'(e''') = c'(e''') \quad \text{for each edge } e'''$$

Once the conditions above are satisfied we calculate the initial feasible flow for every edge of the original network N with the expression (f(e) = f'(e') + lb(e)) . The total initial feasible flow emanating from a vertex v_1 can be found in the edge b = (v_n, v_1). It is the amount of flow entering the vertex v_1 through the edge (v_n, v_1) .Once a feasible flow is calculated we return to our original network N to determine a path and apply the general method for obtaining a maximal flow which is described above.

2.2 The Ford and Fulkerson algorithm

As an illustration of the simplest algorithm which solved the problem of a maximum flow, we propose to describe in this section the algorithm of Ford and Fulkerson. Their algorithm is the first which was suggested to find a solution to the problem [14] [15], [18], [44]. It mainly uses a method which is called a " labelling method " and it will be described later.

The main idea of their algorithm is in general:

- start with some feasible flow
- look for an augmenting path
- update the flow

These steps are repeated until there is no path left in the network. The most important step is to find an augmenting path. This path is a simple one from the source to the sink and may contain edges of the direction of the path i.e from the source to the sink (forward edges) and edges of the opposite direction of the path (backward edges).

For a forward edge e , in order to increase the flow through it , the flow must be less then its capacity ($f(e) < c(e)$). On the other hand, if we have a backward edge e in the path, in order to increase the flow through the whole path, we must be able to cancel or reduce some of its flow. Hence, we must have f(e) strictly greater than zero , ($f(e) > 0$).

To find an augmenting path for a given network, a labelling process is used. In this process a vertex v gets a label of the form (u,+/-,z), where u is the vertex from which v is labelled. The vertex u must be labelled before any labelling from u of other vertices take place. Then u becomes a scanned vertex after labelling v.

If e = (u,v) is a forward edge then v gets a label $(u,+,z_v)$, where

$z_v = \min \{ z_u, (c(e) - f(e)) \}$

$f(e) < c(e)$

the sign "+" stands for forward.

If e = (u,v) is a backward edge Then v is labelled by $(u,-,z_v)$

$z_v = \min \{ z_u, f(e) \}$

$f(e) > 0$

"-" stands for backward.

The general algorithm presented below is applied to a capacitated network and may be stated as follows :

Algorithm of Ford and Fulkerson:

```
-----------------------------------------------------------
Begin
  For each edge e, associate f(e) = 0.
  maxflow := 0
  halt := FALSE   ( at the beginning the flow is not maximal )
  Repeat
    Begin
      Initially, every vertex is unlabelled and unscanned.
      Label the source s by ($-\infty$,+,$+\infty$).
      While { there is a labelled and unscanned vertex u and
                sink is unlabelled } Do labelandscan(u) ;
      If sink is unlabelled  Then halt := TRUE
      Else changeflow
    End
  Until halt := TRUE
End
-----------------------------------------------------------
labelandscan(y) ;
Begin
  For every unlabelled vertex v that can be labelled from y by
      either a forward or a backward labelling Do
    If  f(y,v) < c(y,v)  Then   label v by (y,+,$z_V$)
    Else If f(v,y) > 0 Then label v by (y,-,$z_V$) ;
  change the scan.status of y to ' scanned '
End
```

```
Changeflow ;
Begin
  { Starting from the sink and going backward using the labels
  until the source is reached. The amount by which the flow is
  increased is found in the ' z part of the sink'. }
  Begin
    x := sink
    amount := ' z part of the sink '
    Repeat
      Begin
        from label of x
        If sign = ' + ' Then
          increase flow of the edge (previous,x) by amount
        Else
          decrease the flow of the edge (previous,x) by amount
          { previous is the vertex which labelled x.}
        x := previous
      End
    Until x = source
    maxflow := maxflow + amount
End
```

3. Sequential algorithms

3.1. Overview

In this chapter we are going to present two sequential implementations for the algorithm of a maximum flow. The first implementation is the original method of Edmonds and Karp [12] . The main principle is that it finds an augmenting path from the source to the sink , then it increments the flow along those edges. The path found from the source to the sink is not an arbitrary one, but it is chosen in a way it increases the flow by the greatest amount compare to any other in the network. In fact, the search is always performed from the vertex which outputs the highest amount of flow, therefore it leads to a path which increases the flow with the highest value . The second method is based also on the Edmonds and Karp mentioned above but using Breadth-first search for traversing a network to get a path. The search in this method is always done from the vertex which was visited first and trying at the same time to find a path which increases the flow by a high value. The Breadth-first search is considered as the most efficient method for traversing a graph and it is well adapted for parallelism [2] [13]. The description of these two methods will be followed by examples for a better understanding.

3.2. Data structure of the sequential algorithm

The following data structure is the backbone of the sequential algorithm which will be described later.

queue : implemented as an array used for keeping the vertices or nodes in. In general this queue represents a waiting list . The insertion is from the end of the queue , while the removal is from the beginning. It is used in the second method which uses breadth first search.

val : is a linear array where each vertex of the network has an entry. Each entry will contain the maximum output of flow from that vertex , or we can say also that it will contain the maximum input of flow to that vertex, this comes from the fact that the total input of flow to a vertex is equal to the total output flow from a vertex.

visited : an array identical to val. Each entry will be represented by one of the numbers 0,1 or 2.

- 0 stands for unvisited vertex
- 1 stands for vertices that have been visited but are still waiting for the search to be performed from them.
- 2 stands for vertices that have been visited and the search has been performed from them.

dad : this also is an array of the same size as the preceding. It will help us to store the path. Each element of the array is used to store its father. Suppose we have an edge i ---> j , and j has been visited from i then in the j^{th} entry of dad will contain i.

sizes : the matrix for upper and lower bounds. In general it is used for representing a network. Each edge (i,j) is represented by two elements of the matrix. The first one is (i,j), this contains the upper bound or capacity, while the second (j,i) contains the lower bound but it is represented with a negative sign for differentiating between the two (i.e " - (value of lower bound) "). The auxiliary network is also represented by the same matrix **sizes** but with some modifications which will be described later.

flow : the flow of all the edges of the network is represented by a matrix of the same size than the matrix **sizes**.

q,p : the source and the sink of the network respectively. In our algorithm , they are represented by the 1st and the nth vertices for the original network N and by the (n+1)th and the (n+2)th vertices for the auxiliary network N'.

3.3. General implementation of the maximum flow algorithm

In this subsection , we will present a general algorithm for obtaining a maximum flow in any network. This algorithm shows in general the steps which lead to a maximum flow in a network and it is valid for both methods which will be described later one by one. It is described in the following page

```
- Initialisation of the corresponding tables, variables
- **If** network is bounded **Then Begin**
- Proceed to the modification of the original network into an
  auxiliary one ( The new one should be capacitated).
- Find the maximal flow for the new network. This leads to
  an initial feasible flow for the original network. It is found
  in the edge (n,1) ( i.e the value of flow(n,1) in the matrix
  flow )
- Update the matrix sizes to its original form ;
- Go back to the original network, after finding the starting
  feasible flow for N. The flow of the edges can be calculated
  using the expression flow(e) = flow(e') + lb(e) ( i.e the flow
  of an edge e in N is equal to a flow of the same edge e' in N'
  plus its lower bound) ;
  **End;**
- find the maximal flow for the original network N.
```

3.4. Algorithm for transforming the network into an auxiliary network

For transforming the network into an auxiliary one , we need to modify the matrix sizes of the upper and lower bounds. The modified matrix has two additional rows and two additional columns. The new source is represented by $(n+1)^{th}$ vertex and the sink by $(n+2)^{th}$ vertex.

```
For i = 1 to n Do
    For j = 1 to n Do
        If (sizes[i,j] > 0) and (sizes[j,i] <> 0) Then Begin
            { there exist an edge (i,j) and lb(i,j) > 0 }
            sizes[i,j] := sizes[i,j] + sizes[j,i]
                { this mean that c'(i,j) := c(i,j) - lb(i,j), c'(i,j) is the capacity
                of (i,j) in N' and sizes[j,i] < 0 ( lower bound of (i,j) )}
            sizes[n+1,j] := sizes[n+1,j] - sizes[j,i]
                { adding the edge (n+1,j) to N' with the corresponding
                upper bound sizes[n+1,j] . If there are redundant edges,
                they are compacted all together }
            sizes[i,p] := sizes[i,p] - sizes[j,i]
                { adding the edge (i,p) (i.e from i to the new sink; p:=n+2).
                Redundant edges are also compacted };
        End;
sizes[n,1] := k1
    { k1 is an integer with a very high value, It represents the
    upper bound of the added edge (n,1)}.
```

Note that when the matrix sizes is put back to its original form, the additional two rows, two columns and the edge (n,1) are eliminated. The upper bounds c of the edges in N are updated as follows:

```
    For each edge e in N
        c(e) := c'(e) + lb(e)  { e is also an edge in N' }.
```

3.5 First method

3.5.1 ALgorithm

```
Initialisation of the vectors dad, val, visited to zero.
    {q := 1; p := n (original network) and q := n+1; p := n+2 (for the
    auxiliary network) }
val[0] := 0
max := q { The search is performed from max }
While ( max <> 0 ) Do
    i := max
    max := 0
    visited[i] := 2
    If i = q Then val[i] := maxint { output of the source is maxint }
    For j := 1 to p Do
      If visited[j] <> 2 Then Begin
          If ( there is an edge from i to j ) Then Begin
            If sizes[i,j] > 0 Then
                pri := sizes[i,j] - flow[i,j]
            {the edge (i,j) is forward. pri is the residual capacity}
                Else If ( the network is an auxiliary)
                    Then pri := flow[j,i]
                    { The edge is backward ( sizes[i,j] = 0 ), the
                    returning flow is equal to the flow value of
                    the edge .}
                    Else pri := flow[j,i] + sizes[i,j] ;
                    { the network is the original; (sizes[i,j] <= 0); the
                returning flow is pri := flow[j,i] - lb(j,i) }
          If val[i] < pri Then pri := val[i] ;
             { the output of i is the maximum amount that can
             reach j}
          If val[j] < pri Then
             val[j] := pri
             dad[j] := i
             { the amount of flow that reached j is the highest
             compare to any other that has reached j until now.
             Update of the corresponding values in dad and
             val};
       End ;
    If val[j] > val[max] Then max := j;
       { testing if this vertex j possesses the highest output of
       flow so that its value is recorded in max and the next
       search should resume from max };
    End .
```

The algorithm described above is the original Edmonds and Karp algorithm to find one path from the source q to the sink p. To find all the paths, the algorithm above is repeated many times until there is no path left in the network. This happens when the sink cannot be reached (i.e val[p] = 0). Once the execution of the above algorithm has terminated , the retrieval of a path is done as follows:

```
y := p
x := dad[p]
While (x <> 0) Do
   If edge (x,y) is forward  Then flow[x,y] := flow[x,y] + val[p]
                             Else flow[y,x] := flow[y,x] - val[p]
   y := x
   x := dad[y]
```

Starting from the sink p and going backward, the edges are found one by one and the flow of the path is increased by val[p]. val[p] contains the value by which the flow of the path should be increased. This value is the highest one. The value of the maximal flow is updated in the following manner:

```
    If network is not an auxiliary
        flowmax := flowmax + val[p]
```

Initially, the value flowmax of the network N is equal to zero. In the case of a bounded network , when the maximum flow is found in the auxiliary network, the value of maxflow is put to flow[n,1] which is the value of the initial feasible flow of the original network N. The whole algorithm is illustrated in the next section by some examples.

3.5.2 Test cases

Here we show two examples of networks. The first one is a capacitated network and the second is a bounded one. We will illustrate in general how the algorithm works:

Example1

The first example is a capacitated network (see fig.3.1). The matrix given below is the matrix sizes which represents a network of the fig.3.1

$$\begin{matrix}
0 & 8 & 7 & 6 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 7 & 0 & 0 \\
0 & 0 & 0 & 0 & 2 & 4 & 0 \\
0 & 0 & 0 & 0 & 0 & 4 & 2 \\
0 & 0 & 0 & 0 & 0 & 0 & 6 \\
0 & 0 & 0 & 0 & 0 & 0 & 5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{matrix}$$

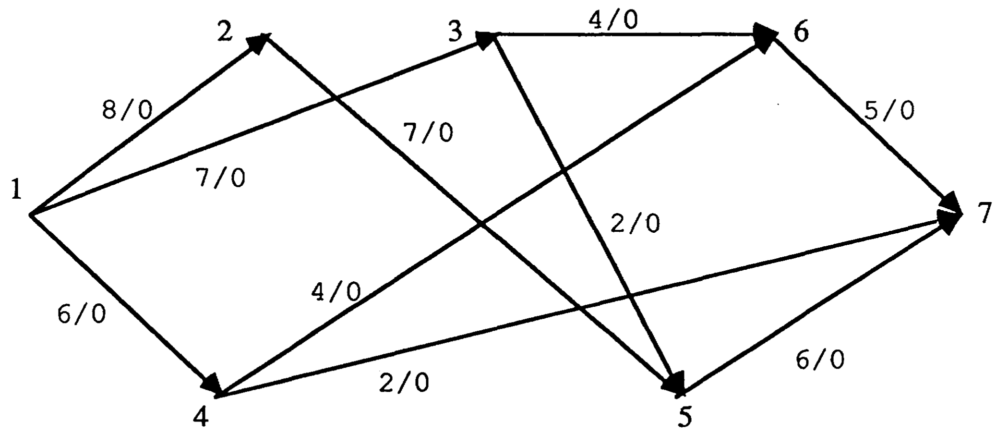

Fig.3.1

The lower bounds are all zeros in this case. So the starting feasible flow in the network is obviously zero for each edge of the network. Applying the algorithm for finding the maximal flow, the paths which have been found are in the following order:

[1 2 5 7] which increases the flow by val[7] = 6

[1 3 6 7] which increases the flow by val[7] = 4

[1 4 7] which increases the flow by val[7] = 2

[1 4 6 7] which increases the flow by val[7] = 1

The paths found are in decreasing order. The first path increases the flow by the highest value while the last one increases the flow by the lowest value. The value of flowmax is the sum of all values of the paths found above.

flowmax = 13

Example2:

Case of a bounded network (see fig.3.2)

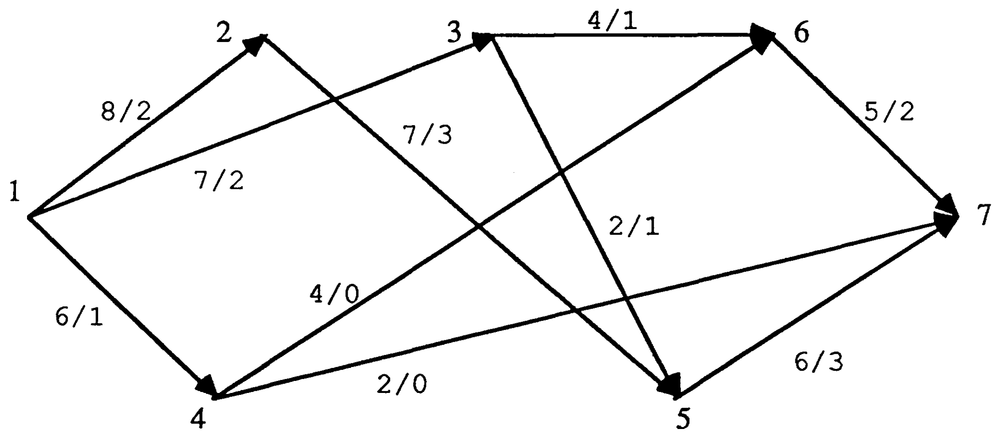

Fig.3.2

The second network is a bounded one. It is the same network as the precedent but with lower bounds of the edges which are not all equal to zero . The first thing to do is to apply the algorithm for transforming the network into an auxiliary one. The resulting network is represented in fig.3.3 which shows all the edges which have been added. After compacting all the redundant edges, the final representation is in fig3.4

The matrix sizes representing the original network N of fig.3.2

$$\begin{array}{ccccccc}
0 & 8 & 7 & 6 & 0 & 0 & 0 \\
-2 & 0 & 0 & 0 & 7 & 0 & 0 \\
-2 & 0 & 0 & 0 & 2 & 4 & 0 \\
-1 & 0 & 0 & 0 & 0 & 4 & 2 \\
0 & -3 & -1 & 0 & 0 & 0 & 6 \\
0 & 0 & -1 & 0 & 0 & 0 & 5 \\
0 & 0 & 0 & 0 & -3 & -2 & 0
\end{array}$$

The upper bounds are denoted by a positive sign, while the lower bounds are denoted by a negative sign.

