This paper focuses on how to program parallel machines at a very high level — even higher than conventional sequential programming — by using sets, and how sets are supported among a large scale of cooperative, distributed, parallel processes. The example program — finding connected components of an undirected graph — illustrates how an abstract algorithm description can be a parallel program. In many recently discovered parallel algorithms, clauses — such as (1) some predicate `z` applied to all elements `q` of a set `S`, (2) function calls `y(q)` for those elements `q in S` satisfy some predicate `z(p)`, (3) an operator (min, max, summation, etc.) applied to a set of function values `y(p), p in S` — frequently appear. ***Aggregate set operation*** is a way of programming abstraction for such tests and evaluations of predicates and functions on a set of arguments. Collectively, these operations are treated as a single entity. An implementation of aggregate set operations on an abstract "cross-bar array" is given, and followed by concrete implementations of these operations on various multi-processor networks.

Very-high-level Parallel Programming by Aggregate Set Operations

Marina C. Chen

Research Report YALEU/DCS/RR-499
October 1986

Work supported in part by the Office of Naval Research under Contract No. N00014-86-K-0296.
Approved for public release: distribution is unlimited.

Contents

Very-high-level Parallel Programming by Aggregate Set Operations

(Extended abstract)

Marina Chen

Department of Computer Science, Yale University
New Haven, CT 06520-2158
chen-marina@yale

Abstract

This paper focuses on how to program parallel machines at a very high level — even higher than conventional sequential programming — by using sets, and how sets are supported among a large scale of cooperative, distributed, parallel processes. The example program — finding connected components of an undirected graph — illustrates how an abstract algorithm description can be a parallel program. In many recently discovered parallel algorithms, clauses — such as (1) some predicate `z` applied to all elements `q` of a set `S`, (2) function calls `y(q)` for those elements `q in S` satisfy some predicate `z(p)`, (3) an operator (min, max, summation, etc.) applied to a set of function values `y(p), p in S` — frequently appear. *Aggregate set operation* is a way of programming abstraction for such tests and evaluations of predicates and functions on a set of arguments. Collectively, these operations are treated as a single entity. An implementation of aggregate set operations on an abstract "cross-bar array" is given, and followed by concrete implementations of these operations on various multi-processor networks.

1 Introduction

One of the most critical problems in parallel processing today is that of programming parallel machines. The difficulty lies in the task decomposition: how to partition a given task into pieces, one for each processor, so that it can be accomplished by the cooperation of many processors in parallel. There have been two main approaches: (1) programming in a conventional sequential language, and relying on a parallelizing compiler to generate code for parallel machines (as in the area of numerical computing) or relying on a parallelizing interpreter and run-time supports for dynamically spawning parallel processes (as in the area of functional programming); and (2) devising a parallel programming language and expressing parallelism explicitly in a program.

The first approach has the advantage that programs can be written in familiar languages and old programs can be transformed by parallelizing compilers for executing on the new machines. However, the parallelism discovered this way is limited by the algorithm embodied by the program. It is unlikely that the transformations provided by the parallelizing compilers are sophisticated enough for the task of redesigning programs better suited for parallel processing. Such redesigning is necessary for utilizing parallel resources in an optimal way. Take the problem of sorting as an example. Consider parallelizing a Quick-sort program, which is a very good sequential solution. Since parallelism can be gained by the parallel evaluations of independent sub-tasks, the parallelizing strategy would be spawning a process for each of the two recursive calls to quick-sort at the level below. The time complexity of the parallelized program indeed is improved from $O(n \log n)$ of the sequential version to $O(n)$ (since $O(n)$ comparisons are needed at the top level and the number of comparisons is halved every one level downwards) by using $O(n)$ independent parallel processes. However, what can be achieved by various parallel sorting networks (see for example, [11]) with $O(n)$ processors is $O(\log^2 n)$, which is significantly faster for large problem size n. Numerous other problems have the same property that their sequential solutions do not lend themselves to efficient parallel implementations, as exemplified

by many of the newly devised parallel algroithms [8] which are considerably different from their sequential counterparts.

This point leads to the second approach — parallel programming, where parallelism is explicitly expressed in a program. The flexibility of explicitly programming for parallelism allows this approach to be applied to either class of parallel machines (shared-memory machines or message-passing machines) as well as any kind of parallel algorithms. But, parallel programming and debugging can be extremely difficult due to the complexity that might arise from the interaction of hundreds of thousands of simultanenously executing processes. Most of the parallel languages, either proposed or in use, have explicit constructs for parallelism. Programmers specify how taskes should be partitioned and which ones can be run in parallel (such as using the "future" construct in Multilisp [4]); or they specify how processes are mapped to processors (such as using annotation in ParAlfl [7]); or they specify explicitly the communications between processes and the point at which a communication should take place (as in CSP [5]). Such programming activity requires a programmer to keep track of both the processor's own state and its interactions with other processes, which may grow combinatorially complex if care is not taken.

A critical research question raised here is: can a parallel program be written in a highly abstract form such that the detailed interactions among processes in space and time are suppressed, and yet the parallelism is explicit enough for generating efficient code for an assemblage of concurrent, communicating processes? In this paper, we show how these two seemingly conflicting goals of *ease of programming* and *efficient target code* can be achieved through a very-high-level parallel programming language **Crystal** [2] and a set of communication-efficient distributed data structures. Since the difficulty of explicit parallel programming seems to become more pronounced as the processor number of a machine grows, we therefore consider, in the following, machines equipped with an enormous number of processors. Due to the sheer number of processors, such machines must fall in the class of network of processors with localized memory, i.e., message-passing machines. The term ultra-parallelism is used for the kind of parallelism supported by machines of this scale, as opposed to the parallelism that can be obtained from parallel machines of a much smaller scale, say 32 processor machines.

The goal of ease of programming is achieved by providing **Crystal**, in which source programs look like concise, formal mathematical descriptions. High level data structures such as sets, vectors, matrices, and multi-dimensional arrays are provided. They are called distributed data structures because, potentially, each element of a data structure might be mapped to an active, autonomous process. The gap between high-level language constructs and efficient parallel implemenations is bridged by three classes of techniques: (1) a set of program-synthesis methods for generating efficient parallel programs, (2) communication-efficient operations supported on the distributed data structures, and (3) an effective run-time processes management and load balancing mechanism. The synthesis methodology for deriving parallel programs — in particular, for applications in scientific computing, and communication-efficient operations on vectors and arrays — has been presented elsewhere [1,3]. The issue of run-time process and load balancing management is a topic of its own, and will not be presented here.

This paper focuses on how to program parallel machines at a very high level — even higher than conventional sequential programming — by using sets, and how sets are supported among cooperative, distributed processes. Examples are drawn from the area of symbolic computing where data structures other than vectors and arrays are strongly favored.

In the following, first a description for finding connected components of a graph is given as an example of a **Crystal** program. We show how this seemingly mathematical description can be interpreted as a parallel program. Next, aggregate set operations are introduced. An implementation of these operations on an abstract "cross-bar array" is introduced. Last, concrete implementation of these operations on multi-processor networks, in particular on hypercube machines, is illustrated.

