Basic Communication and Data Sharing Patterns

CSE 531

Spring 2023

Mahmut Taylan Kandemir

Topic Overview

- One-to-All Broadcast and All-to-One Reduction
- All-to-All Broadcast and Reduction
- All-Reduce and Prefix-Sum Operations
- Scatter and Gather
- All-to-All Personalized Communication
- Circular Shift
- Improving the Speed of Some Communication Operations

Basic Communication Operations: Introduction

- Many interactions in practical parallel programs occur in well-defined patterns involving groups of processors.
- Efficient implementations of these operations can improve performance, reduce development effort and cost, and improve software quality.
- Efficient implementations must leverage underlying architecture. For this reason, we refer to specific architectures here.
- We select a descriptive set of architectures to illustrate the process of algorithm design.

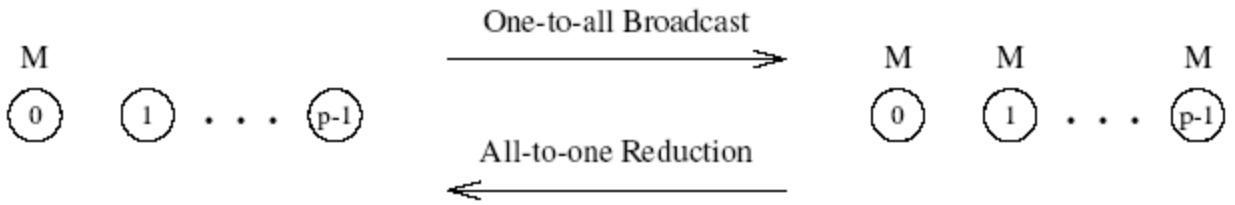
Basic Communication Operations: Introduction

- Group communication operations are built using point-to-point messaging primitives.
- Recall from our discussion of architectures that communicating a message of size m over an uncongested network takes time $t_s + t_w w$.
- We use this as the basis for our analyses. Where necessary, one can take congestion into account explicitly by scaling the t_w term.
- We assume that the network is bidirectional and that communication is single-ported.

One-to-All Broadcast and All-to-One Reduction

- One-to-All Broadcast: one processor has a piece of data (of size *m*) it needs to send to everyone.
- The dual of one-to-all broadcast is All-to-One Reduction.
- In all-to-one reduction, each processor has *m* units of data. These data items must be combined piece-wise (using some associative operator, such as addition or min), and the result made available at a target processor.

One-to-All Broadcast and All-to-One Reduction

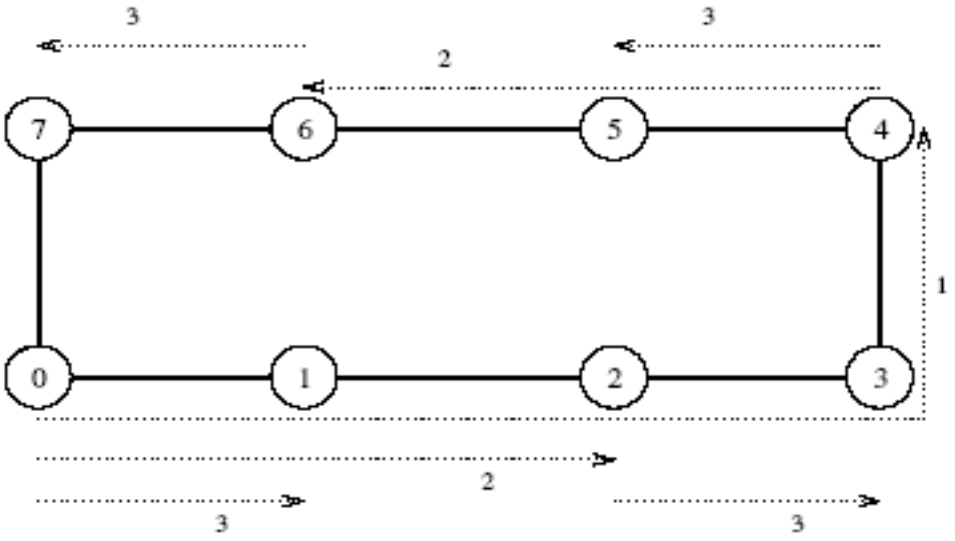


One-to-all broadcast and all-to-one reduction among processors.

One-to-All Broadcast and All-to-One Reduction on Rings

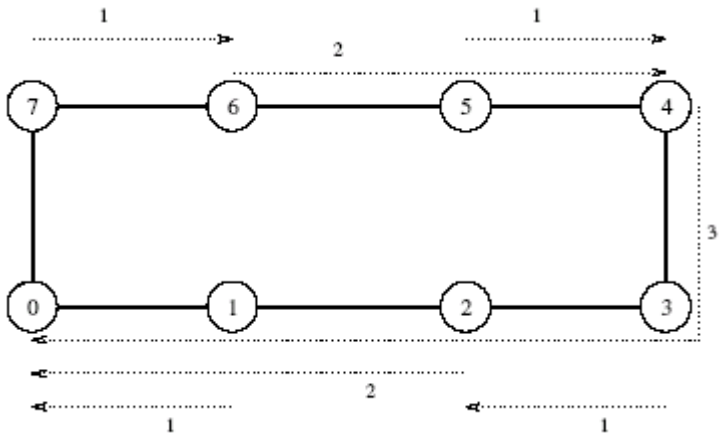
- Simplest way is to send *p-1* messages from the source to the other *p-1* processors - this is *not* very efficient. Why?
- Use Recursive Doubling: source sends a message to a selected processor. We now have two independent problems defined over halves of machines.
- Reduction can be performed in an identical fashion by inverting the process.

One-to-All Broadcast



One-to-all broadcast on an eight-node ring. Node 0 is the source of the broadcast. Each message transfer step is shown by a numbered, dotted arrow from the source of the message to its destination. The number on an arrow indicates the time step during which the message is transferred.

All-to-One Reduction



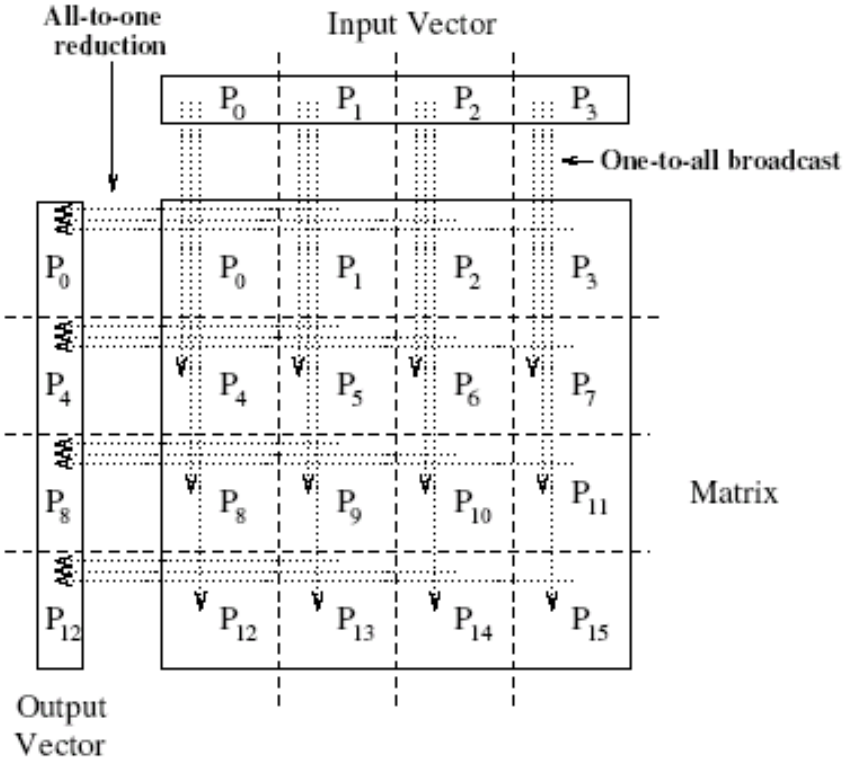
Reduction on an eight-node ring with node 0 as the destination of the reduction.