The resulting matrix sizes after applying the algorithm for transforming a network described previously is as follows :

$$\begin{array}{ccccccccc}
0 & 6 & 5 & 5 & 0 & 0 & 0 & 0 & 5 \\
-2 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 3 \\
-2 & 0 & 0 & 0 & 1 & 3 & 0 & 0 & 2 \\
-1 & 0 & 0 & 0 & 0 & 4 & 2 & 0 & 0 \\
0 & -3 & -1 & 0 & 0 & 0 & 3 & 0 & 3 \\
0 & 0 & -1 & 0 & 0 & 0 & 3 & 0 & 2 \\
k1 & 0 & 0 & 0 & -3 & -2 & 0 & 0 & 0 \\
0 & 2 & 2 & 1 & 4 & 1 & 5 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

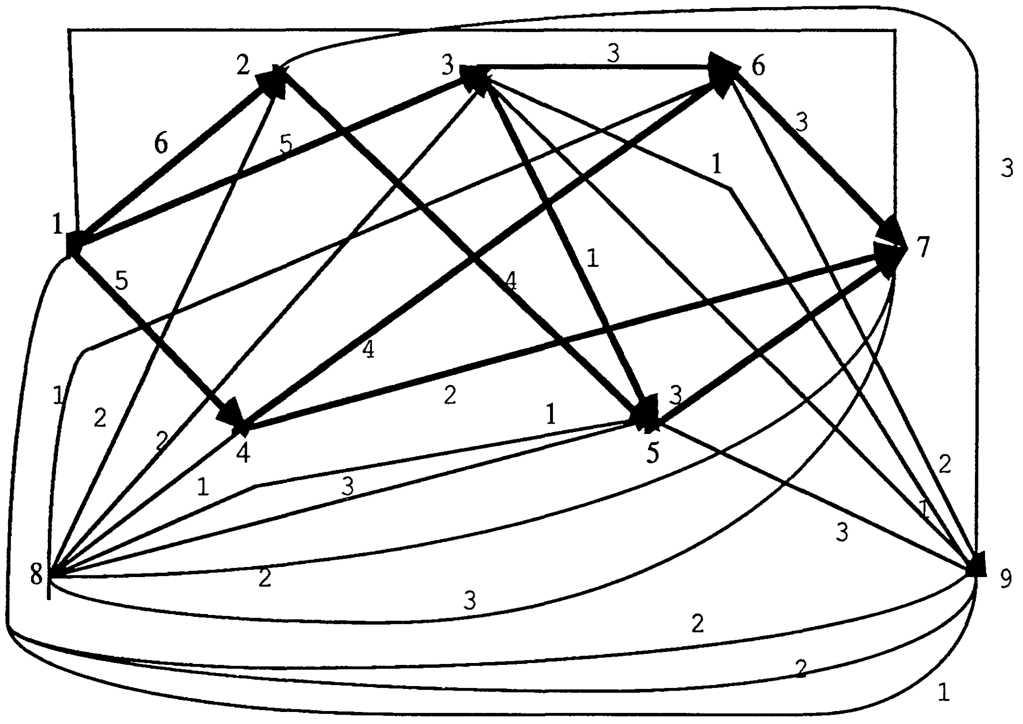

Fig.3.3

Note that the resulting auxiliary network is a capacitated one and all the lower bounds are equal to zero. Therefore, the bounds represented in the augmented matrix sizes with a negative sign are not the lower bounds of the network N' but they are lower bounds of the network N.

Initially, the elements of the matrix flow are all equal to zero. Applying the algorithm for obtaining the maximum flow in N' , the results obtained are as follows:

[8 7 1 9] which increases the flow by val[9] = 5

[8 5 9] which increases the flow by val[9] = 3

[8 2 9] which increases the flow by val[9] = 2

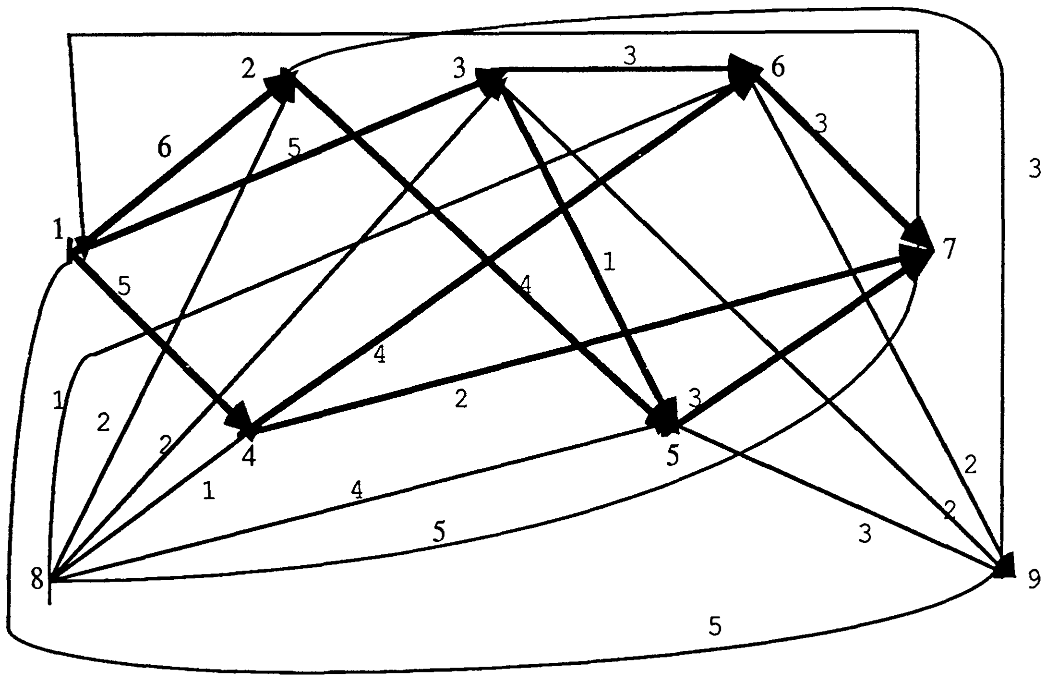

Fig.3.4

[8 3 9] which increases the flow by val[9] = 2

[8 6 9] which increases the flow by val[9] = 1

[8 4 6 9] which increases the flow by val[9] = 1

[8 5 7 1 2 9] which increases the flow by val[9] = 1.

the resulting matrix flow after finding the maximal flow in N' is given below:

$$\begin{matrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 3 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 2 & 2 & 1 & 4 & 1 & 5 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{matrix}$$

The total initial feasible flow is found in the edge (7,1) = 6

Now we give the starting flow for each edge in N using the matrix flow found above. The flow in each edge may be calculated using the expression flow(e) := flow(e')+lb(e), where e is an edge in N and e' is the same edge in N'.

$$\begin{matrix} 0 & 3 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

The value of flowmax is initialised to 6 (which the value of the initial feasible flow) .Using the matrix flow above we calculate the maximum flow for the original network and the paths obtained are :

[1 3 6 7] which increases the flow by val[7] = 3

[1 4 7] which increases the flow by val[7] = 2

[1 2 5 7] which increases the flow by val[7] = 2

It is shown from the example that all the paths which have been found are in decreasing order.

The flowmax value is equal to the initial value plus the values of the paths found in this part.

flowmax = 13.

3.6. Second method

The second method which is presented in this section is based on the previous one, but using the breadth-first search for traversing a network. Using this method, the search from vertices is done in first in first out. In fact, it does not need to search each time for the vertex from which the search should be performed like in the first one and the breadth-first search is well adapted for parallelism [2] [13].

3.6.1. General method of breadth-first-search

This classic traversal method [3] [6] [36] uses and maintains a queue Q as a waiting list and works as follows :

```
Procedure BFS( v : vertex)
    { v is the vertex where the search should start }
    { Q is empty at the beginning and all the vertices are unvisited}
    Put v in Q
    Mark v "visited"
    While Q is not empty Do
       Remove a vertex u which is first in Q
       For each vertex w adjacent to u Do
         If w is unvisited Then put w in Q
                                mark w visited.
{ End of BFS } :
```

3.6.2. Algorithm

```
Initialisation of val dad,visited to zero.
  { (q := 1 ; p := n) for the original network and (q := n+1 ; p :=
  n+2) for the auxiliary network }
Put the first vertex into "queue". Usually it is the source q from
which the search starts . { val[q] := maxint }.
While ( queue is not empty ) Do
  Remove the first vertex from the queue which we denote by u
  For each vertex v adjacent to u Do Begin
    If sizes[u,v] > 0 Then pri := sizes[u,v] - flow[u,v]
      { the edge is Forward , pri is the residual capacity}
      Else If (network is an auxiliary)
          Then pri := flow[v,u]
          Else pri := flow[v,u] + sizes[u,v] ;
      { If the network is an auxiliary ( lb(e) = 0 ) then the
      returning flow is equal to flow(v,u). However, if the
      network is the original  then pri := flow(v,u)-lb(v,u) }
    If v is "not visited" Then
      visited[v] := 1  { v is put "visited" }
      insert v in the queue ;
    If val[u] < pri Then pri := val[u]
      { The output of u is the maximum which can reach v} ;
    If val[v] < pri Then
      val[v] := pri
      dad[v] := u ;
    { This mean the flow which is outputted from u to reach v
    is bigger than any other flow outputted by other vertices
    to v  till now. The vectors val and dad are updated. } ;
End
```

The procedure above is used to get a path from the source q to the sink p and it is repeated until there is no path left in the network (case where val[p] = 0)

Procedure for retrieving a path and update of flow

Once a path is found, it is retrieved by going backward from the sink to the source, and the value by which the flow is increased is also calculated from the residual capacities of the path edges . The amount by which the flow is increased is denoted by min. This value is not necessarily equal to val[p]. If we consider that min is equal to val[p] then it may result in that no edge of the path will be saturated . Therefore, this path will be found in the next search (redundancy). The procedure is presented in the next page.

Note that the paths which are found by this method are not necessarily in decreasing order and there may not be all the same.

The updating of the value of flowmax is done in the same manner as in the first method but instead of adding val[p] to flowmax , the value min is added to flowmax.

```
min := maxint    { Initially the flow value of the path is maxint }
y := p {  starting from the sink }
x := dad[p]  { going backward }
While ( x <> 0 ) Do
   If edge (x,y) is forward  Then residual := sizes[x,y] - flow[x,y]
         Else If (network = auxiliary) Then residual := flow[y,x]
            Else residual := flow[y,x] + sizes[x,y]   { sizes[x,y] < 0 }
            {(x,y) is backward;  sizes[x,y] := - lb(y,x) }
   If min < residual  Then min := residual
   y := x
   x := dad[y]     { find another vertex of the path }
If dad[p] <> 0 Then { The path exist}
   for each edge e of the path
       flow(e) := flow(e) + min ; { update of matrix flow }
```

3.6.3. Test cases

Applying the method which uses breadth-first search to the same examples presented in the test cases section of the first method, the results obtained are exactly the same. All the paths which are found are in the same order as well. We will present now another example where the two methods give different results. The network is a capacitated one and it is shown in fig.3.5 which is presented below.

Applying the first method to this network the results which are obtained are as follows

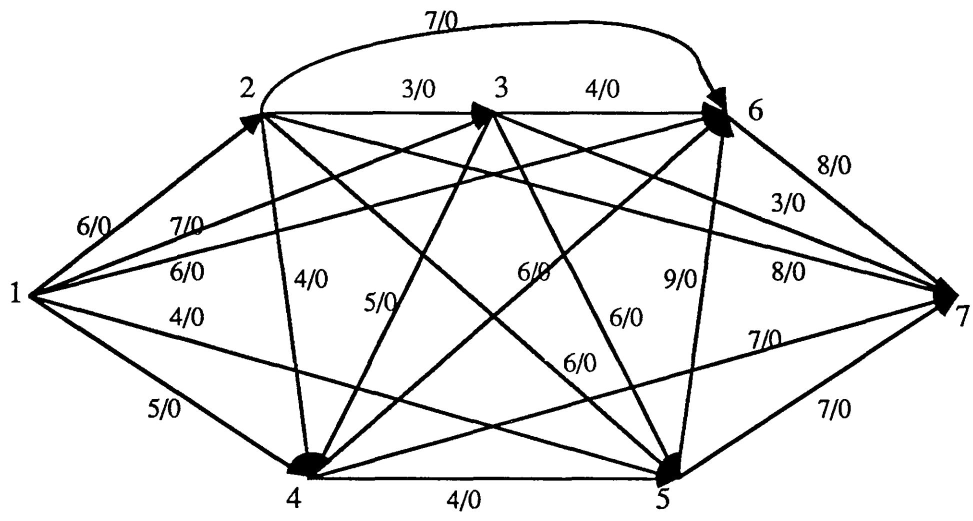

Fig.3.5

Matrix sizes of a network in Fig.3.5

0	6	7	5	4	6	0
0	0	3	4	6	7	8
0	0	0	5	6	4	3
0	0	0	0	4	6	7
0	0	0	0	0	9	7
0	0	0	0	0	0	8
0	0	0	0	0	0	0

The paths which are obtained are in the following order:

[1 2 7] which increases the flow by val[7] = 6

[1 3 5 7] which increases the flow by val[7] = 6

[1 6 7] which increases the flow by val[7] = 6

[1 4 7] which increases the flow by val[7] = 5

[1 5 3 7] which increases the flow by val[7] = 3

[1 3 4 7] which increases the flow by val[7] = 1

[1 5 7] which increases the flow by val[7] = 1

The flowmax value obtained is equal to 28.

Applying the second method to the same network, the results are :

[1 2 7] which increases the flow by min = 6

[1 3 5 7] which increases the flow by min = 6

[1 6 7] which increases the flow by min = 6

[1 4 7] which increases the flow by min = 5

[1 5 6 7] which increases the flow by min = 2

[1 5 3 7] which increases the flow by min = 2

[1 3 7] which increases the flow by min = 1

We notice from these results that some paths which are found are different from the others found by the first method and with different values as well. There are many other examples when the second method is applied, the values by which the flow is increased are in different order (not necessarily in decreasing order) but they are not presented here.

4. Parallel Computers and parallelism

4.1. Concept

The availability of cheap processors and the introduction of VLSI technology has made a lot of improvements in the development of parallel computers. These computers now consists of many processors interconnected in some ways. Many computational tasks can be divided into subtasks which need not be executed sequentially. Parallel computers provide the opportunity for the distribution of computations and , therefore exploiting the parallelism. Parallel tasks may be defined in general as tasks which can be executed all together simultaneously.