2 Algorithm-descriptions as Programs

Due to the page limitation, formal syntax and semantics of **Crystal** can not be included here. An example program should suffice to illustrate the level of description of the language. Figure 1 shows a working **Crystal** program which finds the connected components of a graph [11]. Here a particular input data set is included in the program in place of I/O commands for illustration purpose. Comments to the program are prefixed by "!" at every line. The program body consists of a system of mutually recursive equations, a familar construct in many functional or applicative programming languages. Set notation is used as in conventional mathematical notations. The notation `\binary_op` turns an associative binary operator into an n-ary operator on a set of n elements, such as `\min` appeared in the program.

The program reads: The demanded output of the program are the connected **components** of a graph, where the graph consists of `n` vertices, in this example, 16 vertices. The identifier `log_n` is a constant, in this case 4, which is the ceiling of the base-2 logarithm of `n`. The set of vertices is called **vset**. The edges of the graph is given by a set `E`.

The result **components** is defined as a set of **partial_component** at stage `log_n` for those vertices `v` in the vertex set **vset** such that their component identifier **comp** at stage `log_n` equals `v`. The **partial_component** of a vertex `v` at stage `i` is defined as the set of all vertices `u` such that their component identifier **comp** at stage `i` equals `v`. The component identifier **comp** of a vertex `v` is defined initially at stage 0 as the vertex `v` itself. For any later stage, it is defined, in turn, by the function **next**. For each vertex `v` at each stage `i`, **next** is defined initially as `min_comp(v,i)`, and then by a double recursion. The minimum component identifier `min_comp` of a vertex `v` at stage `i` is defined as the minimum value of the set which is the union of a singleton set and the set of all values `min_nbr` for those vertices `u` at stage `i` such that `comp(u, i-1) = v`. Finally, the minimum neighboring component identifier `min_nbr` of a vertex `v` at stage `i` is defined as the minimum value of the set of values `comp(u,i-1)` over all vertices `u` such that the edge `{u,v}` appears in the edge set `E` of the graph.

Two key points should help to convince readers that this program does indeed correctly find connected components of a graph. First, the transitive closure of the relation `u = min_comp(v,i)` on two vertices `u` and `v` can be obtained in `log_n` steps. To see this, first we must note that this relation is not cyclic. If it were, then there should result a contradiction that `min_comp(v, i) < min_comp(v, i)` for some vertex `v`. Since an acyclic chain cannot be longer than length `n-1` for a graph of `n` vertices, and since the chain is shortened by half in each iteration `j`, a total of `log_n` steps suffice to obtain its transitive closure. The second point to prove is that **components** can be obtained from **partial_component** in `log_n` stages. At each stage, every component merges with at least one other component if such connection exisits. Since the number of components is `n` initially, in at most `log_n` steps, all connected partial components are merged into a single component.

So much about the algorithm. It is promised that such a high-level algorithm description is, in fact, a parallel program — but where is the parallelism?

3 Discovering Latent Parallelism

Instead of augmenting conventional languages with explicit language constructs for parallelism, **Crystal** provides conventional mathematical notations and languages, in their foundation, an alternative computational model from the sequential one, so as to reveal the latent parallelism in a **Crystal** program. In other words, built-in parallelism is interpreted from familiar notations and languages in which no explicit commands for communication appear. In the following, such interpretation of programs is illustrated for the example program.

3.1 Processes

For every *argument tuple* (*arg-tuple* for short) in a program, such as `(v,i)` and `(v,i,j)`, there corresponds a process. The set of all arg-tuples appearing in a program corresponds to the set of (parallel) processes that are going to appear during program execution. The ensemble of parallel processes will be implemented by some network of processors. More than one process may be mapped to any given processor. Such concrete implementation issues are discussed in Section 6.

When an arg-tuple appears on the right-hand side of an equation, the corresponding process is called an RHS process. A left-hand-side (LHS) process is defined similarly. For instance, in Equation D4, which defines `min_comp`, `(v,i)` is the LHS process while `(v, i-1)`, and `(u, i-1)` such that `u in vset` and `comp(u, i-1) = v`, are RHS processes.

3.2 Inputs and outputs of a process

Any function call with a given arg-tuple as its argument corresponds to an output produced by the process corresponding to that arg-tuple; for instance, `min_comp(v, i)` is an output produced by process `(v,i)`.

Any function call is an input to the LHS process if it appears on the RHS of the same equation; for instance, `comp(v, i-1)` is an input to process `(v,i)` due to Equation D4.

3.3 Process code and communications

The code a LHS process supposes to execute is defined by the right-hand side of the equation with the aforementioned RHS function calls as arguments. For instance, in Equation D5, the code for computing `min_comp` in process `(v,i)` takes the first input (`comp(v,i-1)`) and makes a set out of it, takes the other inputs and forms a set, performs a set union on both, and then takes the minimum of the new set.

Since each process needs to obtain its inputs in order to execute its code, a communication from each of the RHS processes that supplies an input is necessary. In Equation D6 of the example, a communication from each of the processes u such that `{u,v} in E` is necessary.

The above description of the code and communications sounds as if all communications needed by a process are performed first, and then the process starts its local execution of the code. In the actual implementation of the process code and communications, in particular with non-shared memory multiprocessor networks, the communications often must be routed through intermediate processes as well as have the code distributed among these processes. The discussion of such implementation will be delayed until later.

