Binary Search Mesh: A Concurrent Data Structure for Hypercube

Mohammad R. Meybodi
Computer Engineering Department
Amirkabir University of Technology

Abstract

In this report, a concurrent data structure for implementing dictionary operations on a distributed-memory message passing multiprocessor with hypercube topology is presented. The implementation is based on the concept of binary search mesh which is introduced for the first time in this paper. Unlike balanced binary search tree, binary search mesh can be mapped into a hypercube in such a way as to preserve the proximity property.

Index Terms: Concurrent Data Structure, Parallel Algorithm, Binary Search Mesh, Hypercube, Multicomputer

1 Introduction

A dictionary is a set of pairs (key,record) and the operations *INSERT* , *DELETE*, *SEARCH*, *MIN*, *MAX*, and *NEAR*. The effect of the various operations are defined as follows. Our notations follow [3]. Let F denote the set of all key-record pairs in the dictionary and (k,r) denote a pair with key k and record r. For a key k and set F, we say k is stored in the dictionay if there exists an r such that (k,r) is in F. Define

$$F(k) = \{(k,r) | (k,r) \in F\};$$

so $F(k)$ is a singleton set if k is in F and empty if k is not in F. The effect of dictionary operations are as follows:

INSERT (k,r):
$F \leftarrow (F - F(k)) \cup \{(k,r)\}$.
Response is null.

DELETE (k):
$F \leftarrow F - F(k)$.
Response is null.

SEARCH(k):
F is not changed.
Response is $F(k)$ if k is in F and "not in" if k is not in F.

MIN:
$F \leftarrow F - F(k_{min})$.
Response is $F(k_{min})$ where k_{min} is the smallest key in F.

MAX:
$F \leftarrow F - F(k_{max})$.
Response is $F(k_{max})$ where k_{max} is the largest key in F.

NEAR(k) :
F is not changed.
Response is $F(k_{near})$ where k_{near} is the stored key closest to k

An insertion operation is *redundant* when the pair requested to be inserted into the dictionary already exists in set F. A deletion operation is redundant when the pair requested to be deleted does not exist in set F. For the simplicity of presentation a pair is represented by its first component. In this report, we assume that operations *INSERT* and *DELETE* are nonredundant and also values stored in the dictionary are unique.

Three kinds of implementations of dictionary have been reported in the literatures: sequential algorithms for implementation on uniprocessor[1], parallel algorithmic architecture for realization in hardware[2-8], and parallel algorithms for implementation on parallel computers [9-14].

Existing algorithms for parallel computer are divided into two groups: those for shared memory multiprocessors and those for distributed memory multicomputers. When designing algorithms for shared memory multiprocessors focus in on reducing the interference between concurrent processes accessing the data structure whereas in the case of distributed memory multiprocessor the focus is on exploiting the parallelism in the data structure. Concurrent algorithms for manipulating binary tree due to Kung and Lehman [20], concurrent algorithms for B-tree due to Lehman and Yao [14], algorithms for current search and insertion of data in AVL-tree and 2-3 trees due to Ellis [15,16], concurrent algorithm for insertion and deletion on the heap due to Rao and Kumar [21], and Biswas and Browne [22] are examples of algorithms for shared-memory multiprocessor. Example of algorithms for distributed memory multiprocessor are:

balanced cube due to Dally [11], sorted chain due to Omondi and Brock[12], and banyan heap due to Meybodi [9]. All three algorithms are for hypercube multicomputer.

In this paper an implementation of dictionary data structure on distributed-memory message passing with hypercube topology is presented. The implementation is based on the concept of binary search mesh which is introduced for the first time in this paper. Unlike balanced binary search tree, binary search mesh can be mapped into a hypercube in such a way as to preserve the proximity property. A binary balanced tree with $2^d - 1$ nodes can not be embedded into a d-cube for $d > 2$ [23].

A hypercube model is a particular example of a distributed- memory message passing parallel computer. In a hypercube of dimension d, there are 2^d processors. Assume that these are labeled 0,1,2,...,2^{d-1}. Two processors i and j are directly connected iff the binary representation of i and j differ in exactly one bit. Each edge of figure 1 represents a direct connection of a 4-d hypercube. Thus in a hypercube of dimension d, each processor is connected to d others, and 2^d processors may be interconnected such that the maximum distance between any two is d.

The rest of this paper is organized as follows. section 2 gives the definition of binary search mesh and the implementation of dictionary operations on this structure. In section 3 the implementatiom of binary search mesh on hypercube is discussed. Concept of consistency is introduced in section 4. The last section is the conclusion.