4.2. Computational model

One of the earliest division of different computers was due to Flynn [17] . He has effectively classified the different computers into four categories . They are SISD, MISD, SIMD, MIMD. The two last categories are parallel computers which have been built.

SISD : (Single Instruction stream, Single Data stream) Is in general the conventional serial von Neuman machine in which there is only one instruction processing unit. Each arithmetic instruction initiates one arithmetic operation, leading to a single data stream.

MISD : (Multiple Instruction stream, Single Data stream) This machine includes a specialised streaming organisation using multiple instruction streams on a single sequence of data stream.

As far as we know there is no MISD machine which has been built, because this architecture has not received any attention.

SIMD : (Single Instruction stream, Multiple Data stream) This sort of machine may have an array of processors working in lockstep under a common control. In fact, a central control unit broadcasts an instruction to be executed by all the processors. SIMD machines are restricted in a way that each processor must perform the same instruction simultaneously. In general, it is easy to instruct a lot of processors to do the same thing than to instruct them to do different things. For this reason this kind of machine usually is constructed from a lot of processing elements. A typical machine which has been widely used is the ICL DAP (Distributed Array Processors), now being built and marketed by Active Memory Technologies. It consists of 4096 parallel processors and they are arranged in a grid of 64*64. There is also other machines like Goodyear MPP and CLIP.

MIMD : (multiple Instruction stream, Multiple Data stream) This category is a class of a multiprocessors, where all the processors execute a number of independent instruction streams on a separate data streams. These types of systems become possible at present , this is due to the availability of a complete processor at a low cost. The processors are linked together to facilitate communication. There are machines in which all the processors can access memories through a switch, these are generally called MIMD shared memory machines. Examples of such machine are the BBN butterfly and Cmmp. The BBN butterfly has been built by Bolt, Baranek and Newman while the Cmmp has been built by Carnegie-Mellon University. There is also another type of machine where the processors have independent memories and the processors communicate only by messages passing like the Caltech hypercube and IBM LCAP (Loosely Coupled Array Processors). A network of transputers falls in this category, where a single transputer is an SISD machine.

4.3. Transputers and Occam

4.3.1. Transputer concept

A transputer is a fully programmable component which is designed for the implementation of concurrent systems. VLSI technology offers a high level of integration and in order to exploit this opportunity, it is necessary to built concurrent systems. These systems are composed from a number of the same devices which are interconnected in some ways to form concurrent systems. The name transputer come from transistor and a computer. A single transputer is considered as a computer which is built on a chip. It contains its own processor, local memory and links for interconnection. A system of interconnected transputers form a multicomputer.

4.3.2. Relation between Occam and transputer

The transputer is totally programmed in Occam which is considered to be the native language of this component. This language is specifically designed to facilitate communication and also for implementing concurrent systems. The transputer can also be programmed in most high level languages which exist, such as Pascal, C, Fortran. In a situation where it is possible to exploit concurrency, but still use standard languages, Occam can be used to link modules written in the selected language. For exploiting the maximum benefit of the transputer architecture, the whole system should be written and programmed in Occam from the system configuration, down to low level i/o and real time interrupts. The implementation of a system in Occam provides all the advantages of a high level language as well as the possibilities to use the features of the transputer.

4.3.3. Transputer architecture

A transputer implements the process model of computation. A process may be defined as an independent computation, with its own program and data, and which can communicate with other processes.

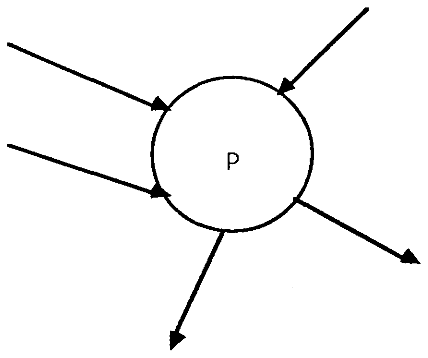

Fig 4.1 A process

A process may also be considered as a black box that can perform an action. A process in general receives inputs and sends outputs. The processes communicate using channels by passing messages (see Fig. 4.1).

In addition, processes can be connected together to form a more complex concurrent systems. The processes {P1,P2,P3,P4,P5} are all connected together forming a more complex structure. These processes communicate through channels which are represented with arrows in Fig4.2.

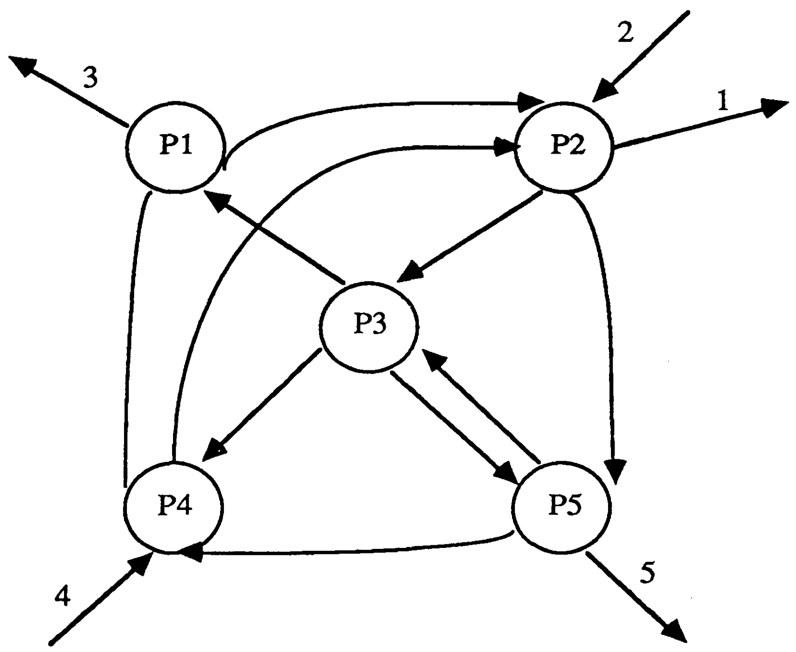

Fig 4.2 Example of interconnected processes

Fig4.3 shows that a collection of processes may be regarded as a process. This process, itself can be interconnected to other processes . Then generally, a process can have an internal concurrency. The processes can only communicate using the channels. The process P of Fig.4.3 replaces all the interconnected processes {P1, P2, P3, P4, P5} of Fig.4.2.

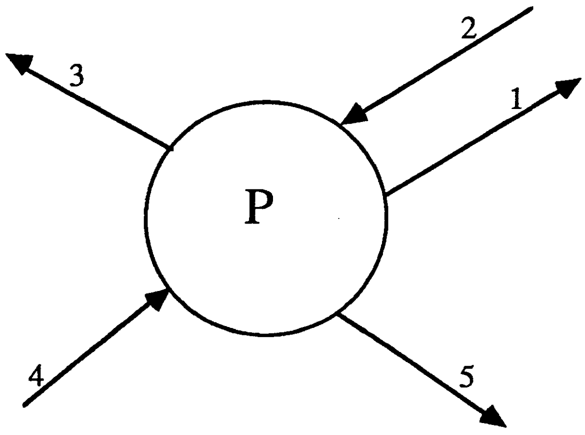

Fig 4.3 Internal concurrency in a process

The description of the architecture has a hierarchic structure which shows how a system of interconnected transputers is designed and programmed in Occam . In fact, Occam enables more complex and powerful systems to be designed by connecting many transputers together. More generally, a transputer system consists of a number of connected transputers , where each of them executes an Occam process. Therefore, the Occam programming model is supported internally.

4.3.4. System hardware

A. Communication and links

The transputers communicate with each other using point-to-point communication links. Each member of the transputer family has one or more standard links. Each link is bidirectional and provides two Occam channels, one in each direction. One of them is used for inputting and the other for outputting. This will allow an Occam program to be mapped onto an appropriate network of transputers (see Fig.4.4). The links are necessary to connect the transputers to build networks of different sizes and topologies (Fig 4.5). Therefore, there is no problem in communication when many transputers are interconnected. For each link and for both directions synchronisation is provided. A communication link is wordlength independent. Therefore, transputers of different wordlength may be interconnected and programmed as single system.

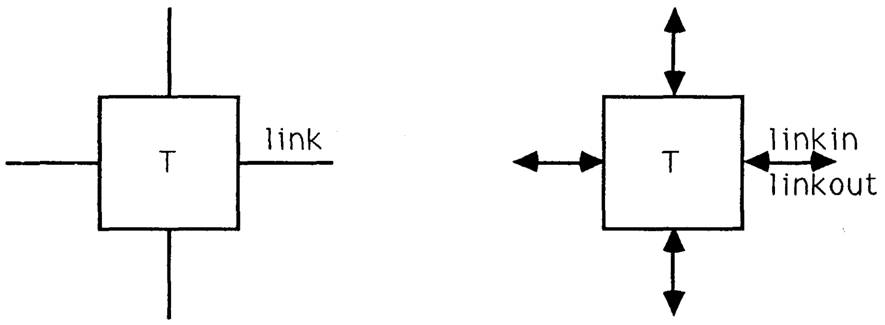

Fig 4.4 Representation of a transputer

B. Memory

Since all the memory is local, the memory grows in proportion to the number of transputers (more transputers implies more memory)

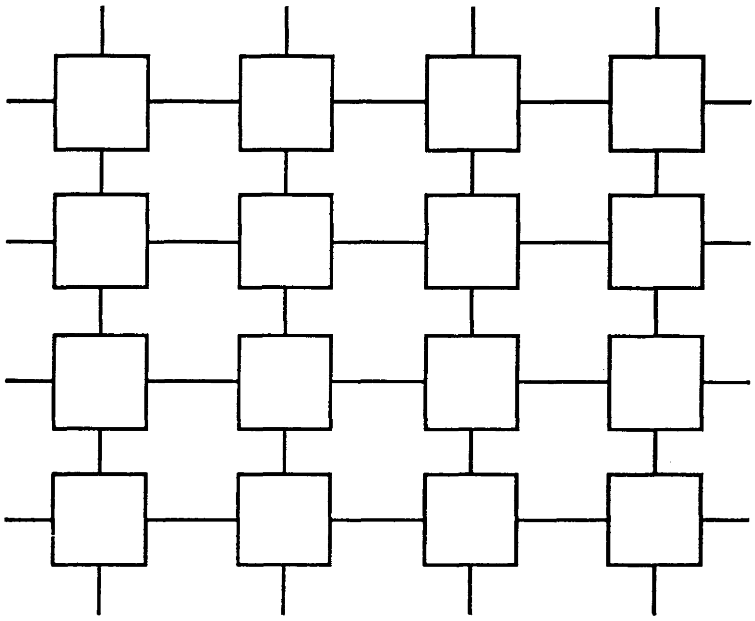

Fig 4.5 Example of 16 interconnected transputer in an array of (4*4)

4.3.5. Basics in Occam

Occam enables a system to be described as a collection of concurrent processes which communicate with each other and peripheral devices through channels. An Occam channel does not depend on a particular hardware implementation because it just describes communication in the abstract. Thus, an Occam program which uses channels may be written and tested without describing where particular processes will be executed.

An Occam process is constructed from a small number of building blocks called primitive processes which we will now describe:

- Assignment : computes the value of an expression and changes the value of a variable, and it is done in the same way as in most other programming languages.
- Input : a process gets a value from a channel.
- Output : a process puts a value to a channel.

Processes can be combined to form sequential, parallel or alternative constructs. These are described as

Sequential : the processes are executed one after another.

Parallel : the processes are executed together at the same time (simultaneously).

Alternative : the component process which is ready first to communicate is executed.

A construct is itself a process and may be used as a component of another construct. Concurrent processes can be expressed with channels, inputs and outputs which are combined in parallel and alternative constructs. Two important properties are described below :

- The first one, concerns a channel, is that it provides a one way connection between two concurrent processes. Communication is synchronised. If a channel is used for input in one process, and output in another, communication occurs when both processes are ready. The inputting and outputting processes then continue, and the value to be output is copied from the outputting process to the inputting process.

- The second property is that an Occam program is the same whether it involves communication between processes on different transputers or on a single transputer. More generally, a program intended for a network of transputers, may be compiled and executed on a single transputer, which shares its time between the concurrent processes. A process which is waiting for communication does not consume any processor time.

4.3.6. Configuration

Occam programs may be designed, written, tested and debugged on a single processor (transputer), and then transferred on a network of transputers. Configuration associates specific processes with real processors and specific Occam channels with real hardware links. More generally, it is what happens at the topmost level of an Occam program in order to determine how the program is loaded on particular hardware. It does not affect the behaviour of a program. However, it does enable the program to ensure a better performance.

4.4. Network of transputers used

4.4.1. Links

Each transputer of the network used has four links. Each link is bidirectional (one way for each direction) . See fig.4.4.

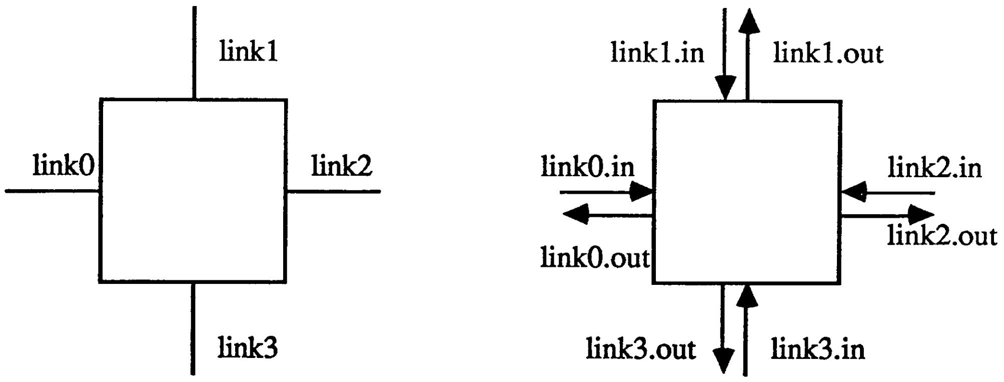

Fig.4.4 Links in a transputer

4.4.2. Network and connections

The network of transputers used is composed of two different boards. They are : IMS B003 and IMS B004 (fig.4.5).

- The IMS B003 is composed of four identical transputers. They are interconnected together using hard wires forming a square. Each transputer has a memory of 256 KBytes RAM. The links provided allow the user to extend the array of transputers by connecting other boards . The links (link1, link0) of each of the couples (T0, T1), (T1, T2), (T2, T3) are added by switching them with wires.

- The IMS B004 consists of a single transputer IMS T414 , 32bit transputer with 2 MBytes memory RAM. The B004 is added inside the IBM PC XT which provides the access to the terminal (Keyboard, Screen) and the filing system. It provides standard links to allow the use of a multitransputer systems. It is considered as the one of the family of compatible evaluation boards. The link1 is switched to link0 of T0 and link2 is switched to link1 of T3. The link0 is connected to the PC server for interfacing as was mentioned before. The transputer development system (the integrated environment which is developed for supporting the programming of transputer network in Occam and which consists of the editor, file manager, compiler, and debugger) runs on this board. An Occam program, which is designed and debugged within a TDS , is configured for either a B004 or a network as a whole. The resulting code is then downloaded to the corresponding system where it is executed.