3.4 Data dependency and parallel evaluation

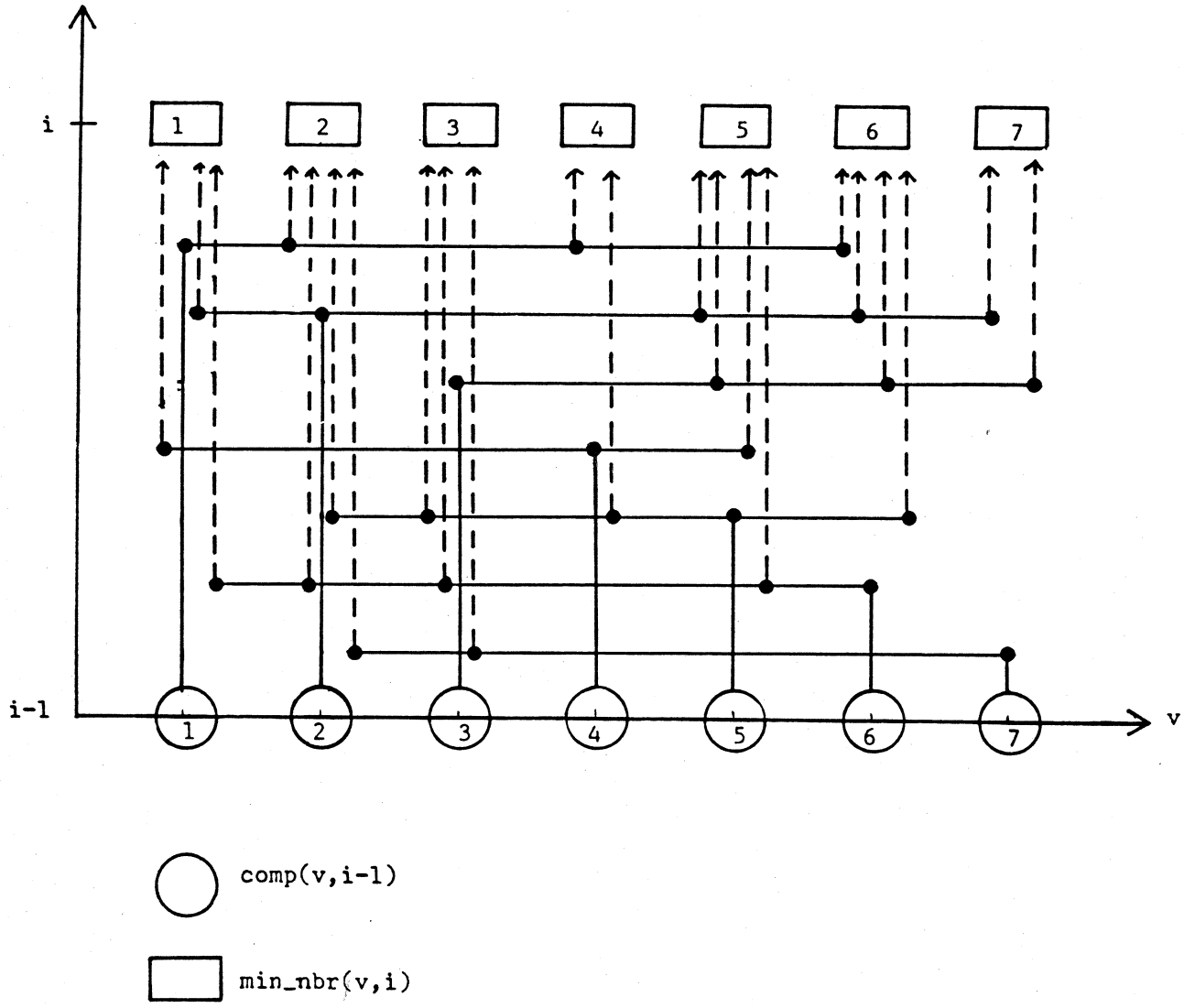
In the following, a model for the execution of a program is described by the aid of a Directed Acyclic program Graph (DAG). The ensemble of processes and its data flow can depicted by a DAG where a portion of it is shown in Figure 2. It consists of nodes, where each node corresponds to an output `x(p)` of a process `p` (or a function call on an arg-tuple), and directed edges between the nodes. One can think of each node as the code segment of process `p` for computing its output `x(p)`. A directed edge goes from a node corresponding to `x(p)` of process `p` to the node corresponding to `y(q)` of process `q` if `y(q)` is on the left-hand side of an equation while `x(p)` is on the right-hand side of the same equation, i.e., `x(p)` is the input to process `q` for computing its output `y(q)`.

The directed edges of the DAG define the data dependency relation of the program. We say that a node **u** immediately precedes **v** ($\mathbf{u} \prec \mathbf{v}$), or **v** immediately depends on **u** ($\mathbf{v} \succ \mathbf{u}$), if there is a directed edge from **u** to **v**. The transitive closure "$\overset{*}{\prec}$" of this relation, called "precede", is

Figure 1: A **Crystal** program for finding the connected components of an un-directed graph.

```
Aug 15 09:32 1986  //astra/yale/chen/temp/cc.cr Page 1


! Connected Components
! Algorithm in Ullman, Computational Aspects of VLSI, pp. 160-164.

!   comp(v, i)       the component of vertex v at stage i, which we take
!                    to be the smallest index of any vertex in that component.
!
!   min_nbr(v, i)    The minimum value of comp(u, i) over all nodes u adjacent
!                     to node v.
!
!   min_comp(v, i)   The minimum value of comp(u, i) over all nodes u
!                    adjacent to component v at stage i, including vertex v
!                    itself.
!
!   next(v,i,log_n) The transitive closure of min_comp(v, i):
!                    the minimum value of comp(u, i) over all components
!                    reachable from v at stage i.
!                    We know this is computable in at most log(n) steps,
!                    hence value of the third index j.

! OUTPUT REQUEST
components where (

! INPUT DATA

    n = 16

    log_n = 4

! vertex Set
    vset = 1:n

! edge Set
    E =  {[1, 2], [1, 6], [1, 4],
          [2, 6], [2, 5], [2, 7],
          [3, 7], [3, 5], [3, 6],
          [4, 5], [5, 6],

          [8, 9], [8, 10], [9, 10],

          [11, 12], [11, 16], [11, 14],
          [12, 16], [13, 16], [13, 15],
          [14, 15], [15, 16]}

! PROGRAM DEFINITION
! D1:
      components =
        [partial_component(v, log_n) | v in vset, comp(v, log_n) = v]

! D2:
      partial_component(v, i) =
        [u | u in vset, comp(u, i) = v]

! D3:
      comp(v, i) =
        << i = 0 -> v,
           i > 0 -> next(v, i, log_n) >>
```

```
Aug 15 09:32 1986  //astra/yale/chen/temp/cc.cr Page 2


! D4:
      next(v, i, j) =
        << j = 0 -> min_comp(v, i),
           j > 0 -> next(next(v, i, j-1), i, j-1) >>

! D5:
      min_comp(v, i) =
        \min ([comp(v, i-1)] union
              [min_nbr(u, i) | u in vset, comp(u, i-1) = v])

! D6:
      min_nbr(v, i) =
        \min [comp(u, i-1) | u in vset, [v, u] in E]

)

! RESULT: [[16, 15, 14, 13, 12, 11], [10, 9, 8], [7, 6, 5, 4, 3, 2, 1]]
```



comp(v,i-1)

min_nbr(v,i)

Figure 2: The DAG describing the data dependency of Equation D6 for the subgraph containing vertices 1 to 7 of the example graph for the connected component program. Note that due to the large number of edges appearing in the diagram, the edges are drawn in sections and connected by black dots as shown.

a partial order. We are interested only in those programs whose DAG has the property that there is no infinite decreasing chain from any node v, nor is there an infinite number of nodes that immediately precede v. Those nodes that have no incoming edges are called *sources.*

The computation of the program proceeds in the familiar data-driven fashion. It starts at the source nodes which are inputs to processes which are set up initially. It is followed by the "firing" of other nodes, or the execution of code which computes the outputs of processes. Each of such executions starts when all of its required inputs, or dependent data become available.

Except for programs of very simple nature, the DAG of a program is often quite complex, as can be seen from the example in Figure 2 in which only a very small portion of an entire DAG is shown. It is obvious that data dependency of such complexity will result in costly communications with high degrees of data congestions and delays for ultra-parallel machines which are supposed to have the highest potential in the future. Such complexity must be managed in order for such machines to be effective. In the following, it is shown that the complexity of inter-process communicatons can be harnessed by efficient communication schemes. As a result, the proposed very-high-level programming can be realistically supported on ultra-parallel machines, and therefore contributes to realizing the potential power of ultra-parallel computing.