2 Binary Search Mesh

Definition 1 *d-dimensional Mesh: A d-dimensional mesh is a collection of nodes which are arranged along the points of d-dimensional space and there is an edge between nearest neighbors. The nodes of a d-dimensional mesh with n_i points along the ith dimension are d-tuples $(x_1, x_2, ..., x_d)$ where each of the coordinates x_i, $i = 1, 2, ..., d$, takes an integer value from 1 to n_i. The links are the pairs $((x_1, ..., x_d), (y_1, ..., y_d))$ for which there exist some i such that $|x_i - y_i| = 1$ and $x_j = y_j$ for all $j \neq i$.*

Definition 2 *2−d Half Mesh: An nxn 2-dimensional half mesh (nxn half mesh in short) is obtained by dividing a 2-dimensional nxn mesh along the diagonal $\{(1, n), (2, n-1), (3, n-2), ..., (n, 1)\}$ of the mesh as shown in figure 2. The leftmost top node (node $(1, 1)$) is called the root of the half mesh and the nodes along the diagonal (nodes $(i, n-i+1)$ for $1 \leq i \leq n$) are called leaf nodes. At every level, the number of nodes is one more than at the next higher level. Also the leftmost and rightmost nodes (nodes $(i, 1)$ and $(1, i)$ for $1 \leq i \leq n$), called boundary nodes, have only one parent whereas the innermost nodes called internal nodes have two parent nodes, left and right parent. The root has no parent. Nodes $(i, j+1)$ and $(i+1, j)$ are called the right child and the left child of node (i, j), respectively.*

Having introduced the necessary notations and definitions we define binary search mesh.

Definition 3 *Binary Search Mesh: A binary search mesh is a 2-d half mesh which satisfies the following conditions:*

1. each internal node can hold zero or one data items

2. each leaf node can hold zero or more data items

3. the item stored at node (i, j) is greater than the item stored at node $(i, j+1)$ and smaller than the item stored at node $(i+1, j)$.

From the definition of binary search mesh we can state the following results. $item_{(i,j)}$ is the item stored in processor (i, j).

Lemma 1 *For some i, $(1 \leq i \leq n)$, the sequence of items $\{ item_{(i,1)}, item_{(i,2)}, ..., item_{(i,n-(i-1)} \}$ forms an increasing sequence.*

Lemma 2 *For some j, $(1 \leq j \leq n)$, the sequence of items $\{ item_{(1,j)}, item_{(2,j)}, item_{(3,j)}, ..., item_{(n-(j-1)} \}$ forms a decreasing sequence.*

Lemma 3 *For some i, $(1 \leq i \leq n)$, the sequence of items $\{ item_{(i,1)}, item_{(i,2)}, ...item_{(i,i-1)}, item_{(i,i)}, item_{(i,i+1)}, ..., item_{(2,i)}, item_{(1,i)} \}$ forms an increasing sequence.*

Lemma 4 *For some i and some j, $(1 \leq i, j \leq n)$, the sequence of items $\{ item_{(n-1,j)}, item_{(n-2,j)}, ..., item_{(i,j)}, item_{(i,j+1)}, item_{(i,j+2)}, ..., item_{(i,n-1)} \}$ forms an increasing sequence.*

See figures 3a, 3b, 3c, and 3d for examples.

2.1 Operations

Next, we present the implementation of dictionary operations *INSERT, DELETE, SEARCH, MIN, MAX, NEAR* on the binary search mesh structure.

INSERT(x) operation: *INSERT* operation is similar to its counterpart in binary search tree. Item x is first compared with the value at the root. If it is smaller (greater) than this value, then it will be compared with the value at the leftchild (rightchild) of the root. Item x moves to the right or left (depending on the result of comparison) until it finds an empty or a leaf node. This node becomes the home for x. Item x will be inserted into a leaf node if x, in its journey from the root to the leaf nodes, does not find an empty internal node. The list of items in each leaf node is

kept sorted.

SEARCH(z) operation: *SEARCH* operation starts at the root and move to the right or left depending on the result of the comparison of the target value x with the item at the node. The search stops when either the target value is found or it encounters an empty node.