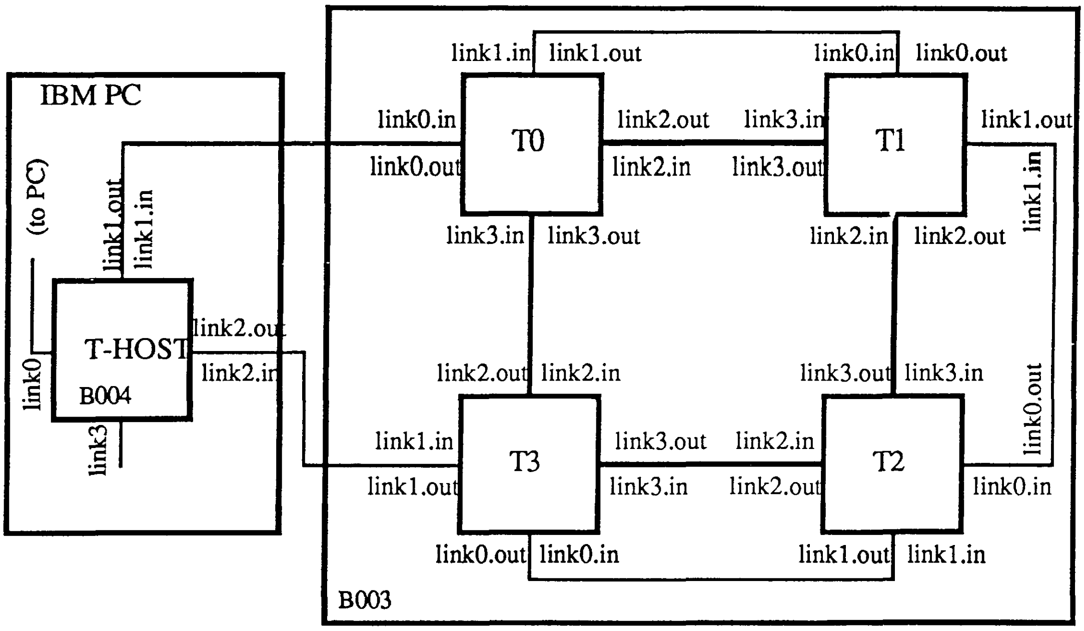

Fig 4.5 Transputer network presentation

5. Previous Parallel algorithm

5.1. High level description

In this chapter, we will describe a synchronised parallel algorithm for the problem of a maximum flow in a directed network. This algorithm is implemented by Shiloach and Vishkin [37]. As far as we know, it is one of the few attempts to parallelise this kind of problem. This is due to the purely sequential nature of the problem. Their algorithm has in fact a parallel implementation, but it is quite difficult to conceive and analyse. The model used in their algorithm is a synchronised parallel computation in which all the processors have access to the common memory (shared memory). Simultaneous reading from the same memory location is allowed and also simultaneous writing is allowed, provided that the processors try to write the same thing. In fact, this algorithm is designed for MIMD machines with shared memory.

The Shiloach and Vishkin algorithm, is in general following the E.A Dinic method [11] in transforming the network into layered networks. The technique of layering has the effect of replacing a single maximum flow problem by several problems, each a good deal, easier than the original. More precisely, for a network with n vertices, the maximum flow is found by solving at most n slightly different problems, each one is called a layered network. Below we will show how a layered network is constructed.

We start from a source s which is contained in the 0^{th} layer. The first layer is constructed from every vertex v such that there is an edge from s to v. In the same way the ith layer is produced from the $(i-1)^{th}$ layer by connecting the vertices of the $(i-1)^{th}$ layer to the next layer with edges. These can be forward or backward. Note that there is no vertex which connects another vertex in the same layer. Obviously, the last layer will only contain one vertex which is the sink. The bounds that are associated with the edges of the layered network may be stated as follows:

Suppose $u \in$ (i-1)th layer and $v \in$ ith layer then

- if $e = u \dashrightarrow v$ (forward edge) and $f(e) < c(e)$ then the new capacity $c'(e) := c(e) - f(e)$
- if $e = u \dashleftarrow v$ (backward edge) and $f(u,v) > 0$ then the new capacity $c'(e) := f(u,v)$

Note that this apply only for a capacitated network.

An efficient method applied for getting the layered network is the breadth-first-search. In fact, a search starts from the source revealing the first layer. In the same way, the ith layer is revealed by performing a search from the (i-1)th layer. Naturally, the BFS applied is performed in parallel.

5.2. Description of the sequential method

For a better understanding of the parallel implementation of the Shiloach and Vishkin algorithm of a max-flow, we shall describe the sequential method because it simulates the parallel one.

In general, the algorithm is divided into pulses. In one pulse, the flow is pushed from one vertex forward as much as possible. For example in the first pulse, s pushes the flow and saturates all the edges which emanate from it. In the succeeding pulses, there will be vertices for which INF(v) = OUTF(v) (incoming flow to v is equal to outgoing flow from v), these kind of vertices are called balanced vertices, and vertices for which INF(v) > OUTF(v). The later are called unbalanced vertices and they always try to push the flow forward. If the flow cannot be eliminated, it is returned backward. Before starting the description of the algorithm, we will introduce some notions used in both the sequential and the parallel implementations.

EXCESS(v) : is the amount of flow that should be pushed forward or returned backward to make the vertex v balanced. This amount is calculated from the expression

$$\text{EXCESS}(v) := \text{INF}(v) - \text{OUTF}(v).$$

AVAILABLE(v) : contains all the edges which emanate from v through which the flow can still be pushed forward.

FLOW QUANTUM Q(e,q) : the flow quantum q is the flow that is pushed through e at a given pulse.

STACK(v) : the stack is used to keep the flow quantums of the edges which enter v. It has the form of (e = u --> v, q). The stack is very useful since the returning of flow from v is done in a last in first out (LIFO) and the stack has this property.

The sequential algorithm is described in two routines. The first one is PUSH(v,EXCESS(v)) implemented for pushing the flow forward, while the second one is used for returning the flow backward RETURN(v,EXCESS(v)). We will see that PUSH and RETURN have a parallel implementation.

```
PUSH(v,EXCESS(v)):
    While ( EXCESS(v) > 0 ) and ( AVAILABLE(v) ≠ ∅ ) Do
    Begin
        e := (v,w)        { the first edge of AVAILABLE(v) }
        q := min( c(e)-f(e) , EXCESS(v) )      { c(e)-f(e) is the residual
            capacity; q is the flow that is going to be pushed
            through the edge}
        Add Q = (e,q) to STACK(w)
        f(e) := f(e) + q      { increment the flow of the edge e }
        EXCESS(v) := EXCESS(v) - q  { reduce the EXCESS(v) by q }
        EXCESS(w) := EXCESS(w) + q  { increment the EXCESS(w) }
        If f(e) = c(e) Then delete e from AVAILABLE(v)  {the edge
            e is saturated then it is removed from AVAILABLE(v) } ;
    End ;
    If AVAILABLE (v) = ∅  { no remaining edges available from v}
        Then the vertex v becomes blocked, and for all u --> v ∈ E,
        eliminate u --> v from AVAILABLE(u)  { u leads to v which
        is blocked } ;
```

```
RETURN(v,EXCESS(v)): { this routine has to reduce the flow in
some edges to balance the vertex v }
    While ( EXCESS(v) > 0 ) Do
    Begin
        Suppose Q = (e = u --> v, q);  e is the first edge in STACK(v)
        q' := min ( q , EXCESS(v))  { returning the flow by q' }
        f(e) := f(e) - q'
        EXCESS(v) := EXCESS(v) - q'
        EXCESS(u) := EXCESS(u) + q'
        If q = q' Then delete Q from STACK(v)  { here we delete Q
            from STACK(v) because all the flow is returned
            backward through the edge e }
            Else Q = (e,q-q')  { replace q by q-q'; reduction of the
            flow by q' on e . It will be the last edge for which
            the flow is returned }
    End;
```

In fact, the two routines which are described above can be executed independently one after another. In fact, they have really a parallel implementation. The corresponding parallel description of these routines are presented in the next section. The general algorithm associated with these two routines is presented below.

```
Begin
    EXCESS(s) := $\Sigma$ c(s --> v), v $\in$ layer1
    PUSH(s,EXCESS(s)) { by pushing the flow from s this will result
       in some unbalanced vertices. They are put into Queue }
    While Queue is not empty Do ( still some unbalanced vertices)
       Begin
          take the first vertex of the Queue ( let it v )
          If v is not blocked Then  PUSH(v,EXCESS(v));
                                    RETURN(v,EXCESS(v));
           Insert all newly unbalanced vertices to Queue
       End;
End;
```

After finding the maximum flow in one layered network, the flow in the original network is updated. Then another layered network is constructed by applying the method described above for getting a layered network. The algorithm of maximum flow described above is applied again. This process is repeated until a layered network cannot be constructed. This mean that starting from the source , a sink cannot be reached. The flow then in the original network is surely maximal.

5.3. Example

The example which is treated in this section is the network represented by the Fig.3.1. in chapter 3. First of all, we construct one layered network from the original and it is represented by the Fig.5.1

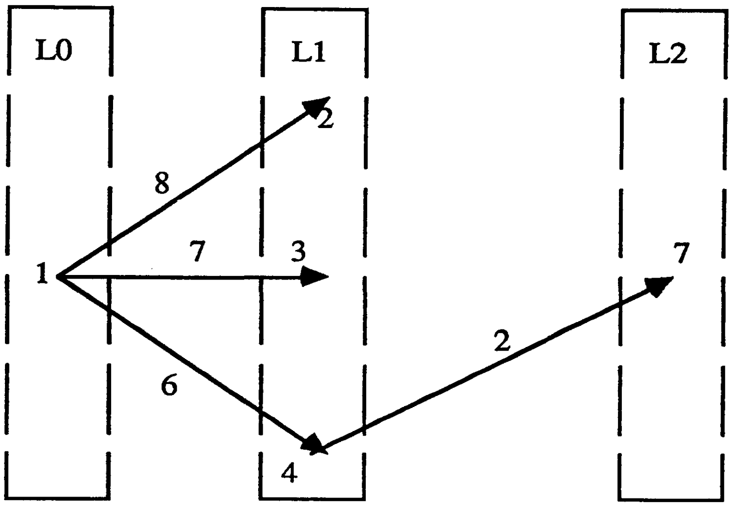

Fig.5.1. First layered network.

For each edge , the new capacity is associated with it.

At the beginning the queue is empty.

From vertex 1:
Push from vertex 1: 8 units through the edge (1,2), 7 units through (1,3) and 6 units through (1,4). It will result in some new unbalanced vertices which are {2,3,4}. No return of flow from 1.

From vertex 2:
Return 8 units through the edge (1,2).

From vertex 3:

Return 7 units through the edge (1,3)

From vertex 4:
Push 2 units through the edge (4,7). Return of 4 units through the edge (1,4). {7} is the newly unbalanced vertex.

From vertex 7:
No return of flow.

The maximum flow which have reached the sink is equal to 2. The flow in the original network is then updated. The Fig.5.2. presented below shows the flow associated with each edge. Each edge is represented by three values. The first one is the original capacity. The second one represents a lower bound while the last one indicates the flow.

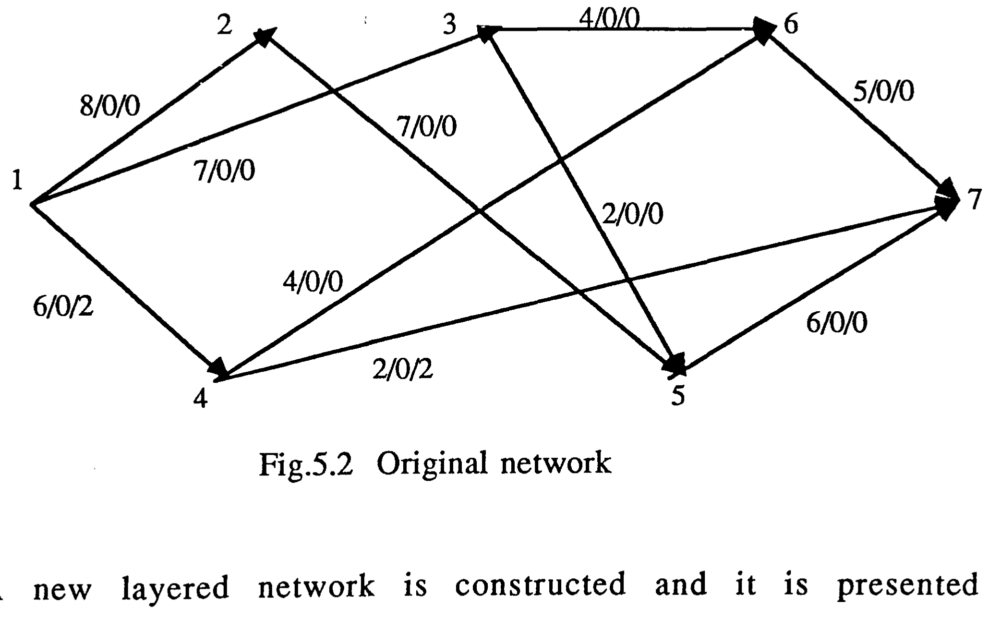

Fig.5.2 Original network

A new layered network is constructed and it is presented in fig.5.3.

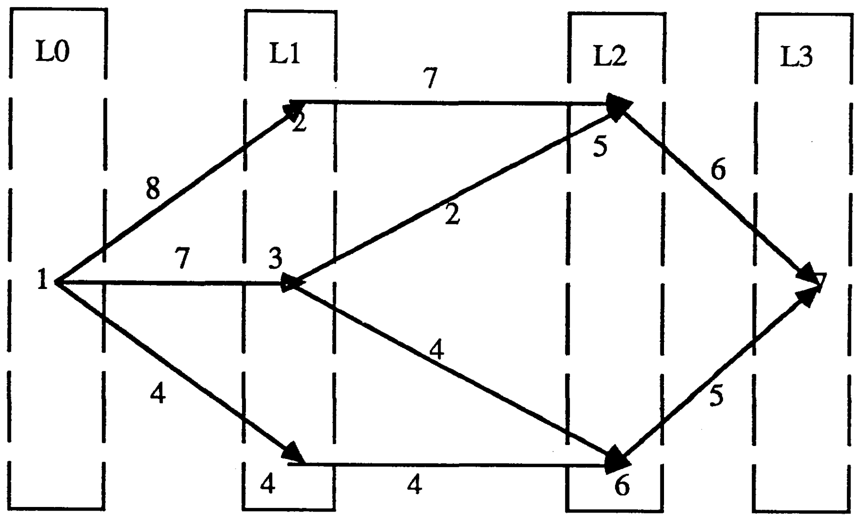

Fig.5.3 Second layered network.

From vertex 1:

Push from vertex 1: 8 units through the edge (1,2), 7 units through (1,3) and 4 units through (1,4). It will result in some new unbalanced vertices which are {2,3,4}. No return of flow from 1.

From vertex 2:

Push 7 units through the edge (2,5). {5} is the new unbalanced vertex. Excess(2) is currently equal to 1.

Return 1 unit through the edge (1,2).

From vertex 3:

Push from vertex 3: 2 units through the edge (3,5) and 4 units through the edge (3,6). {6} is the newly unbalanced vertex.

Return 1 unit through (1,3) because Excess(3) is equal to 1.

From vertex 4:

Push from vertex 4: 4 units through the edge (4,6). Excess(4) is now equal to 0.

No return of flow.

From vertex 5:

Push 6 units through the edge (5,7). The new unbalanced vertex is {7}. Excess(5) is equal to 3.

Return from 5 : 2 units through the edge (3,5) and 1 unit through (2,5). { 2,3 } become unbalanced.

From vertex 6:

Push 5 units through the edge (6,7). Excess(6) will be 3.

Return 3 units through (4,6). {4} becomes unbalanced.

From vertex 7:

No pushing and no return of the flow. The sink is reached.

From vertex 2:

Return of 1 units through (1,2)

From vertex 3:

Return of 2 units through (1,3)

From vertex 4:

Return of 3 units of flow through (1,4). No vertex then remains unbalanced.