4 Aggregate Set Operations

In sequential programming, operations applied to data structures tend to be incremental, such as "head" and "tail" on a list, "push" and "pop" on a stack, or depth-first search on a graph, which only affects one element of the entire structure at a time. For parallel programming, on the contrary, operations applied to a given data structure tend to involve many elements of the structure at once. In parallel programming (for efficient parallel algorithms), descriptions like "The value x of an element p is defined as the minimum (or maximum, summation, union, user_defined_op, etc.) of the set of all values y(q) for those elements p in set S such that the predicate z(p,q) is true," appear very often. In **Crystal**, the above sentence translates to an equation of the form

```
x(p) = \min { y(q) | q in S, z(p,q)} or x(p)= Φ(y,q,S,z(p,q)).
```
(4.1)

For example, Equation D6 in Figure 1 can be written as

```
min_nbr(v,i)=Φ(comp, (u,i-1), vset, u,v in E).
```

The right-hand side of such an equation can be broken up into several operations on elements of a set:

- some predicate z applied to all elements q of set S with respect to p;
- function calls y(q) for those elements q in S satisfy the above predicate are evaluated;
- an operator (min, max, summation, etc.) is applied to a new set formed by the values y(p) obtained above.

In each of the above, the same test of predicates or evaluation of functions are uniformly applied to many elements of a given set. As a way of programming abstraction, it makes sense to treat these tests or evaluations collectively as a single entity rather than each of them individually. Such an entity is called an *aggregate set operation.*

An aggregate set operation which only involves a primitive function or predicates, or a function or predicate defined by other equations on a finite set of elements, is called *a simple aggregate operation.* Function y, predicate z, or the set S in the above equation can have a

more complex structure than the above simple form; in this case, the term *complex aggregate set operation* is used. For instance, function `comp` of Equation `D6` is defined in terms of function `next`, which, in turn, is defined as a double recursion by Equation `D4`. Each set element may have a structure of its own, and predicate `z` may test on any part of it. Moreover, the set elements can be dynamically generated, or the set might even be infinite.

A single aggregate operation may therefore involve functional evaluation of a quite complex structure, and thus is capable of a high degree of descriptive power. Note that in the connected component example, for each stage `i`, the program can be described entirely in terms of aggregate set operations. This is also the case for many other graph algorithms [6], [9], and other combinatorial parallel algorithms. In the case of an infinite set, the implementation described below become impractical. Such a problem falls into the class of problems for which "demand" must be propagated at the run-time and process and load balancing management are necessary.

In the following, the implementation of aggregate set operations is first described in terms of an abstract network, and then followed by concrete implementations.

5 Abstract Networks and Communication Schemes

5.1 A cross-bar array

One type of abstract networks supporting the high-level aggregate set operations is a cross-bar array of n by n cells for a set S of n elements, as shown in Figure 3. Let each cell of the array be indexed by (v, u), and the n diagonal cells of the array be the *home cells* of the set of elements, i.e., (v, v) is the home cell of element `v` of set S. In the following, we use the convention that identifiers of the source program are in typewriter font, whereas identifiers for its cross-bar array implementation are in italics.

5.2 Supporting simple aggregate set operations

We first look at the case when Equation (4.1) requires only simple aggregate set operations.

To be concrete, again the connected component program is used as an example. For this program, an `n` by `n` cross-bar array is needed, where `n` is the number of elements in `vset`. Since the DAG of any program must define a well-founded ordering on the output of processes (or function calls on arg-tuples), induction principle holds on the structure of the DAG. In the description of the implementation, we take advantage of this principle.

Suppose the values `comp(u,i-1)` for some `i>0` on the right-hand side of Equation `D6` are now available, and stored at the home cells (u, u) of the cross-bar array. The implementation for computing the left-hand side value `min_nbr(v,i)` of Equation `D6` can be described as follows:

Procedure A:

1. Each cell has a boolean variable *conn*. The variable $conn(v, u)$ is set to 1 if `{v,u} in E`; otherwise it is set to 0.

2. Broadcast the value `comp(u,i-1)` from each home cell (u, u) vertically to all cells in each column, and call it $num(v, u)$

3. For those cells (v, u) such that $conn(v, u) = 1$, let $nbr\_num(v, u) = num(v, u)$. Otherwise, set the variable $nbr\_num(v, u)$ to infinity, the identity for the forthcoming operation `min`.

4. For each row v of the cross-bar array, apply the operator `min` to all n values $nbr\_num(v, u)$ on that row. Then send the result to home cell (v, v) which is `min_nbr(v,i)`.

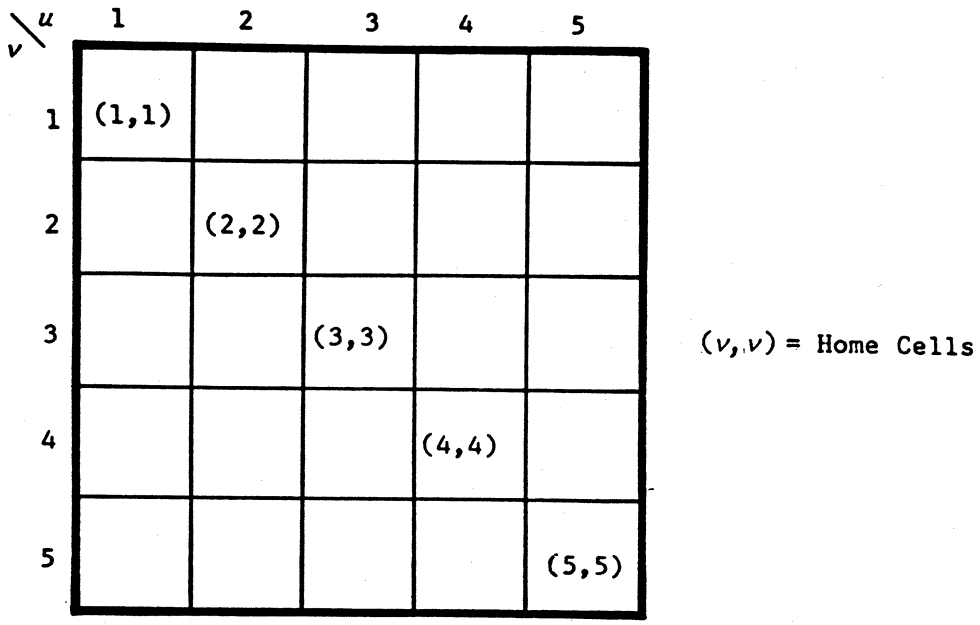


Figure 3: A cross-bar array for implementing aggregate set operations.

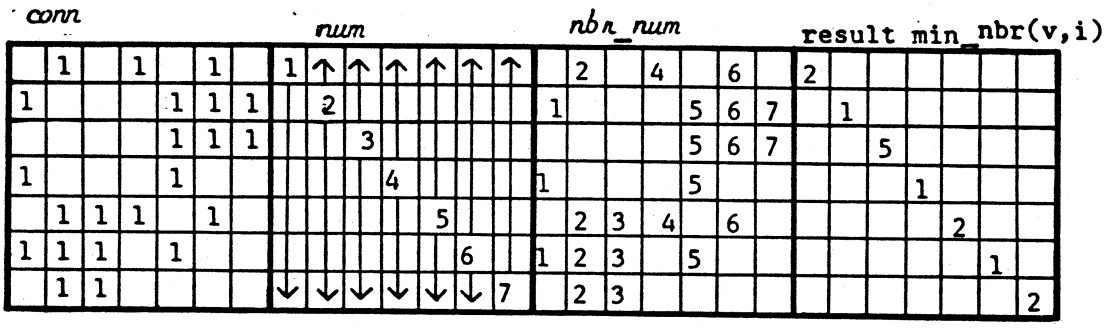


Figure 4: A sequence of snapshots of the cross-bar array obtained from Procedure A. The values `min_nbr(v,i)` for `i=1` and `v=1,2,···,7` are computed. Data used are those appearing in the example program.