DELETE(z) operation: An item x is deleted by replacing x by item y found at the last nonempty node which is reached by going all the way to the right from the left child of the node containing x. If a leaf node is reached then the smallest value at this node replaces x. The smallest item in the leaf node is then removed. If the left child of the node containing x is empty, then x is replaced by the item y found at the last nonempty node which is reached by moving down to the right from the node containing x. If the node containing x does have an empty right child then the replacement for it is found by moving down to the left from the node containing x. If left and right child of a node are both empty then x is deleted and then the node is marked empty. If x is stored in a leaf node then it is simply removed from that leaf node. The leaf node becomes an empty node if x is its only item.

MAX and *MAX* operations: The minimum (maximum) item in the dictionary is found by moving down to the left (right) starting at the root. The last nonemty node encountered contains the minimum (maximum) item in the binary search mesh structure. If the nonempty node encountered is a leaf node then the smallest (largest) item in that leaf node is the value returned by *MAX* (*MIN*) operation.

NEAR operation will be discribed later in this report.

3 Implementation

We now discuss the implementation of the binary search mesh on the distributed memory parallel computer with the hypercube topology by first considering a procedure for embedding the half mesh into the hypercube.

The half mesh is embedded into the hypercube in a such a way as to preserve the proximity property, i.e., so that two adjacent nodes in the half mesh are mapped into neighbor processors in the hypercube [16]. The use of Reflected Gray codes is one known technique to obtain a mapping which preserve the proximity property. This technique can be explained as follows: map the mesh point (x_1, x_2) into the hypercube node with number $s_1 s_2$ where s_i is the d_i - bit binary string which is the $x_i th$ element of the d_i - bit reflected gray code [17]. Figure 4 illustrates the mapping of a 4x4 half mesh into a 4 dimensional hypercube. Without loss of generality we assume the $2^d = nxn$.

The nodes in the hypercube which are not part of the binary search mesh form another half mesh called mirrored half mesh. The mirrored half mesh can be used to store another data structure or combines the result generated by the leaf nodes. These posibilities will be explored in another report.

The process running on each processor in the hypercube consists of two distinct parts. The first part belongs to the application that runs on all the processors of the hypercube. The second part, called executive, is responsible for the execution of the dictionary operations. An executive can receive and process many messages simultaneously. Dictionary operations are initiated by the application part of the processes running on the processors of the hypercube. An operation issued by a process is communicated to the executive of that process. The executive then sends that operation to the root processor for execution. The dictionary operations received by the root processor will be executed in a pipelined fashion. The response to an operation is forwarded to the processor which originally initiated the operation. An executive receives instructions either from another executive or from the application part of the its own processor.

We use the following syntax and semantic for *SEND* and *RECEIVE* instructions. The instruction *SEND*(<processor>, '<instruction>') sends instruction <instruction> to processor <processor> for execution. The execution of *RECEIVE*(<processor>, <information>) causes the information specified by the second argument to be obtained from processor <processor> and forwarded to the requesting processor (the processor executing the receive instruction). Both *SEND* and *RECEIVE* instructions are blocking. If <processor> is *anyprocessor* then *RECEIVE* instruction will be completed if <information> is received from any processor in the hypercube.

The application process of processor p issues a dictionary operation by issuing *SEND*(root, <instruction>) where <instruction> is one of the dictionary operations. It then issues *RECEIVE* (<anyprocessor> <instruction-done>) and waits for the instruction <instruction> to complete. <instruction-done> is one of the *insert-done*, *search-done*, *delete-done*, *sort-done*, *near-done*, *max-found*, and *min- found* and <anyprocessor> denotes any processor in the hypercube

The following formats are used for dictionary operations. P is the processor which has initiated the operation.

INSERT(p,x):
where x is the item to be inserted.

SEARCH(p, x):
where x is the item to be searched for.

DELETE(p,x,tag):

where x is the item to be deleted. The variable tag can take two values red and green. Tag is initially set to red.

NEAR(p,x):
where x is the item whose nearest item will be computed.