The maximum flow which has reached the sink is equal to 11. The flow in the original network is then updated. The Fig.5.4. presented below shows the flow associated with each edge. The total maximum flow is then equal to (2+11 = 13).

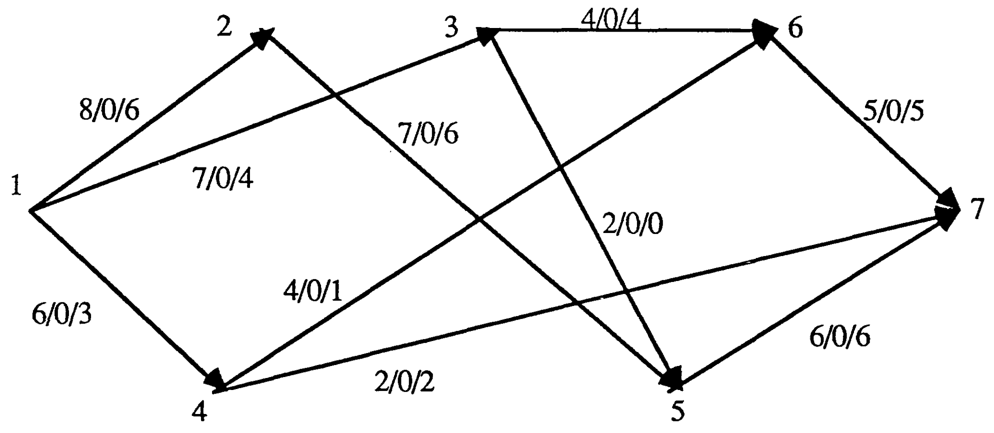

Fig.5.4. Original network

In the next step a new layered network is constructed. It is shown in Fig.5.5 .

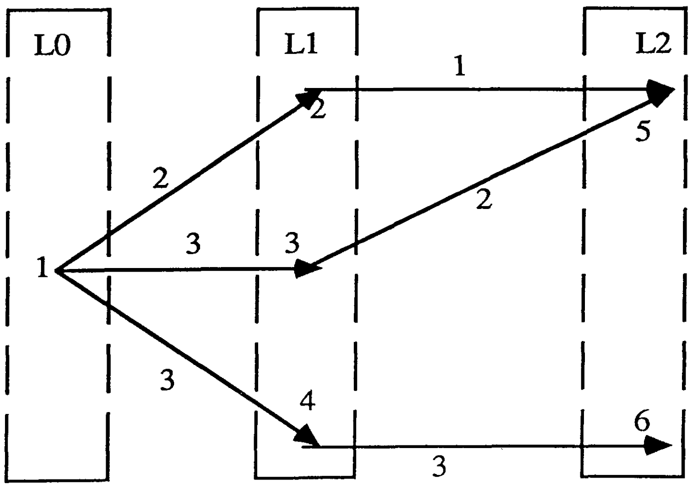

Fig.5.5. Third layered network.

In the third layered network, the sink is not reached, then the flow in the original network is certainly maximal . It is equal to the sum of all the maximum flows of the layered networks. The value of the maximum flow in the original network is equal to 13.

5.4. Description of the parallel method

Before giving the entire algorithm, we start first by describing the data structure used. Most of the structure is based on the partial sums tree (PS tree). The tree is a complete binary tree. An example of the tree with k given numbers ($a_1, a_2, \ldots, a_k$) is shown in Fig.5.6 . It contains $2^{\lceil \log_2 k \rceil}$ leaves, where the leftmost k leaves ($a_1, a_2, \ldots, a_k$) are called active leaves and the rest are all zeros. Every node of the tree is denoted by T[h,i], where T is the tree, h is its height in T and i is its serial number among other nodes. Note that the sums are shown next to the nodes of height (h > 1) between brackets in the Fig5.6 presented below

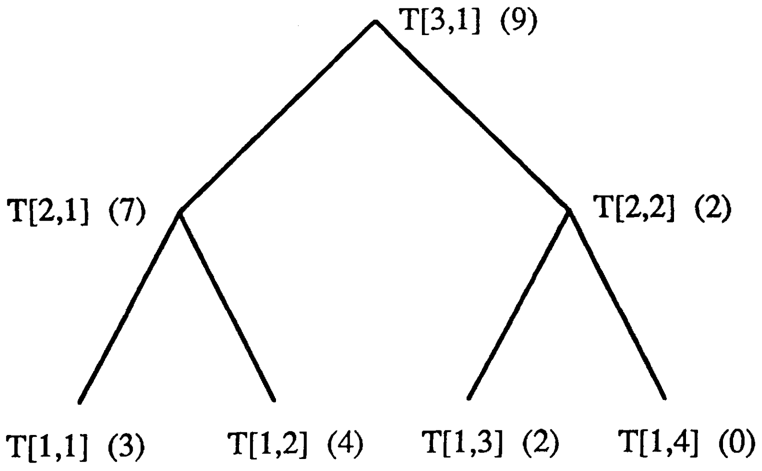

Fig 5.6. Example of a PS tree for a given numbers(3,4,2)

In their algorithm, they attach 4 different PS trees to each vertex v. The name of these trees are:

T-OUT(v) : this tree has a number of active leaves equivalent to the number of emanating edges from v. Each leaf is associated with one edge. The value attached to the leaf is the maximum amount of flow that can be pushed through the corresponding edge.

T-IN(v) : This tree has 2n times the number of edges which enter v of active leaves. n is the total number of vertices and 2n stands for the total number of pulses before the whole algorithm terminates. the demonstration is given on that paper [37]. The represented tree simulates the STACK(v). In the leaves, the flow quanta are recorded from the left to the right.

T-ACCESS(v) : the number of active leaves in this tree is equal to the number of edges entering v. The tree coordinates the activity of the processors that attempt to update the STACK(v) simultaneously.

T-SUM(v) : Each leaf of this tree is associated with one edge outgoing from v. The tree sums the amount of flow that is returned to v at given pulse.
Another tree is associated with each edge of the network.

T-EDGE(e) : This tree has 2n of active leaves. Each is associated with one pulse. It sums the amount of flow that is returned on e at a given pulse.

With the trees represented above 4 different primitive operations are performed.

CLEAR(i) : In this operation, the processor Pi puts zero to all the nodes from T[1,i] to the root. In fact, this operation can be executed by several processors simultaneously.

```
j := 1
While j <= h(T)   { h(T) is the height of the tree } Do
Begin
   $T[j,\lceil i/2^{(j-1)}\rceil] := 0$      { zeroing the nodes }
   j := j+1
End;
```

UPDATE(i,a_i) : this operation can also be executed by several processors. Here the value of the ith leaf is put to a_i and several other changes of the nodes are performed.

```
T[1,i] := $a_i$
j := 2
While j <= h(T) Do
Begin
    $T[j,\lceil i/2^{(j-1)}\rceil] := T[j-1,2\lceil i/2^{(j-1)}\rceil-1] + T[j-1,2\lceil i/2^{(j-1)}\rceil]$
        { changes on the nodes are performed }
    j := j+1
End;
```

SUM(i,S_i) : this operation performs the sum $S_i := a_1+a_2+ .. + a_i$

```
$S_i := a_i$
j := 2
While j <= h(T) Do
Begin
    If $2\lceil i/2^{(j-1)}\rceil = \lceil i/2^{(j-2)}\rceil$ Then
        $S_i := S_i + T[j-1,\lceil i/2^{(j-2)}\rceil-1]$;
    j := j+1
End;
```

FIND(α;k,ρ) : The operation return k,ρ for any given α satisfying:

```
{ $a_1+a_2+ ..+a_{k-1} <= \alpha <= a_1+a_2+...+a_{k-1}+a_k$
$\rho = \alpha - (a_1+a_2+ .. +a_{k-1})$ }.
j := h(T)
k := 1
$\rho := \alpha$
WHILE ( j > 1) Do Begin
    IF $\rho$ > T[j-1, 2k-1] Then  $\rho := \rho$ - T[j-1, 2k-1]
                                  k := 2k
                             Else k := 2k-1
                             j := j-1
                             End
```

For a simplicity of the description , Shiloach and Vishkin suppose that for every vertex v, a processor P(v) is assigned to it , and also every edge e has a processor P(e). In addition, every leaf of T-IN(v) has a processor attached to it and it is denoted as P(Q).

Implementation

In general the algorithm is divided into 4 routines. They are INITIALISE, PUSH, RETURN, CLEAN. The values of the tree nodes are all zero at the beginning of the first phase:

```
INITIALISE(v) : the routine is applied simultaneously for each
vertex v, at the beginning of each phase:
Instruct each processor P($e_j$ = v --> w)
Begin
    UPDATE(j,c'(e)) in T-OUT(v)
        { j is the index of $e_j$ among the edges which emanate from v
        and it is also the index of the leaf of T-OUT(v) associated
        with $e_j$. c' is its new capacity }
    f($e_j$) := 0  { the flow at the beginning of this phase is zero }
    Instruct P(v) :
    begin
    hd(v) := 0   { the pointer hd(v) points to the head of
                 STACK(v) i.e to the rightmost significant leaf in
                 T-IN(v) }
    k'(v) := 1   { k'(v) points to the smallest index of an edge which
                 is in AVAILABLE(v) }.
    End;
End ;{ end of routine INITIALIZE(v) }
```

PUSH(v,EXCESS(v)) : This routine as well as RETURN(v,EXCESS(v)) depend on $\alpha(v)$, $\rho(v)$, $k(v)$, $k'(v)$ for any vertex v. For an easier notation they will appear as α, ρ, k, k'. T[h(T),1] is denoted as T[root].

```
PUSH(v,EXCESS(v)) :
Instruct processor P(v) :
    Begin
    α := min ( EXCESS(v), T-OUT(v)[root])
        { the value in T-OUT(v)[root] is the total amount of flow that
        can be pushed from the vertex v and α is the amount that is
        going to be pushed from v}
    EXCESS(v) := EXCESS(v) - α   { reduction of the EXCESS(v) }
    FIND(α;k,ρ) in T-OUT(v)
        { The processor P(v) finds all the edges {$e_{k'}$,.......,$e_k$} through
        which the flow should be pushed forward from v. The edges
        $e_{k'}$,......,$e_{k-1}$ should be saturated and an amount of ρ should
        be pushed through $e_k$ }
    End;
Instruct each processor  P($e_j$ = v --> w)
If k' <= j <= k Then
Begin
    UPDATE(r,1) in T-ACCESS(w)
        {the leaf of T-ACCESS(w) that corresponds to $e_j$ has an index
```

```
  r}
SUM(r,$S_r$) in T-ACCESS(w)
  { $S_r$ is the serial number of processor P($e_j$) that wants to
  register the flow quanta in STACK(w)}
If j <> k Then $q_j$ := T-OUT(v)[1,j]
          Else $q_j$ := $\rho$ ;
  { $q_j$ is the flow that is going to be pushed through the edge
  $e_j$}
f($e_j$) := f($e_j$) + $q_j$        { incrementing the flow of the edge $e_j$ }
TOTAL(w) := T-IN(w)[root]
  { TOTAL(w) is the the total amount of flow that is pushed
  into w till now}
UPDATE(hd(w)+$S_r$, $q_j$)  in T-IN(w)
  {  update of T-IN(w). $S_r$+hd(w) is the index of the leaf in
  T-IN(w) that corresponds to the flow quantum ($e_j$,$q_j$). The
  flow quantum is recorded in STACK(w).}
UPDATE(j, T-OUT(v)[1,j] - $q_j$) in T-OUT(v).
  {update of the residual capacity of the edge $e_j$ in
  T-OUT(v).}]
hd(w) := hd(w) + T-ACCESS(w)[root]
  { hd(w) points now to the new head of the STACK(w) }
CLEAR(r) in T-ACCESS(w)
  { T-ACCESS(w) is cleared for further use in the next pulse }
```

```
    EXCESS(w) := T-IN(w)[root] - TOTAL(w)
End ;
Instruct P(v) : k' := k
Instruct P($e_d$ = u --> v)
    If EXCESS(v) > 0 Then
    Begin
        put vertex v blocked
        UPDATE(d,0) in T-OUT(u)
        { the vertex v becomes blocked. d denote the index of a leaf
        in T-OUT(u) that corresponds to the edge u --> v. Since u
        leads to a blocked vertex v, then the edge is removed from
        AVAILABLE(u) .}
    End;
{End of routine PUSH(v,EXCESS(v)) }
```

```
RETURN(v,EXCESS(v)):
Instruct processor P(v) :  FIND(T-IN(v)[root] - EXCESS(v); k, ρ) in
    T-IN(v). { Since the vertex v is unbalanced then an amount of
    flow equal to EXCESS(v) is going to be returned from v. The
    appropriate amount should be cancelled from STACK(v). P(v)
    searches for the edges which will remain in the stack and the
    other will be deleted}.
    EXCESS(v) := 0  { excess becomes zero }
Instruct each processor P($e_j$ = u --> v)
```

```
Begin
    If $k < j <= hd(v)$  Then $d_j := q_j$
                         Else $d_j := q_j - \rho$
    If $k < j <= hd(v)$  Then UPDATE (j,0) in T-IN(v)
        { these edges are deleted from the stack and T-IN(v) is
        properly updated.}
                         Else UPDATE(j,$\rho$) in T-IN(v)
        {the flow quanta is decremented for at most one edge }.
    UPDATE($r_j$,$d_j$) in T-EDGE($e_j$).
      { $r_j$ is the pulse number when $Q_j$ was pushed. It is also the
      index of a leaf of T-EDGE($e_j$). The total flow that is returned
      on this edge $e_j$ at this pulse is in T-EDGE($e_j$)[root] }.
    f($e_j$) := f($e_j$) - T-EDGE($e_j$)[root]  { decrement the flow on $e_j$}
    UPDATE($l_j$, T-EDGE($e_j$)[root]) in T-SUM(u)
      { in T-SUM(u), the index that corresponds to $e_j$ is $l_j$. The
      total amount of flow that is returned to the vertex u is
      found in T-SUM(u)[root] }.
    EXCESS(u) := EXCESS(u) + T-SUM(u)[root]
    CLEAR($r_j$) in T-EDGE($e_j$)
    CLEAR($l_j$) in T-SUM(u)
Instruct processor P(v) :  hd(v) := k  { update of the pointer }
{ end of routine RETURN(v,EXCESS(v))  }
```

CLEAN(v) : this routine cleans T-OUT(v) and T-IN(v) for further use . It is applied at the end of each phase.

```
Instruct each processor $P(e_j = v \text{ --> } w)$:
    CLEAR(j) in T-OUT(v).
Instruct each processor $P(Q_i = (u \text{ <-- } v, q_i))$ :
    CLEAR(i) in T-IN(v) for  $1 <= i <= hd(v)$.
    { end of routine CLEAN(v) }
```

It is very important to note that the RETURN routine cannot start before the end of PUSH routine in the same pulse.

5.5. Example

Referring to the example given in the previous section, a parallel BFS is performed to get a first layered network. Then the above routines are applied for each vertex v of the network. The flow is pushed simultaneously from one vertex through the available edges. The flow as well is returned simultaneously through the edges.

From vertex 1: Push 8,7,6 units through the edges (1,2),(1,3),(1,4) at the same time.

From vertex 2 : Return of 8 units through (1,2).

The return of 7 units of flow from a vertex 3 can be done in parallel with push from a vertex 4. 2 units are pushed through (4,7).

Return of 4 units from vertex 4 through the edge (1,4).

The flow maximal in the layered network is then found and it is equal to 2.

Another layered network is constructed (see Fig 5.3) using parallel BFS.