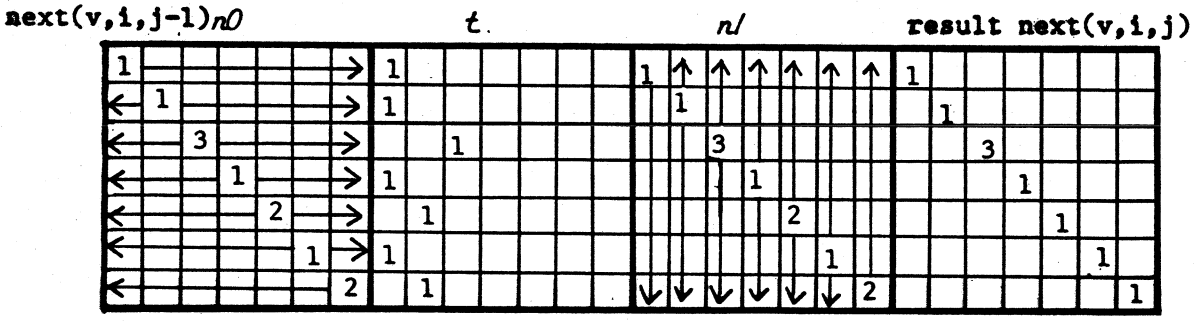


Figure 5: A sequence of snapshots of the cross-bar array obtained from Procedure C. The values `next(v,i)` for `i=1` and `v=1,2,···,7` are computed. Data used are those appearing in the example program.

Figure 4 shows the relevant information on each of the sequence of snapshots for Procedure A. The specific data appearing in the Figure for computing `min_nbr(v,i)` at `i=1` corresponds to those appearing in Figure 2.

Note that in terms of communications, the first step requires simply the loading of the input edge set `E`. The second step requires n parallel broadcasts whose cost is dependent upon the topology of the concrete network. For a real cross-bar network (bus architecture), such a broadcast can be done in a unit time step. For networks such as hypercubes or mesh-of-trees, $O(\log n)$ unit time steps suffice. For a nearest-neighbor-connected mesh, $O(n)$ unit time steps are necessary. The third step involves only local computation at each cell. The communication cost of the last step is the same as step2 except that time must be spent for the n-ary operation `min`. This cost, however, can be absorbed into the communication cost as shown below in the concrete implementation. Hence all of the three aggregate set operations listed in Section 4 can be very efficiently implemented.

5.3 Supporting complex aggregate set operations

In the above description, we assume that the right-hand side values `comp(u,i-1)` of Equation `D6` are available. The next question is, how to compute `comp(u,i-1)` for all `u in vset` and some `i`. As defined in Equation `D3`, to compute `comp(v,i)` for any `v`, if `i=0`, a local computation at each home cell which assigns the row number `v` of the cell to `comp(v,i)` will do. If `i>0`, it is again the same operation except that the value `next(v,i,log(n))` of the home cell is used. The interesting part is then how to compute `next(v,i,log(n))`. As defined by Equation `D4`, another level of iterations is indexed by `j`, where each interation involves some aggregate set operations are needed. Keep in mind that it is not a single value `next(v,i,log(n))` for a particular `v` that is demanded, but that values for all `v` are demanded. Therefore Equation `D4` is a functional definition to be used in another aggregate set operation. The interesting thing now is that the right-hand side of `D4` is neither a primitive function nor a simple function call, but a nested double recursion. The procedure for computing a complex aggregate function such as "for all `v`, `next(next(v,i,j-1),i,j-1)`" is as follows. (Figure 5 shows a sequence of snapshots of Procedure C.)

Procedure C

1. Corresponding to the inner level call `next`, broadcast the value `next(v,i,j-1)` from each home cell (v, v) horizontally to all cells in each row, and call this value $n0(v, u)$.
2. A boolean variable t at each cell (v, u) is set to 1 if the value $n0(v, u) = u$, the column number of the cell, otherwise it is set to zero.
3. Corresponding to the outer level call `next`, broadcast the value `next(v,i,j-1)` from each home cell (v, v) vertically to all cells in each column, and call this value $n1(u, v)$.
4. For those cells where $t(v, u) = 1$, send its value $n1(v, u)$ to the home cell (v, v), which is the result `next(v,i)`.

In general, a function call can be nested arbitrarily deeply. Here we only take into account those calls with at least one of its arguments being an element of the set that involves an aggregate operation. Function calls which have no argument involving inter-process communication are treated as in ordinary sequential programs.

A complex aggregate operation on element `a` of set A can be of the following form:

`g(a,b)=f_1(f_2(···(f_m(a_m,b_m),···), b_2), b_1).`

Without loss of generality, let the function calls be made on arg-tuples which are pairs `(a,b')` for all `a in A` with some `b'` in {`b_1, b_2, ···, b_m`}. Since all RHS functional values are available by induction, they can be send to cell `(a,b)` before the following procedure starts.

Procedure G

1. $new(a,a)$=`f_m(a,b_m)`, $next(a,a)$=`f_m-1(a,b_m-1)`.
2. Broadcast each value $new(a,a)$ from home cell (a,a) horizontally to all cells in each row, and call these values $n0(a,a')$.
3. For $k = m-2$ down to 0:
   (a) The boolean variable $t0$ at each cell (a,a') is set to 1 if the value $n0(a,a') = a'$, the column number of the cell; otherwise it is set to zero.
   (b) Broadcast each value $next(a,a)$ from home cell (a,a) vertically to all cells in each column, and call these values $n1(a',a)$.
   (c) If $k = 0$, go to step 4. If $k > 0$, then for those cells where $t0(a,a') = 1$, broadcast its value $n1(a,a')$ horizontally on the row, and call these values $n0(a,a')$.
   (d) Set $next(a,a)$=`f_k(a,b_k)`; Repeat the loop.
4. For those cells where $t0(a,a') = 1$, send its value $n1(a,a')$ to the home cell (a,a'), which is the desired result.

Figure 6 shows a sequence of snapshots of Procedure G for `m`=4. The values shown in the cross-bar array are only the relevant ones.

It is easy to see how the cross-bar array model can be generalized for more complex functional forms and set notations. For instance, if a set which has tuples rather than scaler as its elements, then nested function calls may appear in more than one place in the arg-tuples, and thus a higher dimensional cross-bar array is demanded.

5.4 Other models for aggregate set operations

The cross-bar array model can be implemented very efficiently on a variety of networks. The time complexity for each aggregate set operation which involves $O(n)$ very global and irregular communications, is only $O(\log n)$ on hypercube networks or mesh-of-tree networks. The only drawback of the cross-bar array is that it may become wasteful in terms of the number of processors needed, which is $O(n^2)$ for a set of n elements. Fortunately, the aggregate set operations can be supported by an alternative "list" model, which uses only $O(n)$ number of processors, but $O(\log^2 n)$ communication time [10].