Broadcast and Reduction: Example

Consider the problem of multiplying a matrix with a vector.

- The *n x n* matrix is assigned to an *n x n* (virtual) processor grid. The vector is assumed to be on the first row of processors.
- The first step of the product requires a one-to-all broadcast of the vector element along the corresponding column of processors. This can be done concurrently for all *n* columns.
- The processors compute local product of the vector element and the local matrix entry.
- In the final step, the results of these products are accumulated to the first row using *n* concurrent all-to-one reduction operations along the columns (using the sum operation).

Broadcast and Reduction: Matrix-Vector Multiplication Example

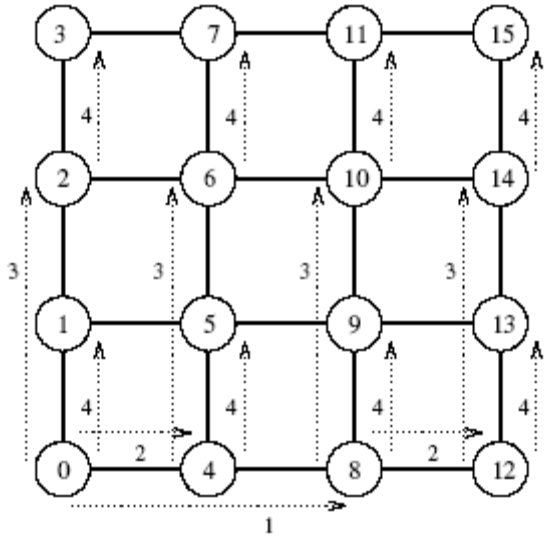


One-to-all broadcast and all-to-one reduction in the multiplication of a *4 x 4* matrix with a *4 x 1* vector.

Broadcast and Reduction on a Mesh

- We can view each row and column of a square mesh of p nodes as a linear array of $\sqrt{p}$ nodes.
- Broadcast and reduction operations can be performed in two steps – the first step does the operation along a row and the second step along each column concurrently.
- This process generalizes to higher dimensions as well.

Broadcast and Reduction on a Mesh: Example

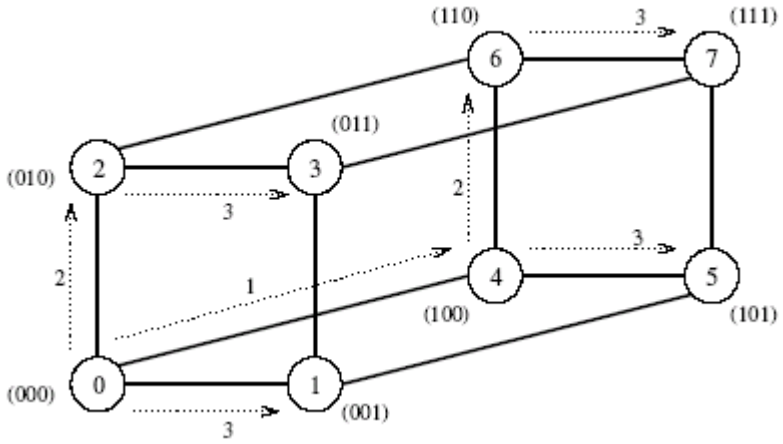


One-to-all broadcast on a 16-node mesh.

Broadcast and Reduction on a Hypercube

- A hypercube with 2^d nodes can be regarded as a d-dimensional mesh with two nodes in each dimension.
- The mesh algorithm can be generalized to a hypercube and the operation is carried out in d (= $\log p$) steps.

Broadcast and Reduction on a Hypercube: Example

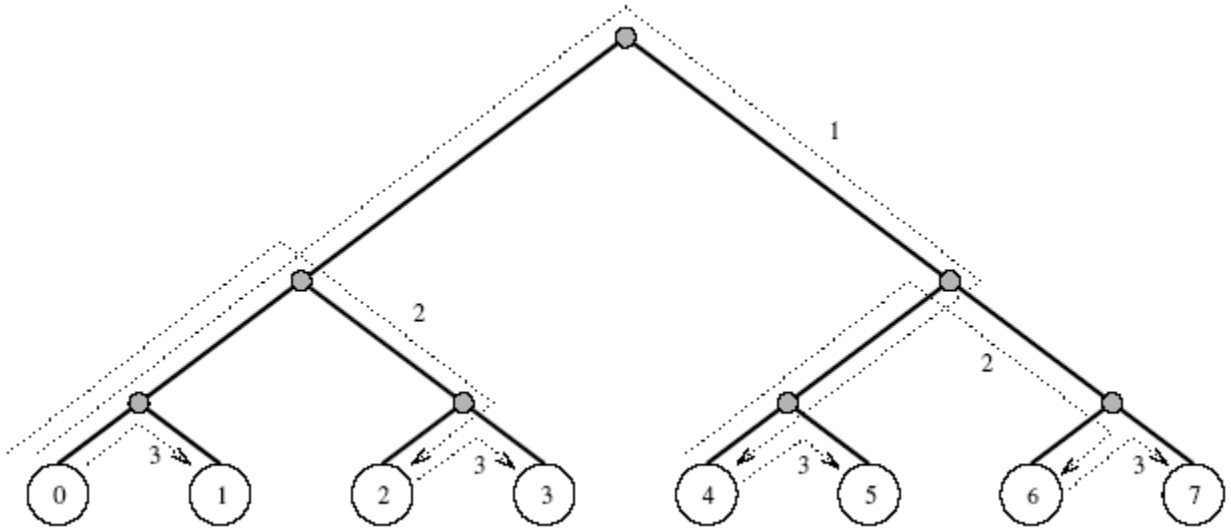


One-to-all broadcast on a three-dimensional hypercube. The binary representations of node labels are shown in parentheses.

Broadcast and Reduction on a Balanced Binary Tree

- Consider a binary tree in which processors are (logically) at the leaves and internal nodes are routing nodes.
- Assume that source processor is the root of this tree. In the first step, the source sends the data to the right child (assuming the source is also the left child). The problem has now been decomposed into two problems with half the number of processors.

Broadcast and Reduction on a Balanced Binary Tree



One-to-all broadcast on an eight-node tree.

Broadcast and Reduction Algorithms

- All of the algorithms described above are adaptations of the same algorithmic template.
- We illustrate the algorithm for a hypercube, but the algorithm, as has been seen, can be adapted to other architectures.
- The hypercube has 2^d nodes and *my_id* is the label for a node.
- *X* is the message to be broadcast, which initially resides at the source node 0.

Broadcast and Reduction Algorithms

```
1.    procedure GENERAL_ONE_TO_ALL_BC(d, my_id, source, X)
2.    begin
3.        my_virtual_id := my_id XOR source;
4.        mask := 2^d − 1;
5.        for i := d − 1 downto 0 do      /* Outer loop */
6.            mask := mask XOR 2^i;    /* Set bit i of mask to 0 */
7.            if (my_virtual_id AND mask) = 0 then
8.                if (my_virtual_id AND 2^i) = 0 then
9.                    virtual_dest := my_virtual_id XOR 2^i;
10.                   send X to (virtual_dest XOR source);
              /* Convert virtual_dest to the label of the physical destination */
11.               else
12.                   virtual_source := my_virtual_id XOR 2^i;
13.                   receive X from (virtual_source XOR source);
              /* Convert virtual_source to the label of the physical source */
14.               endelse;
15.       endfor;
16.   end GENERAL_ONE_TO_ALL_BC
```

One-to-all broadcast of a message *X* from *source* on a hypercube.

Only the nodes with 0 in the *i* least significant bits of their labels participate in communication along dimension *i*.