The root processor performs the following codes upon receiving an instruction *inst* initiated at processor q. L_p, R_p, $L_{p_{parent}}$, and $R_{p_{parent}}$ are used to refer to the left child, right child, left parent, and the right parent of processor p, respectively. Item $item_k$ is the value stored at processor k.

```
If inst = INSERT (p,x) then
  begin
    if item_root = null then
      begin
        item_root = x;
        SEND(p, insert − done(x)
      end
    else
      if item_root > x then SEND(L_root, 'INSERT(q,x))
      else SEND(R_root, INSERT(p,x))

if inst = SEARCH(p,x) then
  if item_root = x then
    SEND(p, search − succeed(x))
  else
    if item_q < x then SEND(L_root, SEARCH(p,x))
    else SEND(R_root, SEARCH(p,x))

if inst = DELETE(p,x,tag) then
  if item_q = x then
    begin
      if tag = red then
        if the left child of root is empty then
          SEND(R_root, move − right(p))
        else
          begin
            tag = green;
            SEND(L_root, DELETE( p,x,tag))
          end
      else
          if the right child is not empty then
            SEND(R_root, move − right(p))
          else
            SEND(root, move − up(item_root))
    end
  else
    if item_q < x then
    SEND(R_root, DELETE(p,x,tag))
  else
    SEND(L_root, DELETE(p,x,tag))

if inst = MIN(p) then
  if the left child of root is empty then
    SEND(p, min − found(x)
  else
    SEND(L_root, MIN(p))

if inst = MAX(p) then
  if the right child of root is empty then
    SEND(p, max − found(x))
  else
    SEND(R_root, MAX(p))
```

Upon receiving an instruction *inst* processor q the following codes will be executed. Instruction *inst* is originally initiated at processor p.

```
if inst =INSERT(p,x) then
  if q is a leaf node then
    begin
      insert x into the leaf node q;
      SEND(p, insert − done(p,x))
    end
  else
    begin
      if item_q = null then
        begin
          item_q = x
          SEND(p, insert − done(p,x))
        end
      else
        if item_q > x then SEND(L_q, INSERT(p,x))
        else SEND(R_q, INSERT(p,x))
    end

if inst = SEARCH(p,x) then
  begin
    if q is a leaf node then
      begin
        if x is in the list of items in processor q then
          SEND(p, search − success(p,x))
        else
          SEND(p, search − fail(p,x))
    else
      if item_q = x then SEND(p, search − success(p,x))
      else if item_q < x then SEND(L_q, SEARCH(p,x))
      else SEND(R_q, SEARCH(p,x))

if inst = DELETE(p,x,tag) then
  if item_q = x then
    begin
      if tag = red then
        if the left child of q is empty then
          SEND(R_q, move − right(p))
        else
          begin
            tag = green;
            SEND(L_q, DELETE( p,x,tag))
          end
      else
          if the right child of q is not empty then
            SEND(R_q, move − right(p))
```

```
        else
          SEND(R_{q_parent}, move − up(item_q))
    end
  else if item_q < x then
    SEND(R_q, DELETE(p,x,tag))
  else SEND(L_q, DELETE(p,x,tag))

if inst = MIN(p) then
  if q is a leaf node then
    begin
      find the smallest value y in the leaf node q;
      SEND(p, min − found(p,y))
    end
  else
    if the left child of q is empty then
      SEND(p, min − found(p, item_q))
    else
    SEND(L_q, MIN(p))

if inst = MAX(p) then
  if q is a leaf node then
    begin
      find the largest value y in the leaf node q;
      SEND(p, max − found(p,y))
    end
  else
    if the right child of q is empty then
      SEND(p, max − found(p,item_q))
  else
    SEND(R_q, MAX(p)).
```

Operations *move-right*, *move-left*, *move-up*, and *NEAR* are described below.

```
if inst = move − right(p) then
  begin
    if q is a leaf node then
      begin
        find the smallest value x in leaf node q;
        SEND(L_{q_parent}, move − up(x));
        delete x from leaf node q
    else if item_q ≠ null then
      begin
        SEND(L_{q_parent}, move − up(item_q))
        SEND(R_q, move − right(p))
      end
    else

if inst = move − left(p) then
  if q is a leaf node then
    begin
      find the smallest value x in leaf node q;
      SEND(R_{q_parent}, move − up(x));
      delete x from leaf node q
    end
  else
    begin
      SEND(R_{q_parent}, move − up(item_q))
      SEND(L_q, move − left(p))
    end

if inst = move − up(x) then
  item_q := x
```

NEAR(x) operation: To find the nearest item to x, the list of items in the binary search mesh is first sorted. There are number of parallel sorting algorithms reported in the literature that can be used to implement the sort operation. Baudet and stevenson's parallel sorting algorithm is one of them [19]. To use this algorithm the items in each row of the half mesh are first brought into the first processor of that row. The algorithm, after sorting, will leave the sorted sequence in the processors of the first row of the half mesh. *NEAR* operation first finds the processor which contains x. The nearest item to x will be either in the processor containing x or in the processor immediately to the left or to the right of the processor containing x.