From vertex 1 : Push 8,7,4 units through the edges (1,2),(1,3),(1,4) simultaneously.

From vertex 2: Push 7 units through (2,5).

The return of 1 unit of flow from vertex 2 through the edge (1,2) can be done in parallel with the push from a vertex 3. In the later operation, 2 and 4 units are pushed through (3,5),(3,6). The return of flow from a vertex 3 can also be done in parallel with the push operation from a vertex 4. 1 unit is returned from a vertex 3 through (1,3) and 4 units are pushed through (4,6) .

The PUSH and RETURN operations continue until there is no unbalanced vertices. The maximum flow in a layered network is then found. Another layered network is constructed using BFS see Fig.5.5. The sink is not reached then the maximum flow in the original is found and it is equal to 13.

6. Parallel algorithm

6.1. Overview

This chapter is devoted to the description of the parallel methods that have already been presented in chapter 3. These algorithms have more less the same principle than the sequential methods. As was indicated previously, one of them is the original method of Edmonds and Karp and the other is following the same approach using Breadth-first search. Many believe that using BFS in parallel for traversing a graph gives an optimal bound and it is considered as the fastest especially for dense graphs [2] [13].

These algorithms are implemented for capacitated and bounded networks. The transformation of a bounded network to an auxiliary one is done in parallel and the corresponding method is presented followed by an example. The process for finding the paths from the source to the sink is described as well in parallel. The whole algorithm is designed to run on the transputers. In general it is divided into tasks, where each task is carried out by one transputer. More precisely, the algorithm is partitioned into processes because of the modularity structure which is required by Occam and the transputer. The processes communicate with each other using defined links. Finally, in the last part of this chapter some results obtained by the two methods are presented and a comparison between them is carried out.

6.2. General schema of the network of transputers and the processes loaded

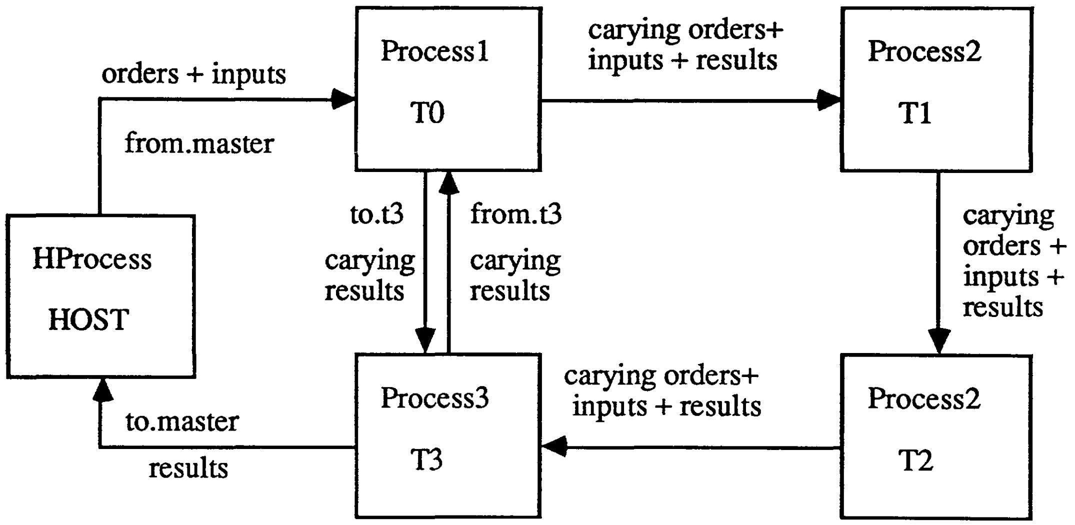

Fig.6.1 The network and the processes

In the last section of chapter four, we have presented our transputer network. It is formed from one transputer called "host transputer" (IMS B004) and four others called "external transputers" (IMS B003). In this section, we present the general configuration used in our implementation. In an other word, which process is loaded on which transputer. Each algorithm is divided into four major processes. They are HProcess, Process1, Process2, Process3 (see fig.6.1). The main process ("HProcess") is loaded on the host transputer. In particular, this process is responsible for giving the orders to the others which are loaded on the external transputers. An order is an instruction to perform certain task. Once an order is received by one of the processes resident on one of the external transputer, it sends it to the next process and starts performing the corresponding action to the instruction. The processes loaded on T0, T1, T2, T3 have in general the same structure. They are composed of a number of small processes where each one of them corresponds to an action which is performed when the order is recognised. The processes on T0, T1, .. etc have the following form:

```
    (instruction = identifier.no.of.the.order)
        action1 {process}
    (instruction = identifier.no.of.the.order)
        action2 {process}
    ......
```

Note that "Process1" on T0 gets its orders and inputs from the host process "HProcess" and carries the orders and results to the next process. "Process2" is loaded on all transputers between the first and the last (i.e on T1,T2), this for making the program more general. Each "Process2" on T1, T2 gets its inputs, orders from the previous process as well as the partial results of the predecessor and passes the results and the orders to the next one. "Process3" gets its main orders from the previous one and outputs the results if any to the host process ("HProcess"). The configuration in general is shown in Fig.6.1 .

6.3. Implementation

Before starting the description of the implementation, we describe some Occam statments which we will use to present the algorithms.

a) **IF**

```
    condition1
      process1
    condition2
      process2
    TRUE
      process3
```

This conditional construction consists of the keyword IF and one or more components, each slightly indented. Each component, consists of an expression (condition) and a little further indented process. The conditional executes by looking at a condition which has a value "True" and therefore the corresponding process is executed. If none of condition1, condition2 has a value "True" then the last component is executed (i.e process3). In this case, if TRUE were missing the whole construction stops.

b) **While** (condition)

```
    process
```

A While statment consists of a condition and a slightly indented process.

6.3.1. Data structure of the implementations

The data structure which we describe below constitutes an important part of the implementations. Since each transputer has its own local memory, then the data has to be kept local for each process on a transputer. The data structure is used for both methods unless stated.

A. Data structure for the host process

Two queues q.m, q.s are used. The insertion takes place at the end, while the removal is done from the top (i.e the beginning).

q.m : used to keep the vertices which have just been visited (first they were unvisited vertices) once a search is performed from one vertex. These vertices will be used to search for the others. **q.m** is used only for the method using the BFS.

q.s : is a secondary queue used to keep the vertices adjacent to the vertex from which the search will be performed. These vertices will be sent to the processes where they are visited and the flow which can reach them through the corresponding edges is calculated.

temp.list : Is a temporary list which is used for different purposes and has an important role for:

(i) Keeping the adjacent vertices of a node from which a search is performed.

(ii) keeping the vertices which will be sent to the processes .

(iii) keeping the unseen or unvisited vertices found from one search.

In addition we have also some data which is described in chapter3

Matrices **sizes** and **flow**

Vector **dad** used only for the method using BFS

Vectors **visited** (only for the original Edmonds and Karp).

B. Data structure for the processes on T0, T1, ...etc

Adjacency matrix for the bounds which corresponds to "sizes". It is denoted here by s.

The flow matrix is denoted by **f.**

The vectors **val, dad, visited**, and a temporary list **temp.** All these have the same structure as the ones described previously.

6.3.2. Assignment of the processors

The policy adopted for assigning the processors is :
At any vertex v, where the search should be performed, processors are assigned to the edges which emanate from v. In other words, each transputer will deal with one edge outgoing from v. If the number of edges which emanate from v is greater than the number of transputers available, then each transputer may be assigned more than once to different edges. In our case we treat each time at most four edges (see Fig 5.2).

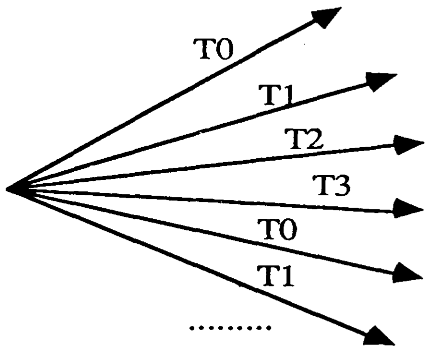

Fig 5.2

6.3.3. General form of the process executed on one of the external transputers

It has been said before that the processes loaded on the external transputers have in general the same structure. Each process is constructed as a repeating one which is waiting for orders to execute the corresponding actions and it is working as follows

```
While True  { repeat and wait for an instruction }
    - Wait for an order
    - Identify it
    - Send it to the next process
    - Starts executing the corresponding action
```

Naturally, the host process sends the instructions to Process1 resident on T0 through the channel "from.master". When Process1 receives an instruction, it forwards it to the next process and the later sends it as well to the next and so on till Process3 receives it. Process3 do not output the instruction to the next one because it is the last one.

6.4. Implementation of the second method

We present first the algorithm of the method which uses BFS. The general form of the processes which are executed on the external transputers is described in the following page. These are required in the implementation of the method and a brief description of each is given. The detailed implementation may be found in the listing.

```
While True
        from.channel ? instruction   { inputting the order }
    IF          { test for identification of the instruction }
      ( instruction = take.mat )
        to.channel ! instruction { output the order if necessary }
        { input from the channel the vertices source and sink,
        output them to the next process and input as well the
        matrices sizes and flow row by row. Each row input will
        be output directly to the next process. All this input
        must reach Process3}.
      ( instruction = initialisation )
        { Output the instruction to the next processes. In this part,
        each process initialises the vectors dad, val, and visited to
        zero as well as the variables used }
      ( instruction = vertex.priority )
        {Output the instruction to the next if necessary. Input the
        vertex.pr. In fact, it is from this vertex that a search
        is performed. If this vertex is the source then the value of
        the flow outputted from it, is put to a maximum i.e
        val[source] is maxint . Output vertex.pr to the next
        processes}
      ( instruction = split.list )
        { Output the instruction to the next processes if necessary.
        Receiving or inputting a list of the adjacent vertices to the
        vertex.pr. The list inputted does not exceed the total
        numbers of the transputers. Each process in one transputer
```

will take one vertex (the first in the list) and output the rest to the next process. The received list is terminated by an indicator "end.data". }

(**instruction = perform.search)**

{ Output the instruction to the next processes. Each process will verify whether or not the vertex taken from the split (or divided) list is visited. It will look also at the maximum flow which can pass from the vertex.pr to the vertex taken. Make an update of the vectors val, dad, and visited if necessary. }

(**instruction = pass.unseen)**

{ Output the instruction to the next processes. Each process which finds the unvisited vertex will send it to the last process where they are kept in a temporary list which is temp. Process3 updates the vector visited. }

(**instruction = get.unseen)**

{Forward the order till it is received by Process3. The last process is instructed to communicate the unvisited vertices to the host process in order to add them to q.m where further search will be performed from them. }

(**instruction = changnetw)**

{ Output the instruction to the the next processes. Each process is instructed to start the transformation of the original network to the auxiliary network. Each process is allocated a certain number of rows of the matrix sizes to be modified.}

(**instruction = update.vect.val)**

{ Output the instruction to the next processes. Each process has attempted changes on the vector val through execution of the action corresponding to perform.search. Process1 will send val to the next and each successor will add the changes to val which were made by the process itself } .

(**instruction = take.vect**)

{ Forward the instruction to the last process. Process3 is instructed to pass the vectors val, dad and visited to the other processes. Process1 gets them through the channel from.t3 and forwards them to Process2 (on T1, T2). In fact Process3 possesses the updated vectors. }

(**instruction = get.dad**)

{ Forward the instruction to the last process. Process3 is ordered to pass the vector dad to the host. This vector contains the path found in the network. The value val[p] is passed also to the host. This value will allow the host process to check whether the maximum flow is found or not.}

(**instruction = find.path.min)**

{ Output the instruction to the next processes. Here the path is transmitted from the host where each process gets it. The value by which the flow is increased is calculated by the last process. The value min is passed to the host. }

(**instruction = update.flow**)

{ Forward the instruction till Process3 receives it. Each process is ordered to update the flow of the edges after finding the path and after each process receives the value min. This means the update of the matrix f of each process.}

(**instruction = get.matf**)

{ Forward the instruction to Process3. The last process is instructed to pass or communicate the matrix flow to the host. This happens only when a maximum flow in the auxiliary network is found to allow the host to make the corresponding update. }

(**instruction = pass.serial.number**)

{ Forward the instruction to the processes. Each process on a transputer will get its serial number for a differentiation among the others. For example, the process on T0 has its serial number kk = 1, and the process on T1 is kk = 2 , etc..}

6.4.1. Transformation of the network

Before we proceed to the main algorithm, we expand in greater detail the transformation of the network into an auxiliary one indicated by the process corresponding to "changnetw" on page 79. The sequential algorithm has been already presented in the chapter 3. The following algorithm is the parallel version. The transformation consists of the modification of the adjacency matrix that represents a network. We divide the matrix into np parts (where np is the number of the external transputers, for example , in our case np = 4). Each part is transformed by one process which resides on one of the external transputers (see Fig.6.3). The number of rows (nr) that are allocated normally to one process is calculated from the expression :

nr := n/np   { n is the number of vertices (rows as well) }

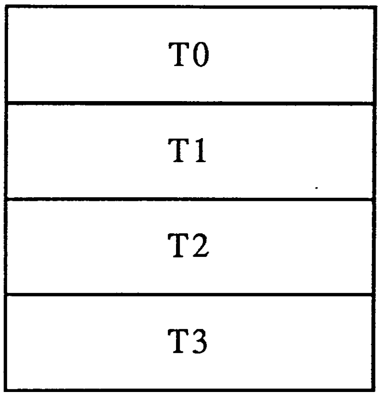

Fig 6.3

However, we have to consider the remainder of the division (n/np). In general, there are three cases:

(i) n < np { number of rows < number of transputers}

From the expression above nr will be zero, but we will allocate to each transputer one row (i.e nr := 1) and there will be at least one transputer which does not transform any row.

(ii) n = np { nr := 1 which is obvious }

(iii) n > np

If Rem(n/np) > 0 then in this case, we have to add one more row for each of the first processes until the remaining is exhausted.

Part of the algorithm executed by the host process

```
- Send to the processes the instruction "take.mat" followed by the
  vertices source and sink and also the matrices sizes and flow .
  { the reason for sending the matrix flow is that after the change
  of the network, all the data will be ready for further use }
- Instruct the processes to get the serial number
  { The host sends kk = 1 for T0, then T0 will pass kk to T1 and
  when T1 receives it, it increments kk by one and
  communicates the value to next etc.. }.
IF
  n < = np
    nr := 1
  TRUE
    nr := n/np
- Send the command "changnetw" to all the processes for
  changing a network
- Communicate to the processes the values nr, n
- Get from Process3 the (n+1)th row and the (n+2)th column.
  { In fact there are two rows and two columns added to the
  matrix after modification, but one row and another column are
  all zeros , we do not need to get them. }
```

The following procedures are parts of the processes executed on T0, T1, ...etc. They represent the actions which correspond to the instruction "changnetw" for changing a network.