The variable *mask* helps determine which nodes communicate in a particular iteration of the loop.

Cost Analysis

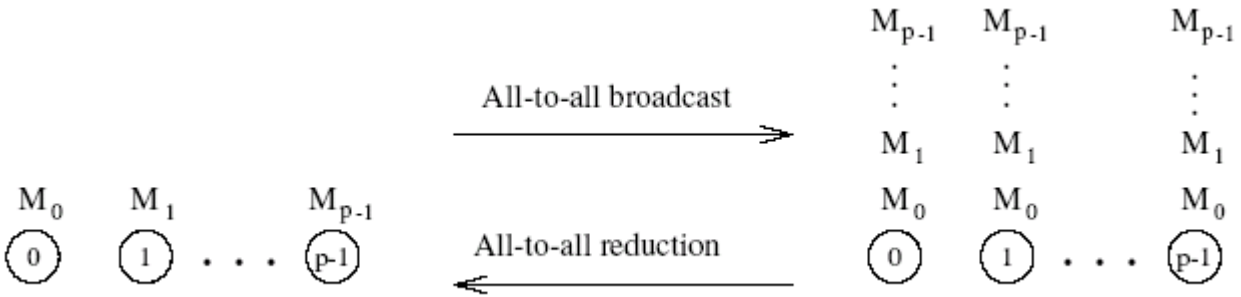
- The broadcast or reduction procedure involves *log p* point-to-point simple message transfers, each at a time cost of $t_s + t_w m$.
- The total time is therefore given by:

$$T = (t_s + t_w m) \log p.$$

All-to-All Broadcast and Reduction

- Generalization of broadcast in which each processor is the *source* as well as *destination*.
- A process sends the same *m*-word message to every other process, but different processes may broadcast different messages.

All-to-All Broadcast and Reduction



All-to-all broadcast and all-to-all reduction.

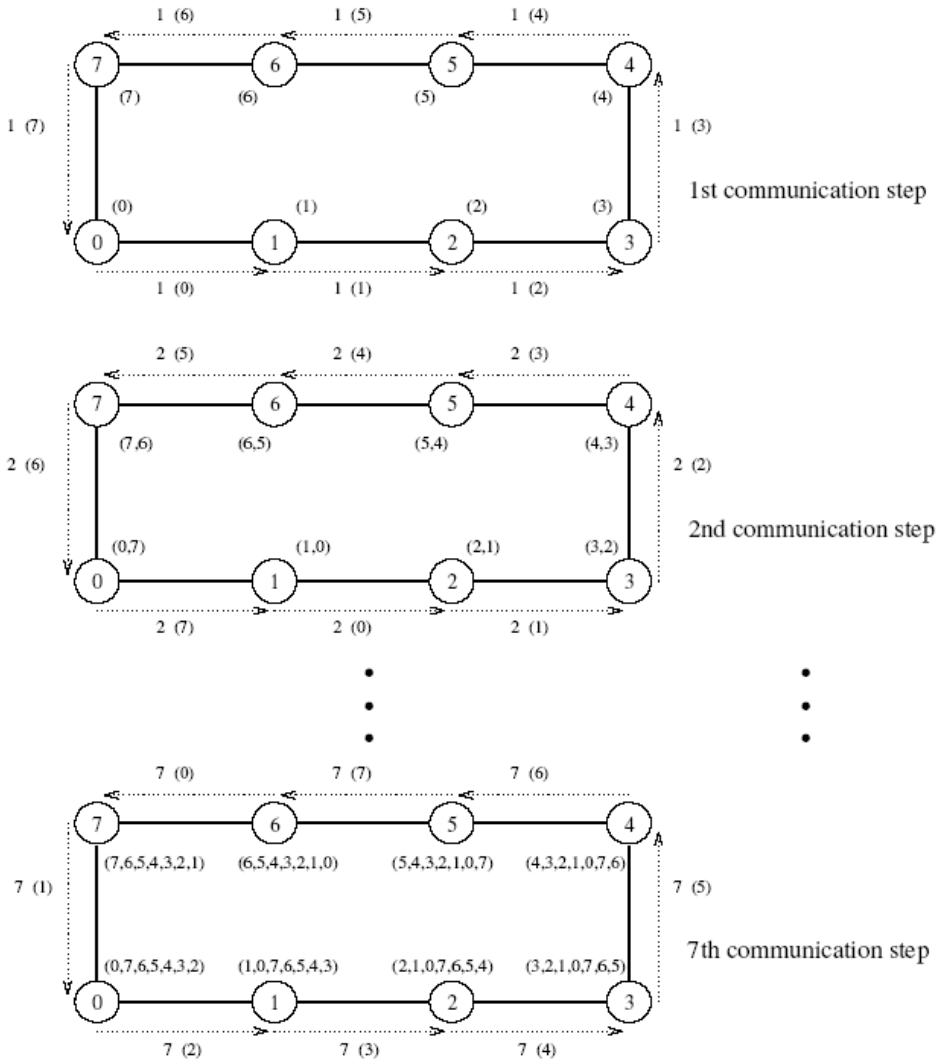
All-to-All Broadcast and Reduction on a Ring

- Any idea?

All-to-All Broadcast and Reduction on a Ring

- Simplest approach: perform *p* one-to-all broadcasts. This is not the most efficient way, though.
- Each node first sends to one of its neighbors the data it needs to broadcast.
- In subsequent steps, it forwards the data received from one of its neighbors to its other neighbor.
- The algorithm terminates in *p-1* steps.

All-to-All Broadcast and Reduction on a Ring



All-to-all broadcast on an eight-node ring.

All-to-All Broadcast and Reduction on a Ring

```
1.    procedure ALL_TO_ALL_BC_RING(my_id, my_msg, p, result)
2.    begin
3.        left := (my_id − 1) mod p;
4.        right := (my_id + 1) mod p;
5.        result := my_msg;
6.        msg := result;
7.        for i := 1 to p − 1 do
8.            send msg to right;
9.            receive msg from left;
10.           result := result ∪ msg;
11.       endfor;
12.   end ALL_TO_ALL_BC_RING
```

All-to-all broadcast on a *p*-node ring.

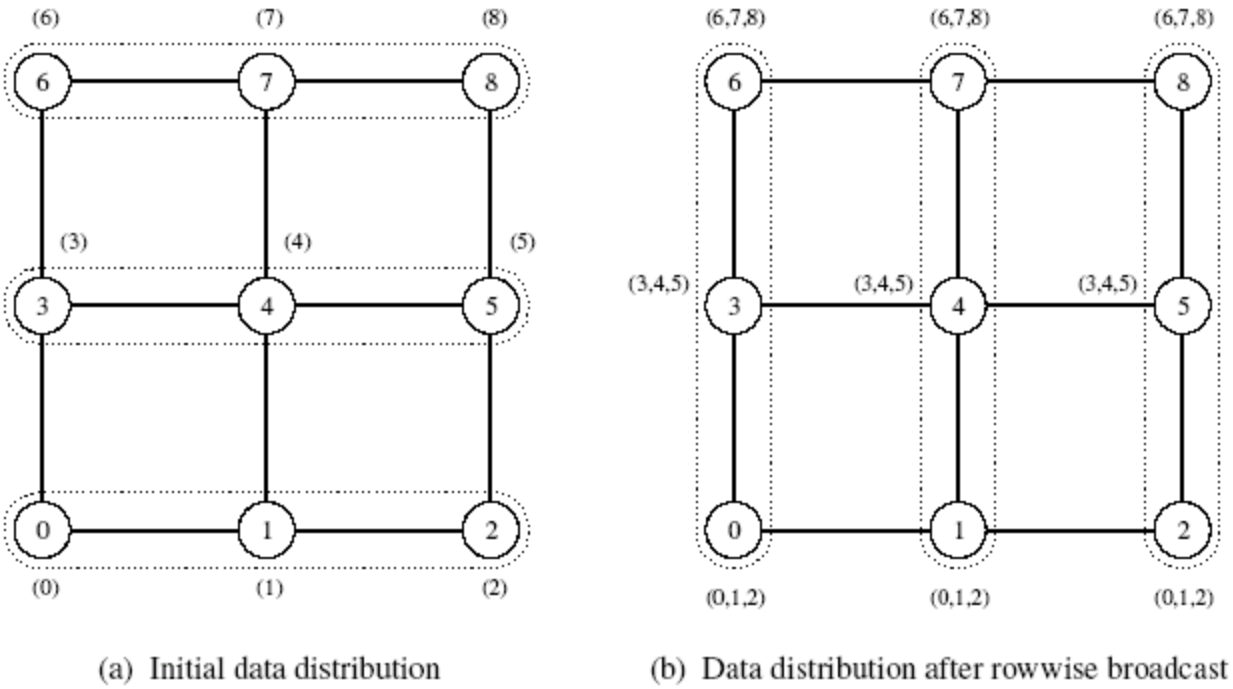
All-to-all Broadcast on a Mesh

- Any idea?