The cross-bar array model is most suitable for aggregate set operations where the inter-element dependency is strong, i.e., where the number of edges in the DAG is large. The "list" model is suitable for the case when the DAG is sparse graph.

6 Concrete Multiprocessor Networks

There are two issues concerning mapping cross-bar arrays to concrete multiprocessor networks. One is the matching of the topologies between the abstract and the concrete networks, and the other is the matching of their sizes, or, the number of processors.

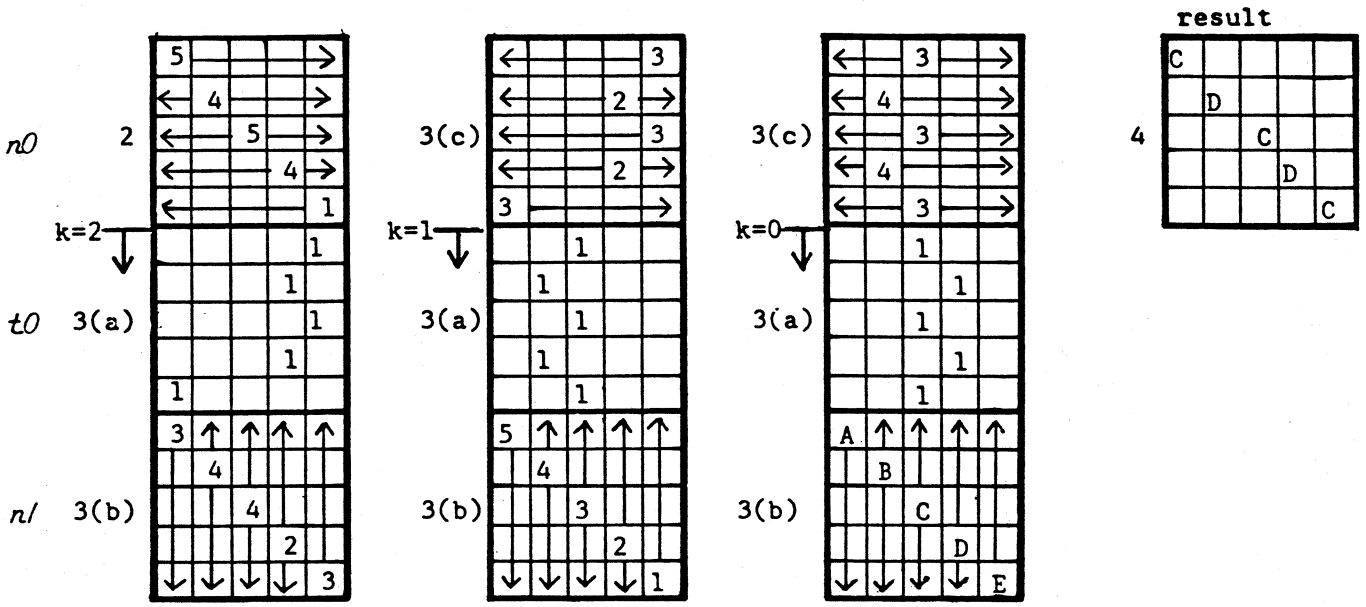


Figure 6: A sequence of snapshots of the cross-bar array for Procedure G. The values g(a,b) for all a are computed, where g(a,b) is defined as g(a,b)=f_1(f_2(···(f_m(a_m,b_m),···), b_2), b_1).

The four example functions are [f_1(a,b_1)| a = 1:5] = ['A', 'B', 'C', 'D','E'], [f_2(a,b_2)| a = 1:5] = [5, 4, 3, 2, 1], [f_3(a,b_3)| a = 1:5] = [3, 4, 4, 2, 3], [f_4(a,b_4)| a = 1:5] = [5, 4, 5, 4, 1].

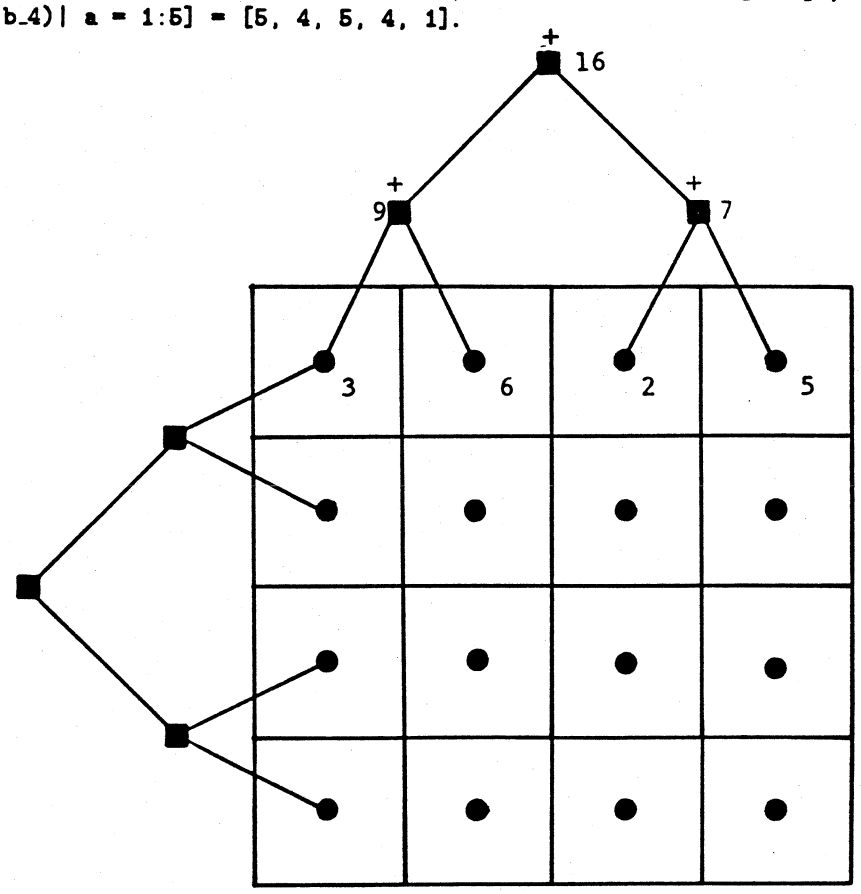


Figure 7: A 2-dimensional mesh-of-tree, but with only two trees shown in the figure.

6.1 Matching the size

The size problem arises when a problem requires aggregate set operations on n elements while the host network has fewer then n^2 number of processors. It can be easily solved, however, by first forcing a cross-bar array to be the size $N = m^2$ of the host network, then subdividing each cell of the cross-bar array into $k = (n/m)^2$ logical cells so as to fit a problem of size n^2. The subdivision has two effects on the processor assigned to any cross-bar array cell. The first one is that the workload of a processor will probably be as much as k-fold the original load if there are enough processors, (or at least n/m-fold [6]). Secondly, the grain size of the data broadcast or transferred along row and columns of the cross-bar array will also become as much as k-fold of the originial one.