Change of network carried out by "Process1"

```
from.master ? nr; n
   { take the values nr and n communicated from the host }
output ! nr; n
  { output them to the next process through the channel output }
IF
  ((np*nr)+kk) <= n  { then in SEQ (sequence) what follows }
    { this test is to see whether or not there are remaining rows}
      sup := ((kk*nr)+1)  { sup is the superior bound (i.e the last
         row which is treated by this process. We are in the case
         where n > np and Rem(n/np) > 0.}
      step := nr + 1  { step is the number of rows to treat }
  TRUE  { in SEQ } { TRUE to mean the opposite case }
      sup := kk*nr
      step := nr   { case where Rem(n/np) = 0 }
output ! sup  { sup is outputted to the next process, the reason is
    that the next process will start the transformation from the
    (sup+1)th row }
SEQ i = 1 FOR step
  SEQ j = 1 FOR n
    IF
      (s[i][j] > 0) AND (s[j][i] <> 0) { then in SEQ }
          s[i][j] := s[i][j] + s[j][i]
          s[n+1][j] := s[n+1][j] - s[j][i]
          s[i][p] := s[i][p] - s[j][i]
          { this part is already described in the chapter 3 and
```

```
            its main task is to make the changes to the bounds
            and add the new edges. We will call this part "
            **process modification** " for a reference when we
            treat other parts. ( p is the new sink = n+2)}
        **TRUE** SKIP { process which does nothing }
**SEQ** i = 1 **FOR** sup
  output ! [s[i] FROM 1 FOR p]
    { it outputs the rows updated by this process to the next one.
    The output is done row by row}.
output ! [s[n+1] FROM 1 FOR p]
    { send the $(n+1)^{th}$ row to the next process. The row contains
    the edges added by this process, since q := n+1 is the source.}
**SEQ** i = 1 **FOR** p
  from.t3 ? [ s[i] FROM 1 FOR p]
  output ! [ s[i] FROM 1 FOR p]
  { After all the processes have finished the transformation, The
  whole updated matrix is found in Process3. This process
  "Process1" gets it from the channel "from.t3". The modified
  matrix sizes will be available for further use. }
```

Change of network carried out by "Process2"

```
input ? nr; n  { input from the previous process }
output ! nr; n  { output to the next process }
input ? inf   { Get the limit where the previous process stopped
            (i.e the sup of the previous process) }
IF
  ( inf = n)  { in SEQ }
      sup := n
      step := 0     {Case where n < np, this process do not proceed
          to the transformation of any row}
  TRUE   { opposite case }
      IF
        ((np*nr)+kk) <= n   { in SEQ }
            sup := inf+nr+1
            step := nr+1   { case where Rem (n/np) > 0 }
        TRUE   { in SEQ }
            sup := inf+nr
            step := nr   { case where Rem(n/np) = 0 }
output ! sup
SEQ i = (inf+1) FOR step
  "process  modification"
    { "Process modification" which is described previously in
    Process1. The modification starts from (inf+1) in this part. }
SEQ i = 1 FOR inf
    input ? [s[i] FROM 1 FOR p]
    output ! [s[i] FROM 1 FOR p]
```

```
        { input the rows transformed by the previous process and
        output them to the next process. Each row inputted is
        outputted directly to the next. }
SEQ i = (inf+1) FOR step
  output ! [s[i] FROM 1 FOR p]
        { output the rows transformed by this process }
SEQ i = 1 FOR (kk-1)
    input ? [temp FROM 1 FOR p]
    output ! [temp FROM 1 FOR p]
        { Input the (n+1)th rows transformed by the previous
        processes and output them to the next }.
output ! [s[n+1] FROM 1 FOR p]
        { output the (n+1)th row updated by this process }
SEQ i = 1 FOR p
    input ? [ s[i] FROM 1 FOR p]
    output ! [ s[i] FROM 1 FOR p]
    { input the whole matrix sizes which is sent by the previous
    process. Each row inputted is outputted directly to the next
    process. Process2 on T2 does not output the matrix sizes to
    the last process (i.e Process3). In fact, Process3 possesses it.}
```

Change of network carried out by "Process3"

```
input ? n; nr    { input previous values n,nr }
input ? inf    { input the value inf }
IF
  (( inf+nr) = n)
    step := nr  { the remaining rows are treated by this process}
  TRUE
    step := 0  { case where n < np }
SEQ i = (inf+1) FOR step
  "process modification"
    { the modification starts from the value (inf+1) }
SEQ i = 1 FOR inf
  input ? [s[i] FROM 1 FOR p]      { input the rows updated till now}
SEQ i = 1 FOR (kk-1)
    input ? [temp FROM 1 FOR p]
    SEQ j = 1 FOR p
      s[n+1][j] := s[n+1][j] + temp[j]
      {input the (n+1)th rows updated by other processes, add
      them all together to make a full updating. }
s[n][1] := k1  { the edge (n,1) added to the auxiliary network is
    assigned an integer which is big enough, we denote it by k1 }
to.master ! [ s[n+1] FROM 1 FOR p]
    { send to the host the (n+1)th row }
SEQ i = 1 FOR n
  to.master ! s[i][p]
    { send to the host the (n+2)th column}
```

```
SEQ i = 1 FOR p
  from.t3 ! [ s[i] FROM 1 FOR p]
    { send the whole matrix to Process1 where it will be
    forwarded to the other processes. }
```

6.4.2 Example

Applying the algorithm above for transforming the network to the second example of fig.3.2 in chapter 3, all steps may be stated as:

HProcess

The host process calculates nr which is equal to 7/4 := 1.

(nr := 1) is normally the number of rows which should be treated by one transputer, but there are remaining rows which will be split among the processes. The host process instructs the processes to start the modification and sends the values (nr:= 1; n := 7) .

Process1

This process notices that there is a remaining, then calculates the new limits (sup := 2; step :=2). The number of rows treated by this process is (step := 2). It will treat the rows from 1 to sup (i.e from 1 to 2). This process sends the value sup to the next process so that it can start the modification from the row after sup.

The results obtained after the transformation of the rows are:

$$\begin{array}{cccccccc} 0 & 6 & 5 & 5 & 0 & 0 & 0 & 0 \;\; 5 \\ -2 & 0 & 0 & 0 & 4 & 0 & 0 & 0 \;\; 3 \end{array}$$

The row whose number is (n+1) is also updated :

```
0 2  2 1  3 0 0  0 0
```

All these rows are sent to the next process "Process2".

Process2

a) On T1

This process gets all its inputs which are nr,n, inf; the value inf contains the sup of the previous process. It then calculates the values (sup := 4; step :=2) from the expressions. The modification starts from the value (inf+1) to sup (i.e from 3 to 4). The results are

```
-2 0  0 0  1 3 0  0 2
-1 0  0 0  0 4 2  0 0
```

The $(n+1)^{th}$ row in this process is

```
0 0  0 0  1 1 0  0 0
```

This process gets the rows transformed by the previous process as well as the $(n+1)^{th}$ row. it outputs them to the next process. Then, it sends all the rows from 3 to the value sup(i.e from 3 to 4) to the next process. It also transmits the $(n+1)^{th}$ row updated by this process to the next process.

b) On T2

In this part, Process2 does exactly the same steps it did previously. The values after calculations are (step :=2, inf:= 4, sup:=6). The modification starts from row 5 to 6. The modified rows 5 and 6 are as follows:

```
0 -3 -1 0  0 0 3  0 3
0 0  -1 0  0 0 3  0 2
```

The (n+1)th row is:

```
0 0  0 0  0 0 5  0 0
```

This process receives the rows updated previously (i.e from 1 to 4) and the (n+1)th rows of the previous processes. It outputs these to the next process. It also sends to the next process the rows (5 to 6) and the (n+1)th row updated by this process.

Process3

In this process, there is only one row left which is the 7th row. It just adds the edge (n,1) (i.e (7,1)). The edge (n,1) has a bound k1.

```
k1 0  0 0 -3 -2 0  0 0
```

This process receives the modified rows of the previous processes from 1 to 6. It receives the (n+1)th rows updated previously. The (n+1)th rows are added together as they were received to make a full updating. Then, it outputs to the host process the (n+1)th row and the (n+2)th column. The whole modified matrix sizes is outputted to the other processes through a channel from.t3.

6.4.3. Algorithm for finding the paths and max flow

In this section we describe the parallel method for finding the paths and how the flow is updated. The algorithm is represented mainly by two main processes executed on the host transputer. These two processes are the process path and search. The process path continually calls the process search for finding one path.

(a) Process path(s) { s is the matrix sizes }

```
{netb is a boolean variable which is set to the value TRUE when
the network is bounded, once the maximum flow in the auxiliary
network is found, the value is set again to FALSE. Initially the
value is FALSE. }
IF
  netb
    p := n+2
    q := n+1      { q,p source and sink of the auxiliary network }
  TRUE
    q :=1
    p := n        { Source and sink of the original network }
    Send.mat.procs(from.master,sizes,flow,p,q)
    Send.pass.serial.number(from.master)
WHILE still  { still initially is TRUE (i.e flow maximal is not true ) }
  Search(s)     {  call the process search for finding a path }.
  from.master ! " get.dad"
  Receive the array dad and val[p].
  from.master ! "find.path.min"
  Retrieve the vertices of the path and pass them to processes.
  to.master ? min   { value min is returned}
  from.master ! "update.flow"
  Increase the value of flowmax value if necessary.
```

```
IF
  val.p = 0
    still := FALSE
    {the source cannot be reached i.e the maximum flow
    is found. val.p is equal to val[p] which is returned by the
    last process }
  TRUE
    SKIP
{ End of While }
```

Send.mat.procs(from.master,sizes,flow,p,q)

```
  from.master ! "take.mat"
  send the source and sink
  send the matrix sizes ligne by ligne
  send the matrix flow ligne by ligne
```

Send.pass.serial.number(from.master)

```
  from.master ! "pass.serial.number"
  kk := 1
  from.master ! kk
    { the first external transputer will be numbered by 1 }
```

(b) Process search

```
from.master ! "initialisation"
Put a vertex into q.m { this vertex should be the source }
WHILE (q.m not empty )
  Take first vertex from q.m {this vertex is called vertex.pr}
  Send.vertex.pr(from.master).
  Get adjacency list of vertex.pr and copy it in temp.list.
  Put a list (temp.list) into q.s
  WHILE ( q.s is not empty )
    Take a list containing a maximum of np vertices from q.s and
    copy it in temp.list .
    Send.split.list(from.master,temp.list).
    from.master ! "perform.search".
    from.master ! "pass.unseen".
  Send.get.unseen.vert(from.master,temp.list)
  IF
    temp.list is not empty ( i.e number of vertices > 0 )
      Put list obtained (temp.list ) into q.m
    TRUE
      SKIP
  from.master ! "update.vect.val"
  from.master ! "take.vect"
```

```
Send.vertex.pr(from.master)
   from.master  !  "vertex.priority"
   from.master ! vertex.pr   { send the vertex (vertex.pr) }
```

```
Send.split.list(from.master,temp.list)
   from.master  !  "split.list"
   SEQ k=0 FOR length.list
      from.master  !  temp.list[k]
      from.master  !  end.data
      { the full list is sent followed by an indicator 'end.data'. }
```

```
Send.get.unseen.vert(from.master,temp.list)
   from.master  !  "get.unseen"
   Receive the unseen vertices and copy them into temp.list
```

Note that the processes already described in pages 78, 79, 80, 81 occur naturally in the above algorithm and their names are listed between " " .

Detailed description of some operation used

1. Retrieval of the path and change of flow

The host process retrieves the path using the dad array by backtracking from the sink to the source. The process is already described in the chapter 3. The vertices which form the path are sent to the processes on the external transputers as they are retrieved. The first process calculates the residual capacities and sends them to the last process where the later calculates the minimum of all values to find the amount by which the path should be increased (it is denoted by min). The value of min is communicated to all the processes. The host then instruct the processes to update the flow of the edges i.e the matrix f, while itself is updating the value of flowmax.

2. Operations performed on the queues

1. Put a vertex

- Add the vertex at the end of the queue
- Increment the index (i.e the number of vertices in the queue). The index is necessary for testing whether the queue is empty or not.

3. put a list of vertices

- Add the list at the end of the queue.
- Increment as well the index.

4. take a list from a queue

- Remove a list of np vertices from the queue if there exist. If there is less than np vertices than all the vertices are removed.

- Decrement the index.

6.4.4. Example

Taking the same example of fig.3.2 in chapter 3, we will just give the steps of one iteration.

After a modification of the network, all the matrices are resident on the transputers. Executing the process path on the host , it calls the process "search". The host instructs the processes (Process1,....) to initialise the vectors val, dad,visited to zero. Starting from the source (q :=8), the queue "q.m" contains initially q.

A vertex.pr is removed from q.m which is "vertex 8".

The adjacency list of the vertex 8 is searched which is temp.list :

temp.list = [2,3,4,5,6,7]

This list is put in q.s. The first four vertices [2,3,4,5] removed from q.s, are sent to the processes for splitting. (i.e Process1 gets "vertex 2", Process2 on T1 gets "vertex 3", Process2 on T2 gets "vertex 4, Process3 gets "vertex 5").

The host instructs the processes to perform a search (i.e the vertices 2,3,4,5 are visited). The flow in each edge (8,2),(8,3),(8,4),(8,5) is calculated and the values in each vector val in each process is updated according to the flow of the visited edges. the vertices 2,3,4,5 become visited vertices and they are regrouped in the last process in the temporary list "temp". The last two vertices 6,7 are removed from q.s and sent to the processes where they are visited in the same manner as the predecessors. the vertices

6,7 are also regrouped in the list temp. All these vertices are passed to the host and added to q.m:

q.m := [2,3,4,5,6,7].

The host process instructs then the processes to assemble all the changes made by each process to the vector val so that the whole updated vector is found in Process3. The updated vectors val, dad, visited will be then sent to the other processes by Process3. Taking again the vertex.pr := 2 from q.m, all the steps above are repeated until q.m is empty. The path then is considered as found.

6.5. Implementation of the first method

We propose to present in this section the original method of Edmonds and Karp which finds a path leading to an increase of a flow by the highest possible augmentation. In general, there is a lot of similarities between the algorithm of the first method and the second one. The processes which are listed below are the same as those described in pages 78, 79, 80 and 81.

(instruction = take.mat) ;
(instruction = initialisation) ;
(instruction = vertex.priority);
(instruction = split.list) ;
(instruction = perform.search) ;

(instruction = changnetw)
(instruction = update.vect.val)
(instruction = get.matf)
(instruction = pass.serial.number)
(instruction = pass.unseen)

The difference is mainly in retrieving the path and updating the flow and also in searching for the next vertex where the search should be performed.

(instruction = get.dad)

{ The instruction is forwarded to Process3. Process3 is instructed by the host to pass the value dad[p] and the value val[p].}

(instruction = find.b)

{ With this method, each time, we need to search for the vertex v which outputs the highest amount of flow so that the next search starts from v. It is needed to calculate the maximum value in the vector val for the vertices where the search has not been performed yet from them. The vertex v corresponds to the vertex which possesses that value. The maximum is calculated by the processes. The vector val is divided into four parts . Each process will find the maximum in one part. In this specific process which corresponds to "find.b" the limits are calculated (i.e which part of the vector val is allocated to which process). The calculation of the limits in each process is done in the same manner than the calculation of the number of rows when the network is transformed.}

(instruction = find.next.v)

{ Forward the instruction to the last process. Here, each process calculates the maximum value of the val part allocated to it. The value is denoted by max. Each process will pass the value found to Process3. Process3 calculates the highest value of the max values obtained. The value of max is the vertex number where the next search should resume. It is passed to the host and it is considered as vertex.pr. }

(instruction = find.path.update.flow)

{ The instruction is forwarded to the last process. Process3 retrieves the path. The vertices forming the path are passed to Process1 and to the host as they were retrieved. Process1 will forward them to Process2. At the same time, all the processes update the flow values of the edges forming the path (i.e update of the matrix f. The augmenting value is val[p].}

6.5.1 Processes for searching paths and flow

Below , we present the main processes for finding the paths and the flow in the network. The two processes are already described and presented previously in section 6.4.3 . The process path is almost the same than the previous one, we include only small changes. It is presented in the next page. The process search will be also presented because it contains some changes.

a) Process path(s)

```
IF
  netb
    p := n+2
    q := n+1
    from.master ! "find.b"
  TRUE
    p := n
    q := 1
    Send.mat.procs(from.master,sizes,flow,p,q)
    Send pass.serial.number(from.master)
    from.master ! "find.b"
WHILE still
  search(s) { call the process search }.
  from.master ! "get.dad"
  Receive the values dad[p] and val[p]
  from.master ! "find.path.update.flow"
  Receive the path from the last process
  Update the flowmax value.
  IF
    val.p = 0
      still := FALSE
    TRUE
      SKIP
```

---

b) Process search(s)

```
from.master ! "initialisation"
initialise the vector visited to zero.
vertex.pr := q  { starting from the source  }.
WHILE (vertex.pr <> 0)
   Send.vertex.pr(from.master)
   Mark vertex.pr "visited" (i.e visited[vertex.pr] := 2).
   Get the adjacency list of vertex.pr and copy it in temp.list.
   IF
     temp.list is not empty
       put a list (temp.list)  in q.s
     TRUE
       SKIP
   WHILE ( q.s is not empty )
     Take list of np vertices ( our case four).
     Send.split.list(from.master,temp.list)
     from.master ! "perform.search"
     from.master ! "pass.unseen"
   from.master ! "update.vect.val"
   from.master ! "take.vect"
   from.master ! "find.next.v"
   receive vertex from the last process. { it should be the next
                                           vertex.pr}
```

Note that the calls to the processes Send.mat.procs,Send.pass.serial

Send.vertex.pr, Send.split.list, Send.get.unseen can be found in pages 93, 95.