All-to-all Broadcast on a Mesh

- Performed in *two phases* – in the first phase, each row of the mesh performs an all-to-all broadcast using the procedure for the linear array.
- In this phase, all nodes collect $\sqrt{p}$ messages corresponding to the $\sqrt{p}$ nodes of their respective rows. Each node consolidates this information into a single message of size $m\sqrt{p}$.
- The second communication phase is a columnwise all-to-all broadcast of the consolidated messages.

All-to-all Broadcast on a Mesh



(a) Initial data distribution

(b) Data distribution after rowwise broadcast

All-to-all broadcast on a *3 x 3* mesh. The groups of nodes communicating with each other in each phase are enclosed by dotted boundaries. By the end of the second phase, all nodes get (0,1,2,3,4,5,6,7) (that is, a message from each node).

All-to-all Broadcast on a Mesh

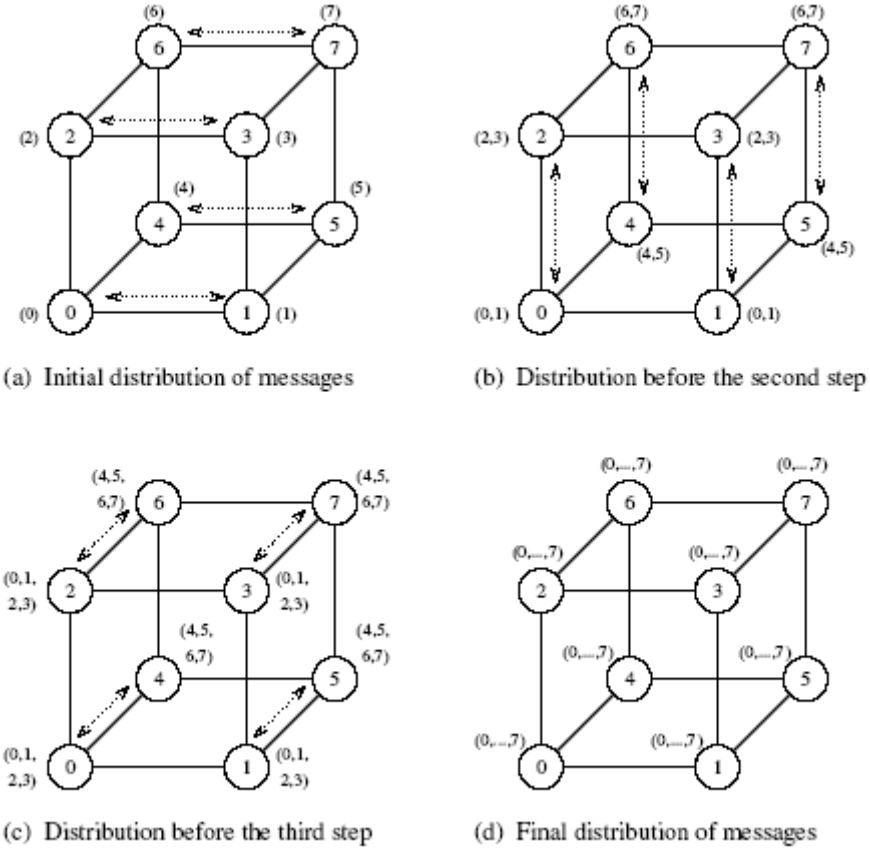
```
1.      procedure ALL_TO_ALL_BC_MESH(my_id, my_msg, p, result)
2.      begin
/* Communication along rows */
3.          left := my_id − (my_id mod √p) + (my_id − 1) mod √p;
4.          right := my_id − (my_id mod √p) + (my_id + 1) mod √p;
5.          result := my_msg;
6.          msg := result;
7.          for i := 1 to √p − 1 do
8.              send msg to right;
9.              receive msg from left;
10.             result := result ∪ msg;
11.         endfor;
/* Communication along columns */
12.         up := (my_id − √p) mod p;
13.         down := (my_id + √p) mod p;
14.         msg := result;
15.         for i := 1 to √p − 1 do
16.             send msg to down;
17.             receive msg from up;
18.             result := result ∪ msg;
19.         endfor;
20.     end ALL_TO_ALL_BC_MESH
```

All-to-all broadcast on a square mesh of *p* nodes.

All-to-all broadcast on a Hypercube

- Generalization of the mesh algorithm to *log p* dimensions.
- Message size doubles at each of the *log p* steps.

All-to-all broadcast on a Hypercube



All-to-all broadcast on an eight-node hypercube.

All-to-all broadcast on a Hypercube

```
1.    procedure ALL_TO_ALL_BC_HCUBE(my_id, my_msg, d, result)
2.    begin
3.        result := my_msg;
4.        for i := 0 to d − 1 do
5.            partner := my_id XOR 2^i;
6.            send result to partner;
7.            receive msg from partner;
8.            result := result ∪ msg;
9.        endfor;
10.   end ALL_TO_ALL_BC_HCUBE
```

All-to-all broadcast on a *d*-dimensional hypercube.

All-to-all Reduction

- Similar communication pattern to all-to-all broadcast, except in the reverse order.
- On receiving a message, a node must combine it with the local copy of the message that has the same destination as the received message before forwarding the combined message to the next neighbor.

Cost Analysis

- On a ring, the time is given by: $(t_s + t_w m)(p-1)$.
- On a mesh, the time is given by: $2t_s(\sqrt{p} - 1) + t_w m(p-1)$.
- On a hypercube, we have:

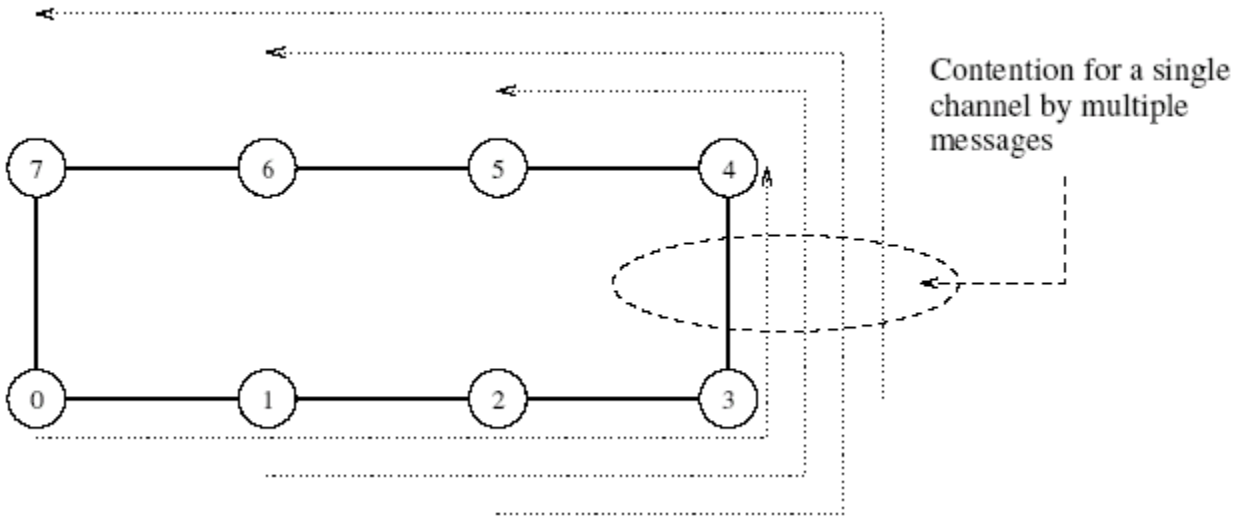
$$T = \sum_{i=1}^{\log p} (t_s + 2^{i-1} t_w m)$$

$$= t_s \log p + t_w m(p-1).$$

All-to-all broadcast: Notes

- All of the algorithms presented above are asymptotically optimal in message size.
- It is not possible to port algorithms for higher dimensional networks (such as a hypercube) into a ring because this would cause contention.