6.2 Matching the topology

Embedding a cross-bar array to a mesh-of-tree network is simple and straightforward. A 2-dimensional mesh-of-tree network with 4 nodes on each side is shown in Figure 7. Not all trees, but only one for the column and one for the row, are shown. One can see that a 2-dimensional cross-bar array can be mapped directly to a 2-dimentional mesh-of-tree of the same size.

Broadcasting a value of any given leaf node to those in the same column (row) can be achieved by first sending the value up the column (row) tree of that node and then sending down to all other nodes of the tree. Applying an associative operator **op** to the values in a row (column) of leave nodes is called a *merge* operation with operator **op**. It can be accomplished by allowing each intermediate tree node of the row (column) tree to perform the corresponding binary operation of **op** on the two values sent up from its children nodes. The merge operation with addition on the values in the first row is shown in Figure 7.

The embedding of the cross-bar array to a nearest-neighbor connected mesh is just as simple, except that all broadcast and merge must be performed serially on the column and the rows; thus the $O(n)$ time complexity mentioned before. The embedding of the cross-bar array into a hypercube network is shown below.

6.3 Embedding a cross-bar array into a hypercube network

A k-dimentional hypercube (or Boolean k-cube) network contains $N = 2^k$ processors, as shown in Figure 8, where $k = 4$. For the sake of simplicity, assume k is even so that that N is a perfect square. Such a host network can accommodate an n-by-n cross-bar array, where $n = \sqrt{N}$. The easiest way to see the embedding is by looking at the binary expansion of the hypercube node label: suppose that they are labeled from 0 to $N - 1$. Let the higher $k/2$ bits of the node label be its row number, with the lower $k/2$ bits being the column number. Figure 9 shows, on the top, the 4 rows of a 4-cube where nodes on the same row are connected by heavier lines, and on the bottom, the 4 columns.

6.4 Broadcast

The above embedding of an array into a hypercube has the property that every row (column) is on a separate hypercube of lower dimension from other rows (columns). Any data movement on a row or a column can be achieved by using a spanning tree of all nodes in the sub-hypercube as shown in Figure 9. Since for all of the aggregate operations data movements are either on every row or on every column, no conflict of communications arises.

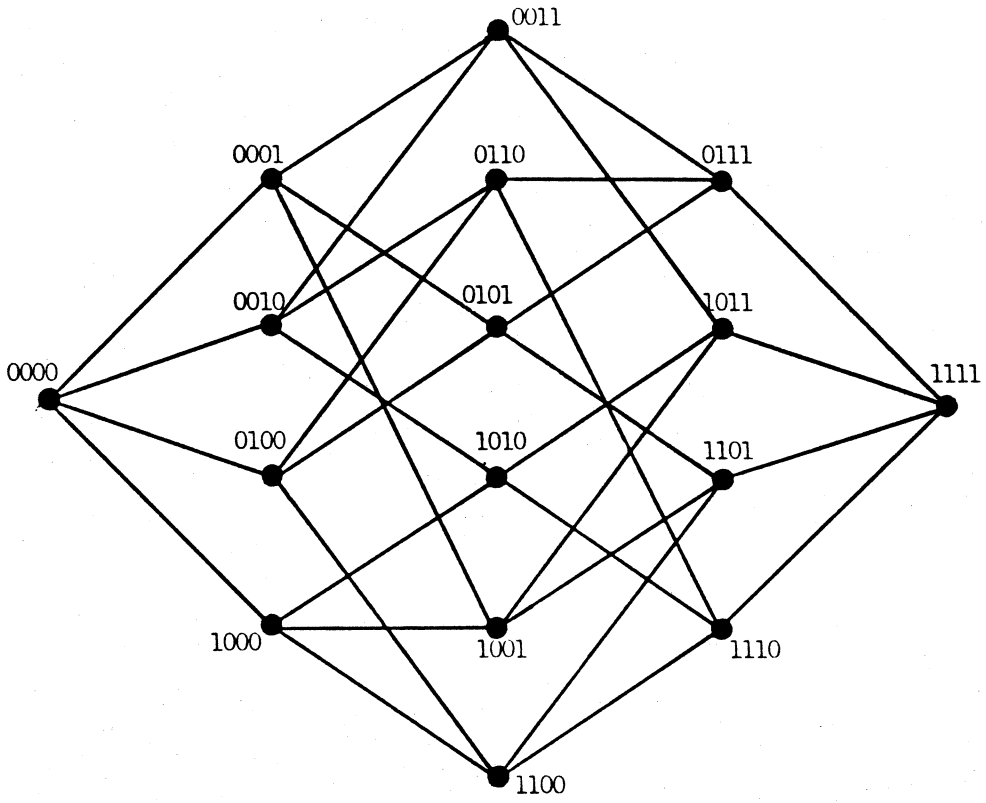
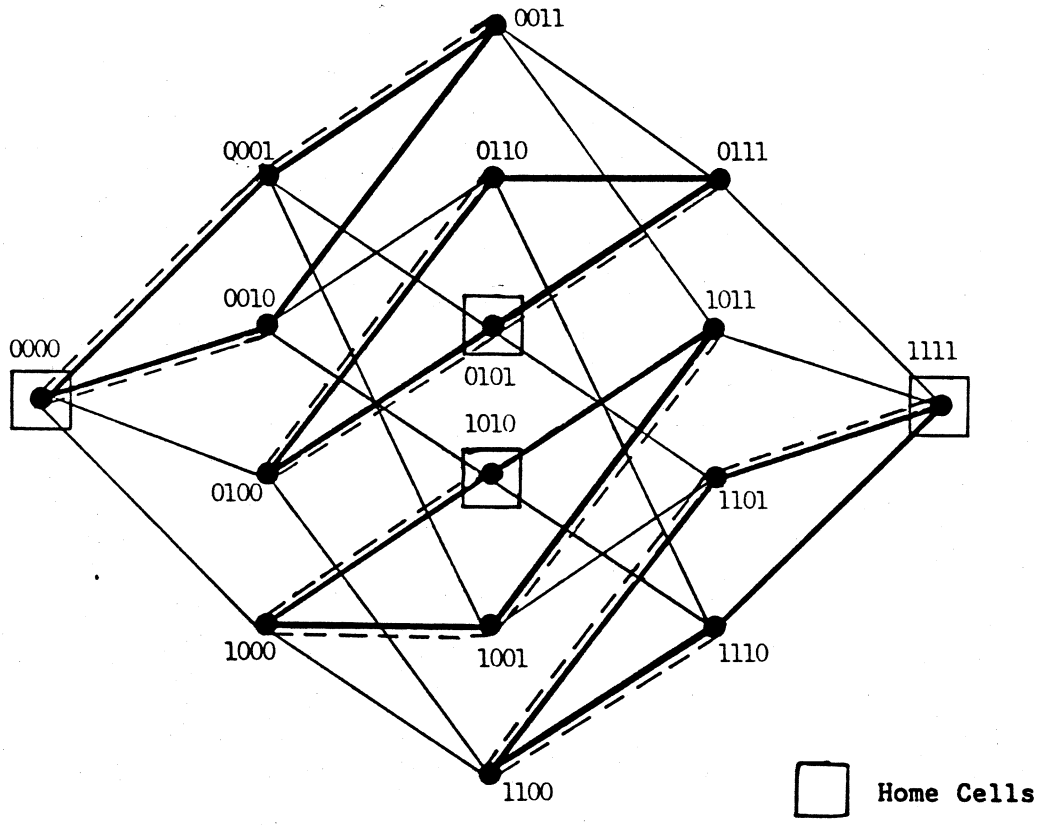


Figure 8: A 4-dimentional hypercube.