The operations concerning the queues which are needed are :

Put a list

Take a list

Get the adjacency list

The only difference is found in "get adjacency list" . In fact, only vertices where the search has not been performed yet from them are searched.

6.6. Presentation of the results

The two algorithms described previously, are applied to two examples. The first example (example1) is presented by fig3.2 in chapter 3, while the second one is shown below for its importance.

The network is presented by its adjacency matrix sizes.

$$\begin{matrix} 0 & 6 & 7 & 5 & 4 & 6 & 0 \\ -1 & 0 & 3 & 4 & 6 & 7 & 8 \\ -1 & 0 & 0 & 5 & 6 & 0 & 3 \\ 0 & 0 & -1 & 0 & 4 & 6 & 7 \\ -2 & -1 & 0 & -1 & 0 & 9 & 7 \\ -1 & -2 & 0 & -2 & -1 & 0 & 8 \\ 0 & -3 & 0 & -3 & -1 & 0 & 0 \end{matrix}$$

1. The paths which are found, by applying the first method (original of Edmonds and Karp) are as follows :

[8 7 1 9]	(5)
[8 6 7 1 2 9]	(5)
[8 5 7 1 4 9]	(4)
[8 7 1 3 4 9]	(2)
[8 2 9]	(1)
[8 3 9]	(1)
[8 4 5 9]	(1)
[8 6 7 5 9]	(1)
[1 3 7]	(3)
[1 6 7]	(2)
[1 5 7]	(2)
[1 4 7]	(1)
[1 3 4 7]	(1)

The number between brackets is the amount by which the flow is increased in the path.

2. The paths obtained by the second method are:

[8 7 1 9]	(5)
[8 5 9]	(2)
[8 6 7 1 2 9]	(5)
[8 2 9]	(1)
[8 7 1 3 9]	(1)
[8 5 7 1 4 9]	(2)
[8 4 9]	(1)
[8 3 4 9]	(1)
[8 7 1 4 9]	(1)
[8 6 7 1 4 9]	(1)

[1 3 5 7]	(4)
[1 6 7]	(2)
[1 5 3 7]	(2)
[1 3 7]	(1)
[1 4 7]	(1)

The total paths found by the first method is 13 whereas in the second the number is 15. Therefore the first one is optimal sequentially.

We will now show some tables representing the timing of the two methods applied on the examples. The algorithms are run on

- A single transputer (host)
- On the the network of transputers.

A program run on a single transputer is considered as sequential.

Host	3057
Network	2502

Tab.1 First method on example1

Host	3914
Network	3041

Tab.2 First method on example2

Host	3135
Network	1750

Tab.3 Second method on example1

Host	4971
Network	2690

Tab.4 Second method on example2

From all the tables displayed above, we can notice that the first method when it is run on a single transputer is faster than the second one and gives better results. It can be deduced from the number of paths which are found by both methods. The first method is optimal if the two sequential algorithms are compared.

When it comes to the execution of the two methods on the network, the second one gives better results than the first one. The percentages of economy of times that are obtained on the network compare to the single transputer are:

- First method : between 20 to 25 percent.
- Second method : between 45 to 50 percent.

We have mentioned at the beginning of this chapter that the second method has an advantage over the first one, because it does not need to look for the vertex from which the search should be resumed. In fact, the vertices from which the search is performed are in the queue as they were visited. In the first method, a search is always done from a vertex which outputs the highest amount of flow. More precisely, the vertex where the search is not already performed from it and whose value in the vector val is the highest compared to the others. It corresponds to the action performed for the instruction "find.next.v" in the program. The task of find.next.v is divided among the processes (Process1, Process2, Process3). Each process finds a maximum value in one part of the vector val allocated to it for vertices where a search has not been performed yet from them. In the next step, each process will send the maximum found to the last process. The latter will calculate the maximum of these values. This value then represents the next vertex where the search is performed and it is passed to the host. Now, we show the timing of the process "search" presented before for the whole algorithm.

method1	Process search	2250
method2	Process search	1330

Tab.5 Timing of process search on example1

The time of the process action which corresponds to "find.next.v" on example1 is 1360 for the whole algorithm. Then 1360 out of 2250 has been consumed by "find.next.v". The time also includes the overhead which is the time consumed by the communication. The overhead comes from passing the vertices from one transputer to another. In fact, the network in example1 contains (7) vertices, and the the number of paths found is (10). To find a path, the search is performed from every vertex of the network, then it results in (7) times the instruction find.next.v is executed. For all the paths, the instruction is executed (70) times (7*10). However, the auxiliary network contains more than (7) vertices. Then we can say that at least (70) times the instruction is repeated. The overhead which occurs in one execution of find.next.v is multiplied by (70). The time which will be obtained must be great.

We will present now a table indicating the timing of communication cost:

Sending a vertex	From (0 to 1) units
Sending a vector (7 vertices)	From (2 to 3) units
Sending a matrix (7*7)	14 units

Tab.6 Examples of overhead

The timing for the table displayed above is done from one tranputer to another one.
We have also timed the process which sends the matrices from the host to the others and the result is 30 units.

The main reason why we have no more than 50 percent of economy of time for the second method is the cost of communication. Taking the process action which corresponds to "take.vect" that is responsible for sending the vectors val, dad, visited from Process3 to the other processes, the time for communicating them takes 5 units to 6 units. The sending of these vectors is done after a search from one vertex. Then, at least (70) times the above time is consumed. The result will be at least 350 units for the whole algorithm. The same thing happens for the first method.

Below, we present the timing for changing a network to an auxiliary one.

Change of network (SEQ)	51
Change of network (PAR)	80

Tab.7 modification of a network on example1.

The sequential modification is done on the host process and it includes the time for sending the matrix sizes to the other processes after changes.
The parallel one is done on the network and it includes also the time for sending the matrix until all the processes get it. The reason why the time of the parallel method is greater than the sequential is also due to the overhead. The communication is mainly based on sending the rows of the matrix sizes from one process to another. This is the main disadvantage of the method.

The algorithms presented previously are general. They are suitable for networks which contain at least three transputers (i.e np >= 3) and which have the same configuration as the one presented by fig.6.1 .In fact, we place a process (Process2) on all transputers between the first and the last one. However, this leads to some problems such as some tasks are resolved in sequence. As an example, the action which corresponds to "update.vect.val" for Process2 on T2 cannot add the changes made to the vector val for a full updating until the same action for Process2 on T1 has finished. In many cases as well, Process2 on T2 cannot output to the next process until it has received the outputs of the previous process (i.e the outputs of Process2 on T1) for keeping the order.

7. Conclusion

The design of parallel algorithms for various parallel computer architectures is motivated by factors such as speed and the need to solve complex problems of practical interest. With the continuing decrease in hardware cost, the objective is to use a number of processors for a gain in computational speed.

Due to the growing number of parallel computer architectures and the algorithms developed on these for a large class of problems, it has become increasingly difficult for a user to select a particular algorithm for any given application. In fact, a choice is usually decided by factors such as ease of implementation of the algorithms and cost - effectiveness of the computers.

Since the design of parallel algorithms depends mainly on the parallel machine, it is necessary to keep the architecture in mind when designing a parallel algorithm. There is no universal method for designing parallel algorithms. In this thesis we have presented two implementations for the problem of a maximum flow in a network. The implementations are designed for a network of transputers which falls in the category of MIMD machines. Another algorithm is also described which is the Shiloach and Vishkin method. They have designed it for MIMD machines with shared memory where the processors communicate through it. It is far from our model, because there is no common memory and the communication is done through messages.

In most sequential algorithms of many applications, some degree of parallelism exists, it is then the role of the programmer to exploit it in the best way. The problem of a maximum flow is not easy to parallelise because of its complexity and also due to the sequential methods that do not show a straight forward implementation. We have tried to exploit the parallelism inherent in the problem by assigning the processors to the edges and also by dividing some tasks into subtasks. The implementation of the second method which uses BFS showed us a better results than the first one, but still did not realise a great speed up because of many factors which are:

1. Synchronisation : the performance may be lost when the processors require to be periodically coordinated such as passing the same data to all processors for continuing their work.
2. Overhead : it is the big problem and it includes the communication between the transputers which slows the program. As an example, the parallel version for transforming the network into an auxiliary one requires more steps than its serial counterpart because of the overhead. This overhead is the cost of managing the parallelism.
3. Generalisation : through it , performance can be lost also and some tasks are resolved in serial.

All these factors are found in most MIMD machines.

The best problems which should be resolved by this kind of machine are those which do not require a lot of communications.

REFERENCES

[1] A. V. Aho , J. E. Hopcroft , J. D. Ullman , "The design and analysis of computer algorithms" , Addison - Wesley , 1974.

[2] E. R. Arjomandi , D. G. Corneil , "Parallel computation in graph theory" , SIAM. J. COMP. 7 , No. 2, pp 230 - 237, 1978.

[3] S. Baase , "Graph algorithms: Introduction to design and analysis" , Addison - Wesley, 1988.

[4] R. Bornat , "A protocol for generalised Occam", Dept of computer science and statistics , Queen Mary College , London , Oct 1984.

[5] K. C. Bowler et al , "An introduction to Occam 2 and the Meiko surface" , Dept of physics , University of Edinburgh , 1987.

[6] G. Brassard , P. Bratley , "Algorithmics theory and practice" , Prentice - Hall , 1988.

[7] A. Burns , "Programming in Occam 2" , Addison - Wesley, 1988.

[8] R. G. Busacker, T. L. Saaty, "Finite graph and networks :

An introduction with application" Mc Graw-Hill , 1965.

[9] V. Chachra , P. M. Ghare , J. M. Moore , "Applications of graph theory algorithms" , North Holland , 1979.

[10] J. M. Crishlow , "Introduction to distributed and parallel computing" , Prentice - Hall , 1988.

[11] E. A. Dinic , "Algorithm for solution of a problem of maximum flow in a network with power estimation", Soviet Math. Dokl. 11 , No. 5 , pp 1277 - 1280 ,1970.

[12] J. Edmonds , R. M. Karp , " Theoretical improvements in algorithmic efficiency for network flow problems" , J. ACM. 19 , No. 2 , pp 248 - 264 , 1972.

[13] D. M. Eckstein , "Parallel processing using Depth-first search and Breadth-first search" , PhD. thesis , Dept of computer Science , University of Iowa , Iowa city, 1977.

[14] S. Even , "Algorithmic combinatorics" , Collier - Macmillan , 1973.

[15] S. Even , "Graph algorithms" , Pitman 1979.

[16] S. Even , "Parallelism in tape sorting" , CACM 17 , No. 4 , pp 202 - 204 , 1974.

[17] M. J. Flynn , "Some computer organisations and their effectiveness" , IEEE Trans Comp. 21, No. 9 , pp 11 - 23, 1972.

[18] L. R. Ford , D. R. Fulkerson , "Flows in networks" , Princeton Press University , 1962.

[19] Z. Galil , A. Naamad , "An o(EV $\log^2$ V) algorithm for the maximal flow problem" , Journal of Algorithms 21 , pp 203 - 207 , 1980.

[20] F. Gavril , "Merging with parallel processors" , CACM 18 , No. 10 , pp 588 - 591 , 1975.

[21] D. Heller , "A survey of parallel algorithms in numerical linear algebra" , SIAM Rev. 20. No. 4 , pp 740 - 777 , 1978.

[22] D. S. Hirschberg , "Fast parallel sorting algorithms" , CACM 21, No. 8 , pp 657 - 661 , 1978.

[23] D. S. Hirschberg , A. K. Chandra , D. V. Sarwate , "Computing connected components on parallel computers" , CACM 22 , No. 8 , pp 461 - 464 , 1979.

[24] C. A. R. Hoare , "Communicating sequential processes" ,

CACM 21 , No 8 , pp 323 - 334 , 1978.

[25] C. A. R. Hoare , "Occam programming manual", Inmos Ltd, Prentice - Hall , 1984.

[26] R. W. Hockney , C. R. Jesshope , " Parallel computers " , Adam Hilger , 1983.

[27] G. Jones , "Programming in Occam", Prentice - Hall , 1987.

[28] A. V. Karzanov , "Determining the maximal flow in a network by the method of perflows " , Soviet Math. Dokl. 15 pp 434 - 437, 1974.

[29] D. May , R. Taylor , "Occam - an overview", Microprocessors and Microsystems . 8 , No. 2 , March 1984.

[30] D. May , "Preliminary: Occam 2 product", Inmos Ltd, June 1986.

[31] C. H. Papadimitriou , K. Steiglitz , "Combinatorial optimisation algorithms and complexity" , Prentice - Hall , 1982.

[32] D. Pountain , " A tutorial introduction to Occam programming" , Inmos Ltd , August 1986.

[33] F. P. Preparata , "New parallel sorting schemes" , IEEE Trans Comp , c27 , No 7 , pp 669 - 673 , 1978.

[34] P. W. Purdom . Jr , C. A. Brown, "The analysis of algorithms" Holt , Renehart and winston , 1985.

[35] C. Savage , "Parallel algorithms for graph theoretic Problems", PhD. thesis , University of Illinois , Urbana , 1977.

[36] R. Sedgewick , "Algorithms" , Addison - Wesley , 1984.

[37] Y. Shiloach , U. Vishkin , "An o(n^2 log n) Parallel Max Flow Algorithm" , Journal of Algorithms 3 , pp 128 -146 , 1982.

[38] Y. Shiloach , U. Vishkin , "Finding the maximum , merging and merging in a parallel computation model " , Journal of Algorithms. 2, pp 88 - 102 , 1981.

[39] Y. Shiloach , U. Vishkin , "An o(log n) parallel connectivity algorithm" , Journal of Algorithms 3 , pp 57 - 67 , 1982.

[40] "Transputer Development System" , Inmos Ltd , Prentice - Hall 1988.

[41] "Transputer family" , Inmos Ltd ,1986.

[42] "Transputer reference manual" , Inmos Ltd , 1985.

[43] L. G. Valiant , " Parallelism in comparison problems" , SIAM.J. 4 , No 3 , pp 348 - 355 , 1975.

[44] S. H. Wilf , "Algorithms and complexity" , Prentice - Hall, 1981 .