All-to-all broadcast: Notes

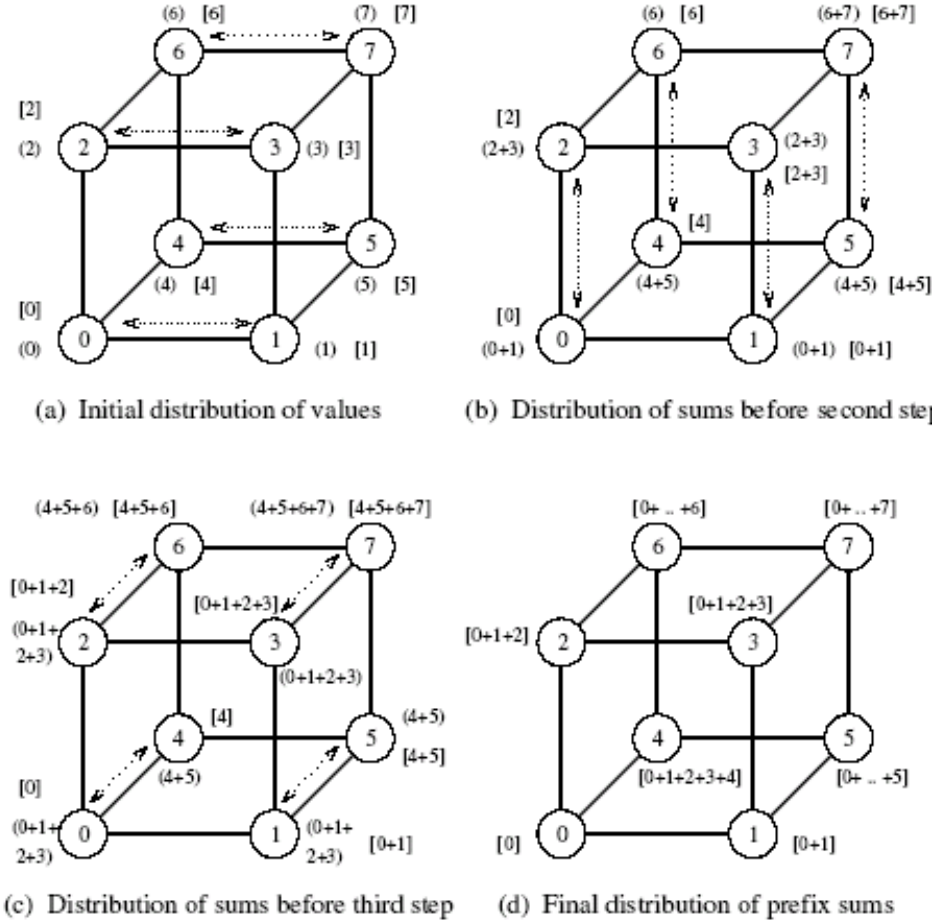


Contention for a channel when the hypercube is mapped onto a ring.

The Prefix-Sum Operation

- Also known as Scan Operation
- Given p numbers $n_0, n_1, \ldots, n_{p-1}$ (one on each node), the problem is to compute the sums $s_k = \sum_{i=0}^{k} n_i$ for all k between 0 and $p-1$.
- Initially, n_k resides on the node labeled k, and at the end of the procedure, the same node holds S_k.
- Similar to all-to-all broadcast

The Prefix-Sum Operation



Computing prefix sums on an eight-node hypercube. At each node, square brackets show the local prefix sum accumulated in the result buffer and parentheses enclose the contents of the outgoing message buffer for the next step.

The Prefix-Sum Operation

- The operation can be implemented using the all-to-all broadcast kernel.
- We must account for the fact that in prefix sums the node with label *k* uses information from only the *k*-node subset whose labels are less than or equal to *k*.
- This is implemented using an additional result buffer. The content of an incoming message is added to the result buffer *only if* the message comes from a node with a smaller label than the recipient node.
- The contents of the outgoing message (denoted by parentheses in the figure) are updated with every incoming message.

The Prefix-Sum Operation

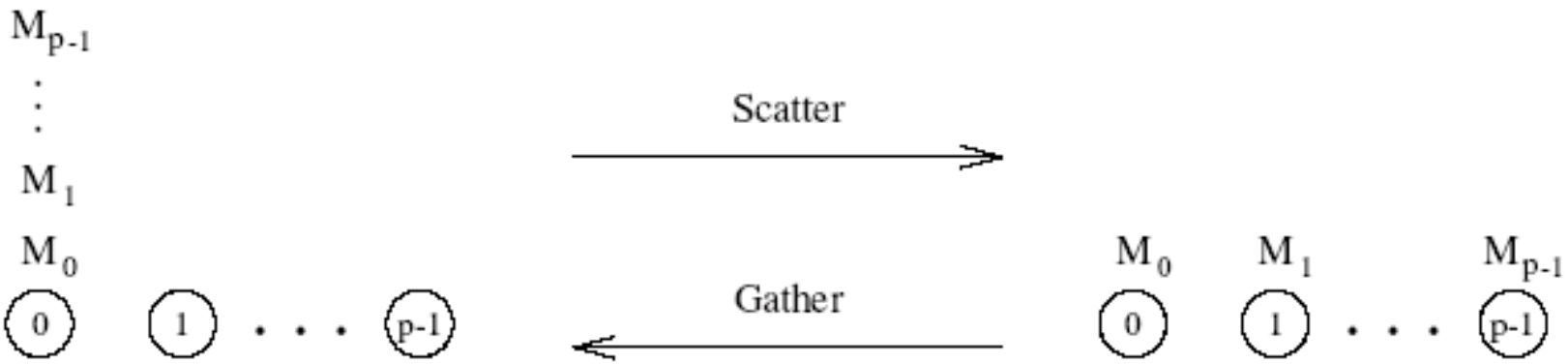
```
1.    procedure PREFIX_SUMS_HCUBE(my_id, my_number, d, result)
2.    begin
3.        result := my_number;
4.        msg := result;
5.        for i := 0 to d − 1 do
6.            partner := my_id XOR 2^i;
7.            send msg to partner;
8.            receive number from partner;
9.            msg := msg + number;
10.           if (partner < my_id) then result := result + number;
11.       endfor;
12.   end PREFIX_SUMS_HCUBE
```

Prefix sums on a *d*-dimensional hypercube.