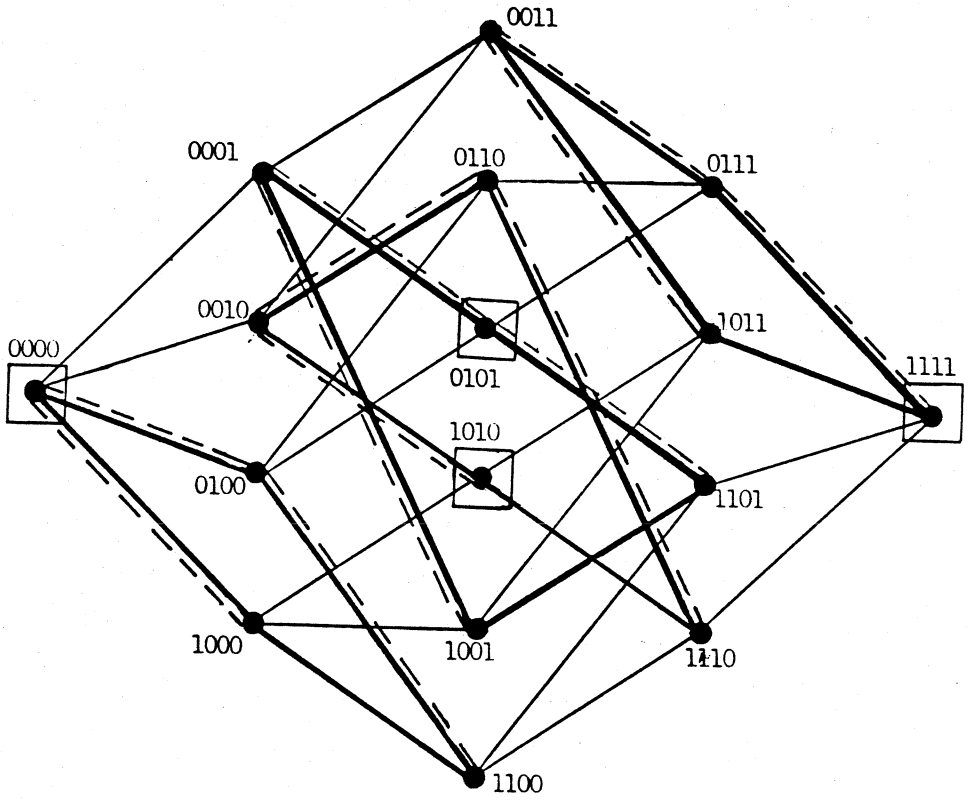


Figure 9: The rows and columns of the embedded cross-bar array.

6.5 Merge

With the spanning tree for each row and column in place, a merge with any associative operator **op** can be accomplished in a similar way as in the case of a tree-of-mesh network. One thing to note is that the choice of spanning tree affects the arity of the operator at each node for accomplishing the merge operation.

There are a variety of ways for generating spanning trees for a hypercube. One of the standard ones can be called a *dynamic spanning tree*, because not all of its tree edges are active at once for the merge operation. To construct the dynamic spanning tree, one starts by choosing a node as the root. A tree edge is then drawn from the root to the node whose label differs from the root only in its k'th bit. The edge is also labeled with the number k. Next, from both of the nodes in the tree, an edge is drawn from each to another node whose label differs from it by the $(k-1)$'th bit. These two edges are labeled $k-1$. The procedure repeats for until all nodes are included in the tree.

The merge with operator **op** proceeds as follows: (1) each leaf node that is connected to its parent node by an edge labeled k sends up its value to the parent node and to be merged by the binary **op** with the value at the parent node. (2) Such nodes can be considered vanished after sending up their values. Now the tree is $k-1$ levels deep. The process repeats until the merge at the root. Figure 10 illustrates the spanning tree for a 3-cube and the merge operation. The advantage of this spanning tree is that only binary operators are needed.

7 Concluding Remarks

We have shown how to program parallel machines in a very-high-level language **Crystal** where programs look like algorithm descriptions. Such algorithm descriptions use set expressions extensively. Operations on set elements can be abstracted and treated collectively as a single entity. Moreover, aggregate set operations can be supported very efficiently on machines with a large-scale of cooperative, distributed, parallel processes.

This paper demonstrates the viability of programming for parallelism without explicitly specifying inter-process communications. The most noteworty effect of the very-high-level language, along with its programming abstractions, is that it brings the activity of programming, debugging, and testing to a new conceptual level. It promotes a style of programming that is most effective for parallel machines and allows programmers to concentrate on algorithm issues instead of implementation issues.

8 Acknowledgement

I am indebted to Ming-Deh Huang and Charles Leiserson for pointing me to their respective work on parallel graph algorithms, which provide much of the inspiration for the aggregate set operations.

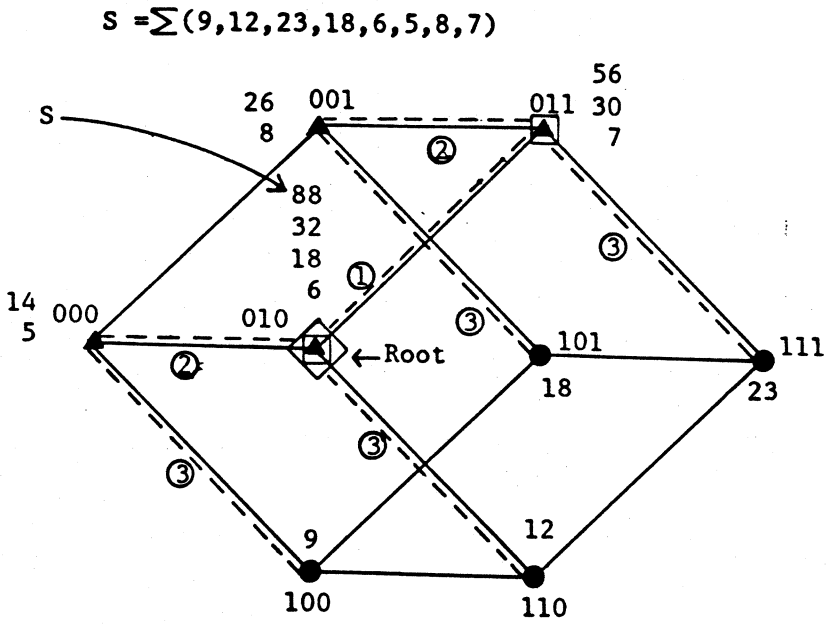


Figure 10: A spanning tree for a 3-cube. To compute the sum of the values stored in each node, every non-leaf node does at least one binary addition while the root node participates in every step. The label on the edge going into a child node indicates the time step at which a node participates, with its own value as one operand and the one from the child node as another. Numbers appearing beside each node include the initial value in each node and those computed at the intermediate steps.