Remark 1 *Using lemmas 1, 2, 3, and 4, a different set of algorithms for dictionary operations can be developed. These algorithms will be discussed in another report.*

From the the properties of mesh and the above algorithms, we can state the following theorems.

Theorem 1 *Operations DELETE and INSERT preserve the binary search mesh property.*

Theorem 2 *On a nxn binary search mesh operations* MIN, MAX, SEARCH, INSERT, *and* DELETE *each require $O(n)$ time to complete.*

Theorem 3 *Operation* INSERT, MIN, MAX, *and* DELETE *each needs no more than $O(n)$. communications.*

Proofs of the above theorems can be found in [22].

Theorem 4 *Implementation of an nxn binary search mesh on a hypercube offers $O(n)$ throughput.*

Proof: The implementation can perform $n(n+1)/2$ operation at a time and each operation requires $O(n)$ time, and hence $O(n)$ throughput.

4 Consistent Concurrent Data structures

In what follows we first define a few terms and introduce the concept of consistency for concurrent data structure.

Definition 4 *The timestamp of an operation is the time at which the operation is initiated at a node and the timestamp of a communication is the timestamp of the operation which generated that communication.*

Definition 5 *Operations $O_1, O_2, ...,$ and O_q with timestamps $t_1, t_2, t_3, ..., t_q$ are said to be a sequence of operations if $t_1 < t_2 < ... < t_q$ and operations O_i, $1 < i \leq$ are the only operations issued between t_1 and t_q.*

Definition 6 *A concurrent data structure is said to be consistent if the concurrent execution of any sequence of operations on the data structure gives the same result as executing the operations sequentially.*

Theorem 5 *Implementation of binary search mesh on the hypercube is not consistent.*

Proof: Consider two operations *INSERT* with timestamp t_1 and *MAX* with timestamp t_2 such that $t_1 < t_2$. It is quite possible that *MAX* operation be received by the root processor before the *INSERT* operation and as a result it finishes first.

Theorem 6 *The implementation of binary search mesh is consistent if the root processor does not start an operation unless all the operations with the lower timestamps have been completed.*

Below we describe a procedure for making the implementation of binary search mesh consistent.

Before a processor forwards an operation to the root processor for execution, it first broadcasts the operation together with its timestamp to all the other processors in the hypercube and then waits to receive acknowledgemnets from those processors. A processor will not acknowledge an operation unless all its operations with lower timestamp have been completed. Once all the acknowledgements have been received for a particular operation, it will be send to the root processor for execution.

5 Conclusion

We have proposed a concurrent data structure, called binary search mesh for implementing dictionary on a distributed-memory message passing multiprocessor with hypercube topology. The structure can be used by any application running on the hypercube without worrying about all the necessary communications and synchronizations. For a $n \times n$ binary search mesh dictionary operations require $O(n)$ time to complete, but since $n(n+1)$ operations may be initiated simultaneously at different processor, $O(n)$ throughput is achievable as compared to $O(1)$ throughput for sequential data structure. Binary search mesh, unlike balanced binary tree can be mapped into a hypercube in such a way as to preserve the proximity property.

6 References

1. A. V. Aho, J.E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithm, Addison Wesley, 1974.
2. J. L. Bentley and H. T. Kung, "A tree Machine for Searching Problems," Proceeding of the International Conf. on Parallel Processing, August 1979.
3. T. A. Ottmann, A. L. Rosenberg, and L.J. Stockmeyer, "A Dictionary Machine for VLSI," IEEE Transaction on Computers, vol. c-31, No. 9, Sept. 1982, pp. 892-897.
4. A. K. Somani and V. K. Agarwal, "An Unsorted VLSI Dictionary Machine," Proceedings of 1983 Canadian VLSI Conference, University of Waterloo, Waterloo.
5. A. K. Somani and V. K. Agarwal, "An Efficient VLSI Dictionary Machine, " 11th Annual International Symposium on Computer Architecture, University Of Michigan, Ann Arbor, Michigan, pp. 142-150.
6. M. J. Atallah and S. R. Kosaraju, "A Generalized Dictionary Machine for VLSI," IEEE Transactions on Computers, Vol C-34, No. 2, Feb. 1985, pp. 151-155.
7. H. Schmeck and H. Schroder, "Dictionary Machines for Different Models of VLSI," IEEE transaction on computers, vol. C- 34, No. 5, May 1985, pp. 472-475.
8. A. L. Fisher, "Dictionary Machines with Small Number of Processors," Proc. Int. Symp. on Computer Architectures, Ann Arbor, June 1984, pp. 151-156.
9. M. R. Meybodi, "Concurrent Data Structures for Hypercube Machine," Computer Science Technical Report, Ohio University, Athens, Ohio, 45701.
10. M. R. Meybodi , "Implementing Priority Queues on Hypercube Machine," Annual Parallel Processing Symposium, Fullerton, California, April 1990, pp. 85-111.
11. W. J. Dally, A VLSI Architecture for Concurrent Data structures, Kluwer Academic Publishers, 1987.
12. A. R. Omondi and J. D. Brock, "Implementing a Dictionary on Hypercube Machine," Proceedings of International Conference on Parallel Processing, August 1987, pp. 707-709.
13. H.T. Kung and P.L. Lehman, "Concurrent Manipulation of Binary Search Trees," ACM Transactions on Databases Systems, Vol. 5, No. 3, Sept. 1980, pp. 354-382.