Scatter and Gather

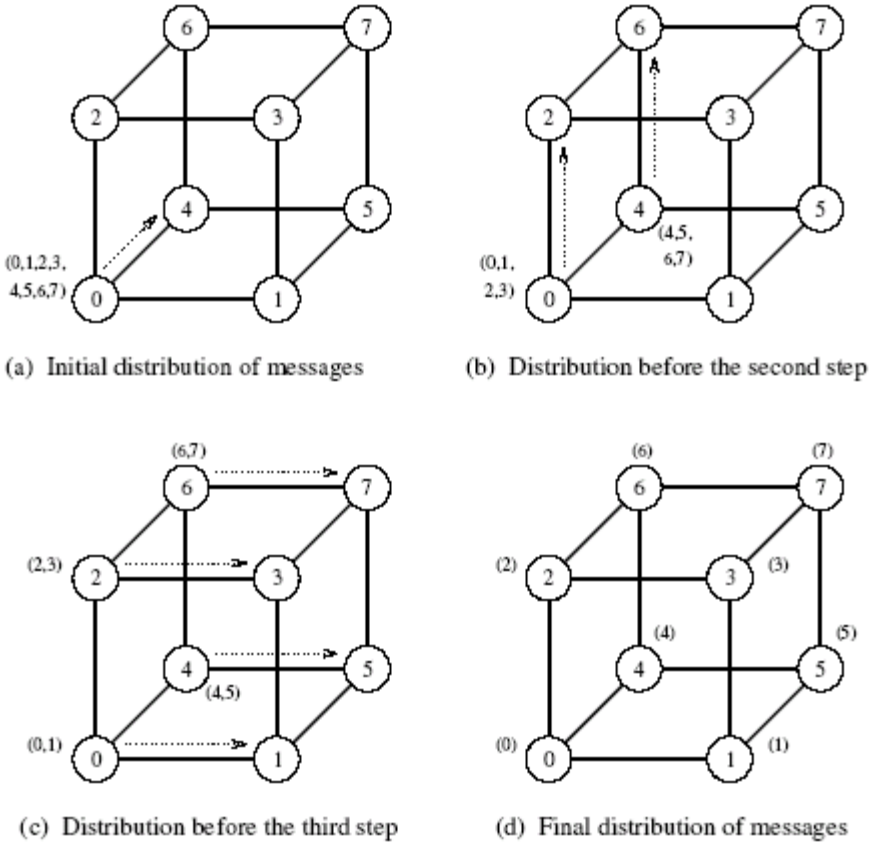
- In the *scatter* operation, a single node sends a unique message of size *m* to every other node (also called a one-to-all personalized communication).
- In the *gather* operation, a single node collects a unique message from each node.
- While the scatter operation is fundamentally different from broadcast, the algorithmic structure is similar, except for differences in message sizes (messages get smaller in scatter and stay constant in broadcast).
- The gather operation is exactly the inverse of the scatter operation and can be executed as such.

Gather and Scatter Operations



Scatter and gather operations.

Example of the Scatter Operation



The scatter operation on an eight-node hypercube.

Cost of Scatter and Gather

- There are *log p* steps, in each step, the machine size halves and the data size halves.
- We have the time for this operation to be:

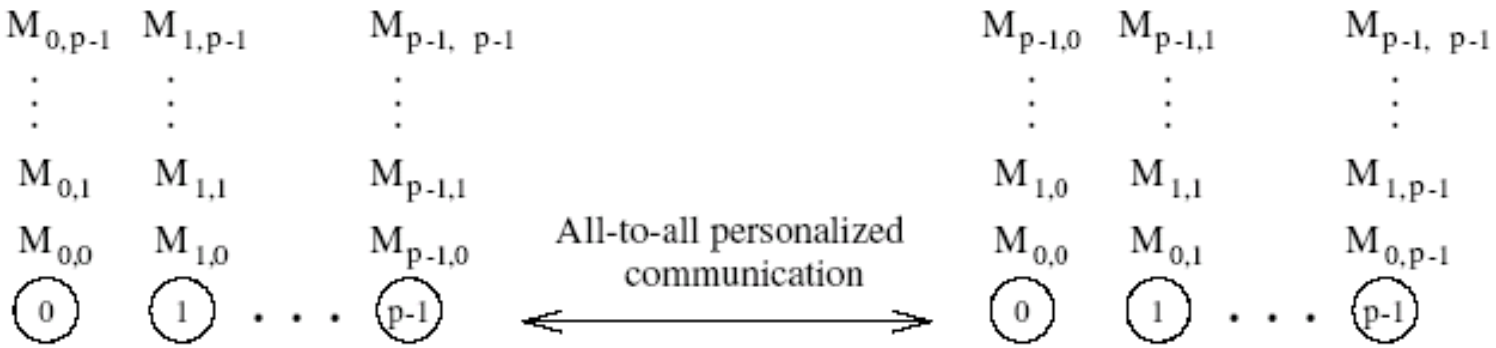
$$T = t_s \log p + t_w m(p-1).$$

- This time holds for a linear array as well as a 2-D mesh.
- These times are asymptotically optimal in message size.

All-to-All Personalized Communication

- Each node has a *distinct* message of size *m* for every other node.
- This is unlike all-to-all broadcast, in which each node sends the *same* message to all other nodes.
- All-to-all personalized communication is also known as total exchange.

All-to-All Personalized Communication

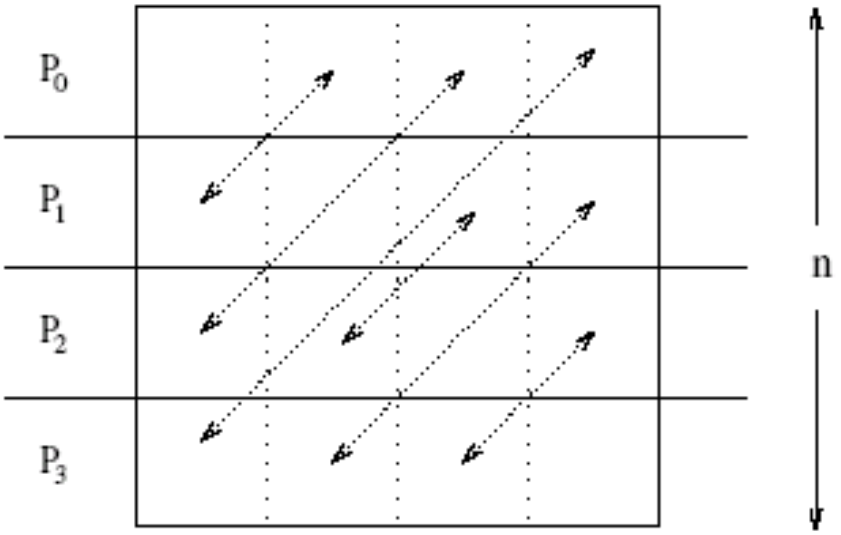


All-to-all personalized communication.

All-to-All Personalized Communication: Example

- Consider the problem of transposing a matrix.
- Each processor contains one full row of the matrix.
- The transpose operation in this case is identical to an all-to-all personalized communication operation.

All-to-All Personalized Communication: Example



All-to-all personalized communication in transposing a *4 x 4* matrix using four processes.

All-to-All Personalized Communication on a Ring

- Each node sends all pieces of data as one *consolidated message* of size $m(p-1)$ to one of its neighbors.
- Each node extracts the information meant for it from the data received and forwards the remaining $(p-2)$ pieces of size m each to the next node.
- The algorithm terminates in $p-1$ steps.
- The size of the message reduces by m at each step.

All-to-All Personalized Communication on a Ring: Cost

- We have *p – 1* steps in all.
- In step *i*, the message size is *m(p – i)*.
- The total time is given by:

$$\begin{aligned} T &= \sum_{i=1}^{p-1} (t_s + t_w m(p-i)) \\ &= t_s(p-1) + \sum_{i=1}^{p-1} i t_w m \\ &= (t_s + t_w mp/2)(p-1). \end{aligned}$$

- The t_w term in this equation can be reduced by a factor of 2 by communicating messages in both directions.

All-to-All Personalized Communication on a Mesh

- Each node first groups its p messages according to the columns of their destination nodes.
- All-to-all personalized communication is performed independently in each row with clustered messages of size $m\sqrt{p}$.
- Messages in each node are sorted again, this time according to the rows of their destination nodes.
- All-to-all personalized communication is performed independently in each column with clustered messages of size $m\sqrt{p}$.

All-to-All Personalized Communication on a Mesh: Cost

- Time for the first phase is identical to that in a ring with $\sqrt{p}$ processors, i.e., $(t_s + t_w mp/2)(\sqrt{p} - 1)$.
- Time in the second phase is identical to the first phase. Therefore, total time is twice of this time, i.e.,

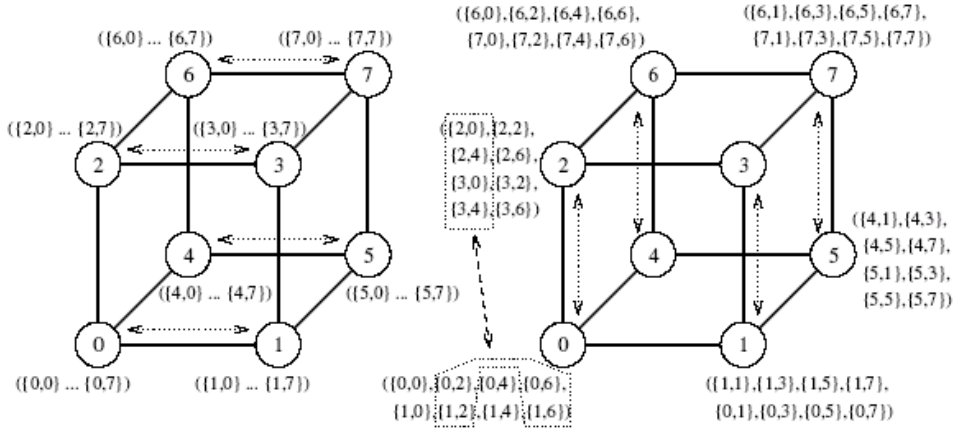
$$T = (2t_s + t_w mp)(\sqrt{p} - 1).$$

- It can be shown that the time for rearrangement is much less than this communication time.

All-to-All Personalized Communication on a Hypercube

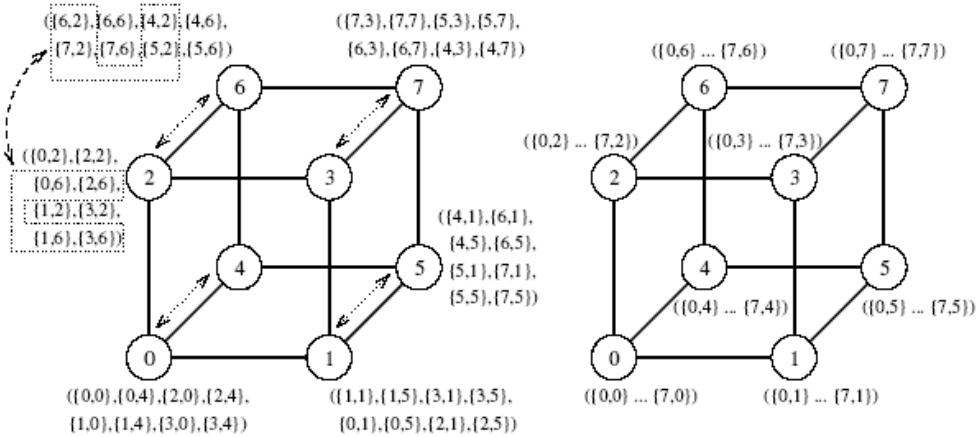
- Generalize the mesh algorithm to *log p* steps.
- At any stage in all-to-all personalized communication, every node holds *p* packets of size *m* each.
- While communicating in a particular dimension, every node sends *p/2* of these packets (consolidated as one message).
- A node must rearrange its messages locally before each of the *log p* communication steps.

All-to-All Personalized Communication on a Hypercube



(a) Initial distribution of messages

(b) Distribution before the second step



(c) Distribution before the third step

(d) Final distribution of messages

An all-to-all personalized communication algorithm on a three-dimensional hypercube.

All-to-All Personalized Communication on a Hypercube: Cost

- We have *log p* iterations and *mp/2* words are communicated in each iteration. Therefore, the cost is:

$$T = (t_s + t_w mp/2) \log p.$$

- Homework: This is not optimal! Why? Can you design an optimal one?

Circular Shift

- A special permutation in which node *i* sends a data packet to node *(i + q)* mod *p* in a *p*-node ensemble *(0 ≤ q ≤ p)*.

Circular Shift on a Mesh

- The implementation on a ring is rather intuitive. It can be performed in min*{q,p – q}* neighbor communications.
- Mesh algorithms follow from this as well. We shift in one direction (all processors) followed by the next direction.
- The associated time has an upper bound of:

$$T = (t_s + t_w m)(\sqrt{p} + 1).$$

Improving Performance of Operations

- Splitting and routing messages into parts: If the message can be split into *p* parts, a one-to-all broadcast can be implemented as a scatter operation followed by an all-to-all broadcast operation. The time for this is:

$$\begin{aligned} T &= 2 \times (t_s \log p + t_w (p-1)\frac{m}{p}) \\ &\approx 2 \times (t_s \log p + t_w m). \end{aligned}$$

- All-to-one reduction can be performed by performing all-to-all reduction (dual of all-to-all broadcast) followed by a gather operation (dual of scatter).

Matix Algorithms: Introduction

- Due to their regular structure, parallel computations involving matrices and vectors readily lend themselves to data-decomposition.
- Typical algorithms rely on input, output, or intermediate data decomposition.
- Most algorithms use one- and two-dimensional block, cyclic, and block-cyclic partitionings.

Matrix-Vector Multiplication

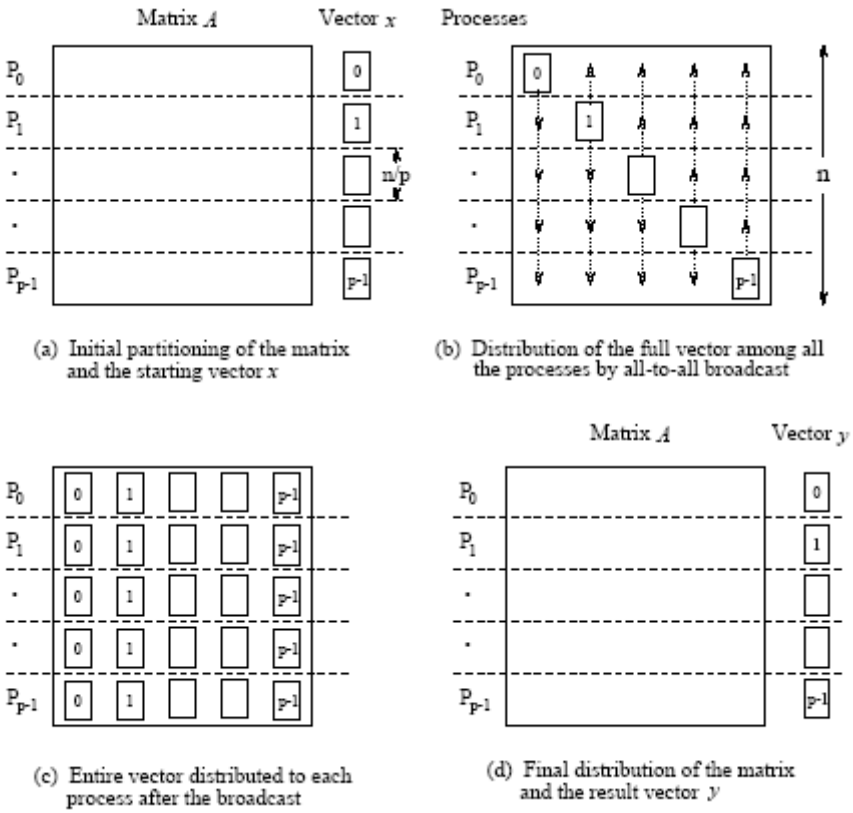
- We aim to multiply a dense n x n matrix A with an n x 1 vector x to yield the n x 1 result vector y.
- The serial algorithm requires n^2 multiplications and additions.