14. P.L. Lehman and S.B. Yao, "Efficient Locking for Concurrent Operations on B-Trees," ACM Transactions on Databases Systems, Vol. 6, No. 4, Dec. 1981, pp. 650-670.

15. C.S. Ellis, "Concurrent Search and Insertion in 2-3 Trees," Acta Information, Vol. 14, 1980, pp. 63-86.

16. C.S. Ellis, "Concurrent Search and Insertion in AVL Trees," IEEE Transactions on Computers, Vol. C-29, No. 9, Sept. 1980, pp. 811-817.

17. Y. Saad and M. Schultz, "Topological Properties of hypercubes," IEEE Transactions of Computers, Vol. 37, No. 7, July 1988, pp. 867-872.

18. J. E. Brandenburg and D. S. Scott, "Emdedding of Communication Tree and Grid into Hypercube," iPSC technical Report, August 1986.

19. S. Lakshmivarahan, S. K. Dhall, and L. L. Miller, "Parallel Sorting Algorithms," Advances in Computers, Vol 23, 1984, pp. 295-354.

20. V. N. Rao and V. Kumar, "Concurrent Access of Priority Queue," IEEE Transactions on Computers, Vol. 37, No. 12, Dec. 1988, pp. 1657-1665.

21. J. Biswas and J. C. Browne, "Simultaneous Update of Priority Structures," Proceedings of International Conference on Parallel Processing, August 1987, pp. 124-131.

22. M. R. Meybodi, "Binary Search Mesh: A Concurrent Data Structure for Hypercube," Technical Report 1M92, Computer Science Department, Ohio University, Athens, Ohio.

23. D. P. Bertsekas and J. N. Tsitsiklis, Parallel and Distributed Computation, Prentic Hall, 1989.

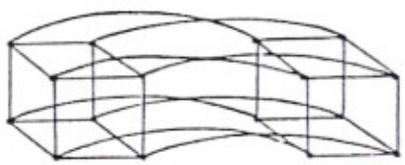

Figure 1

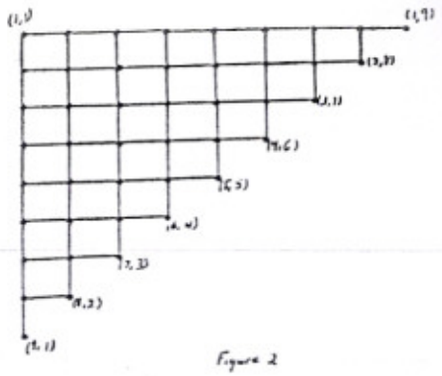

Figure 2

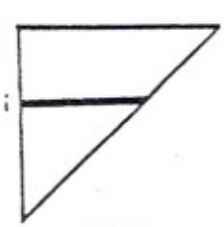

Figure 3a

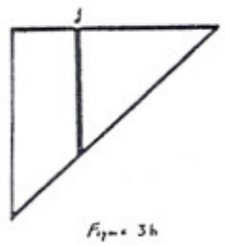

Figure 3b

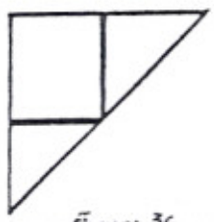

Figure 3c

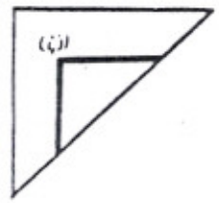

Figure 3d

Figure 4