$$W = n^2.$$

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

- The n x n matrix is partitioned among n processors, with each processor storing complete row of the matrix.
- The n x 1 vector x is distributed such that each process owns one of its elements.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning



Multiplication of an n x n matrix with an n x 1 vector using rowwise block 1-D partitioning. For the one-row-per-process case, $p = n$.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

- Since each process starts with only one element of x, an all-to-all broadcast is required to distribute all the elements to all the processes.
- Process P_i now computes $y[i] = \Sigma_{j=0}^{n-1}(A[i,j] \times x[j])$.
- The all-to-all broadcast and the computation of $y[i]$ both take time $\Theta(n)$. Therefore, the parallel time is $\Theta(n)$.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

- Consider now the case when $p < n$ and we use block 1D partitioning.
- Each process initially stores n/p complete rows of the matrix and a portion of the vector of size n/p.
- The all-to-all broadcast takes place among p processes and involves messages of size n/p.
- This is followed by n/p local dot products.
- Thus, the parallel run time of this procedure is

$$T_P = \frac{n^2}{p} + t_s \log p + t_w n.$$

This is cost-optimal.

Matrix-Vector Multiplication: Rowwise 1-D Partitioning

Scalability Analysis:

- We know that $T_0 = pT_P$ - W, therefore, we have,

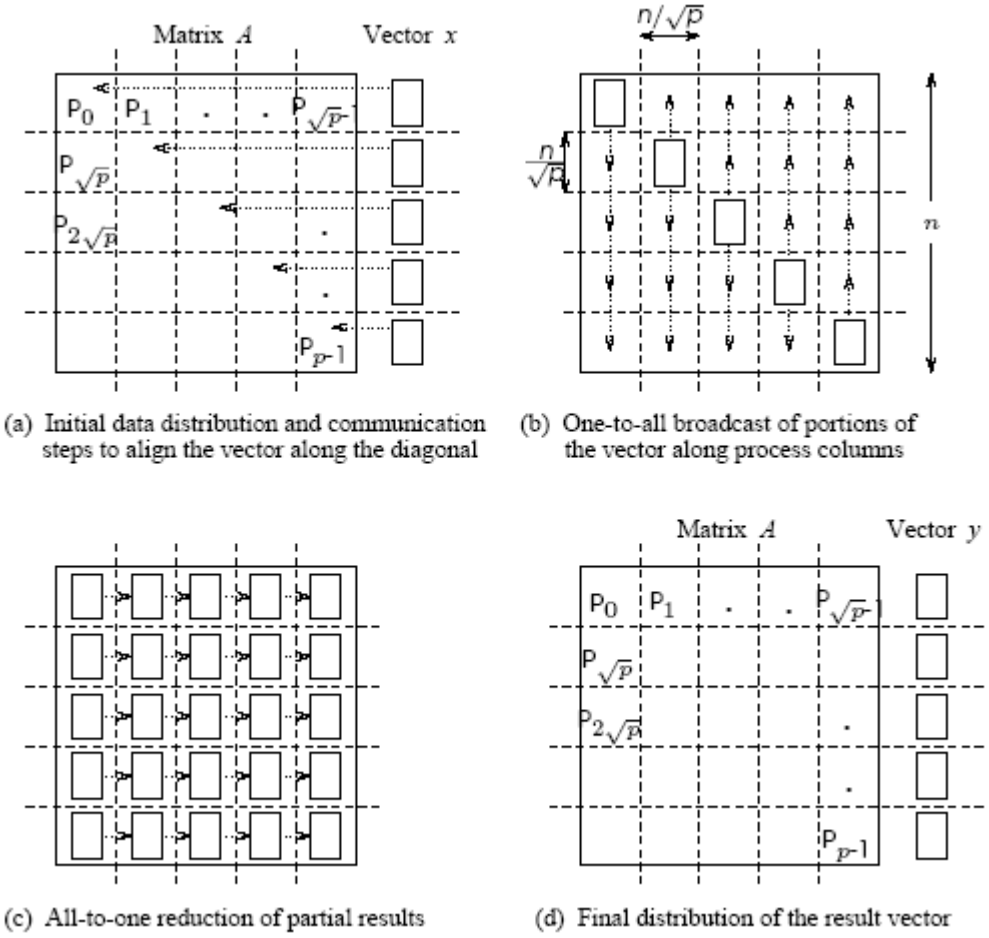
$$T_o = t_s p \log p + t_w n p.$$

- For isoefficiency, we have $W = KT_0$, where $K = E/(1 - E)$ for desired efficiency E.
- From this, we have $W = O(p^2)$ (from the t_w term).
- There is also a bound on isoefficiency because of concurrency. In this case, $p < n$, therefore, $W = n^2 = \Omega(p^2)$.
- Overall isoefficiency is $W = O(p^2)$.

Matrix-Vector Multiplication: 2-D Partitioning

- The n x n matrix is partitioned among n^2 processors such that each processor owns a single element.
- The n x 1 vector x is distributed only in the last column of n processors.

Matrix-Vector Multiplication: 2-D Partitioning



(a) Initial data distribution and communication steps to align the vector along the diagonal

(b) One-to-all broadcast of portions of the vector along process columns

(c) All-to-one reduction of partial results

(d) Final distribution of the result vector

Matrix-vector multiplication with block 2-D partitioning. For the one-element-per-process case, $p = n^2$ if the matrix size is n x n .

Matrix-Vector Multiplication: 2-D Partitioning

- We must first align the vector with the matrix appropriately.
- The first communication step for the 2-D partitioning aligns the vector x along the principal diagonal of the matrix.
- The second step copies the vector elements from each diagonal process to all the processes in the corresponding column using n simultaneous broadcasts among all processors in the column.
- Finally, the result vector is computed by performing an all-to-one reduction along the columns.

Matrix-Vector Multiplication: 2-D Partitioning

- Three basic communication operations are used in this algorithm: one-to-one communication to align the vector along the main diagonal, one-to-all broadcast of each vector element among the n processes of each column, and all-to-one reduction in each row.
- Each of these operations takes $\Theta(\log n)$ time and the parallel time is $\Theta(\log n)$.
- The cost (process-time product) is $\Theta(n^2 \log n)$; hence, the algorithm is not cost-optimal.

Matrix-Vector Multiplication: 2-D Partitioning

- When using fewer than n^2 processors, each process owns an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of the matrix.
- The vector is distributed in portions of $n/\sqrt{p}$ elements in the last process-column only.
- In this case, the message sizes for the alignment, broadcast, and reduction are all $n/\sqrt{p}$.
- The computation is a product of an $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrix with a vector of length $n/\sqrt{p}$.

Matrix-Vector Multiplication: 2-D Partitioning

- The first alignment step takes time

$$t_s + t_w n/\sqrt{p}$$

- The broadcast and reductions take time

$$(t_s + t_w n/\sqrt{p}) \log(\sqrt{p})$$

- Local matrix-vector products take time

$$t_c n^2/p$$

- Total time is

$$T_P \approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p$$

Matrix-Vector Multiplication: 2-D Partitioning

- Scalability Analysis:

- $T_o = pT_p - W = t_s p \log p + t_w n\sqrt{p} \log p$
- Equating T_0 with W, term by term, for isoefficiency, we have, $W = K^2 t_w^2 p \log^2 p$ as the dominant term.
- The isoefficiency due to concurrency is $O(p)$.
- The overall isoefficiency is $O(p \log^2 p)$ (due to the network bandwidth).
- For cost optimality, we have, $W = n^2 = p \log^2 p$. For this, we have, $p = O\left(\frac{n^2}{\log^2 n}\right)$