# Chapter 1:

# Parallel algorithms: Fundamentals

## *Section 3:*

## *1.3 Communication Operations*

Parallel algorithms and so the processors and the software processes in parallel computers have to communicate to each other, i.e. they exchange data and control information during their operation. The effectiveness of the data exchange much affects the speed of the parallel algorithms. For all data exchanges a given time is needed for communication, thus it is very important to minimize the delays introduced by communication operations. In a lot of applications and architectures the correct implementation of communication operations becomes a key issue for the effective execution of the parallel algorithms.

### 1.3.1 Routing Mechanisms in Static Networks

In static networks (unless the network is fully connected) the **routing mechanism** determines what path the message has to take through the network in order to achieve its destination from the source processor. Routing mechanism gets the source and destination addresses from the processor sending the massage. In response it returns one or more paths through the network from the source to the destination.

Various solutions exist for routing. Routing mechanisms may be grouped into classes based on their parameters. Classification is given below.

Based on the **length of** the calculated **path**:

- **Minimal** routing mechanism selects one of the shortest paths, thus each link takes the message closer to the destination, but the solution may lead to congestion.
- **Nonminimal** routing mechanism may select a longer path in order to avoid congestion.

On the base of how the **network status** is taken into consideration:

- **Deterministic** routing mechanism selects a unique path, not using the sate information of the network. This may load the communication network unevenly.
- **Adaptive** routing scheme uses the network state information in order to avoid congestion and find a path that is less loaded.

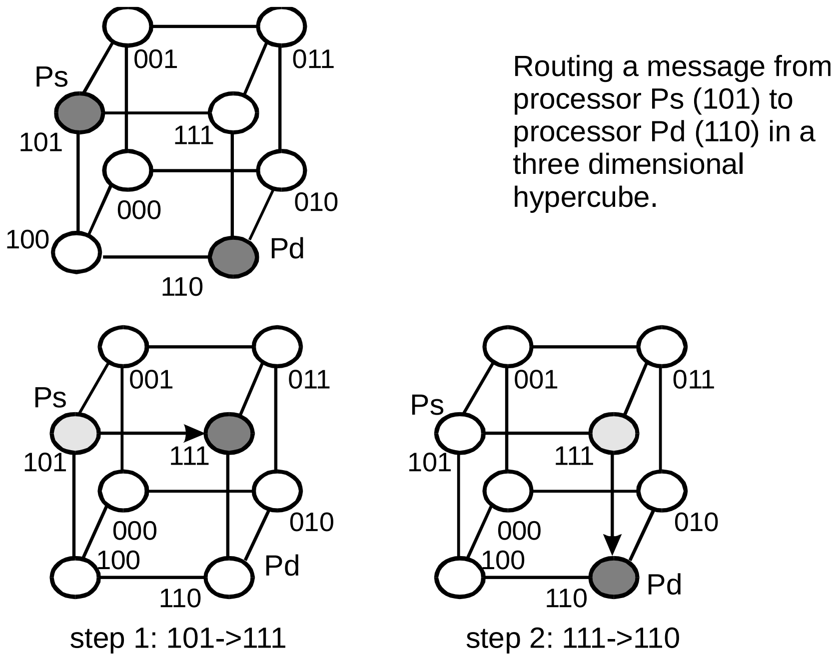



**Figure 1.3.1**

A popular routing scheme is the **dimension-ordered** one. It is deterministic and minimal. Based on the numbering scheme in the network it defines the path from one processor to the next and so on until the destination is reached.

For two-dimensional mesh architectures the dimension ordered routing mechanism is called **XY-routing,** while for a hypercube its name is **E-cube routing.**

*XY routing* sends the message first along the *x* coordinate until it reaches the destination *x* value, then along the *y* coordinate. The length of the path is $|S_x - D_x| + |S_y - D_y|$. *S* and *D* stand for source and destination processors. In the case of *E-cube routing* the bit differences have to be detected in the addresses of the processors. That is $S_d$ and $D_d$ values are XOR-ed:

$$S_d \oplus D_d$$

The number of 1-s gives the distance between the two processors. The message is sent along dimensions (one hop per each dimension), starting from the least significant (or most significant) bit (Figure 1.3.1).

## 1.3.2 Communication Cost in Static Networks

Processors in parallel computers are not only busy with the basic task of computing the algorithm but they do some more calculations due to organize the parallel job. Communication is the major source of the overhead. Communication introduces time delays. The time a message travels between two processors is called **communication latency.**

It depends on three basic parameters:

* **Startup time** *($t_s$):* The sending processor prepares the message. The time that is spent for this is the startup-time. Major components to this time are: putting together the message (data, header, trailer, error correction information), executing the routing algorithm and establishing interface to the next processor.
* **Per-hop time** *($t_h$):* The message travels from one processor to the next. The time that the header of the message spends in between two processors is the per-hop time. This is also called **node-latency.** Its value depends on the routing switch. This latency appears at each node where the message goes through.
* **Per-word transfer time** *($t_w$):* This time depends on the channel bandwidth. A channel can forward $r$ words per second, then $t_w = 1/r$.

## 1.3.3. Routing and Switching Techniques

**Store-and-forward routing (SF)**

The message traverses through multiple links, and each intermediate processor gets the message, stores it into its memory, establishes a link to the next processor and forwards it.

Let us now define the communication latency or **cost of communication** in store-and-forward networks assuming that the message of $m$ words-length goes through $l$ links. Each link adds per-hop time and the message length-time (the word transfer time):

$$t_{comm} = t_s + ( m\, t_w + t_h )\, l$$

If the per-hop time can be regarded negligible compared to the other parameters, we get:

$$t_{comm} = t_s + m\, t_w\, l$$

The disadvantage of this method is that it is slow (the message has to be fully received before it is forwarded). That is why it uses the communication resources inefficiently.

**Cut-Through Routing (CT)**

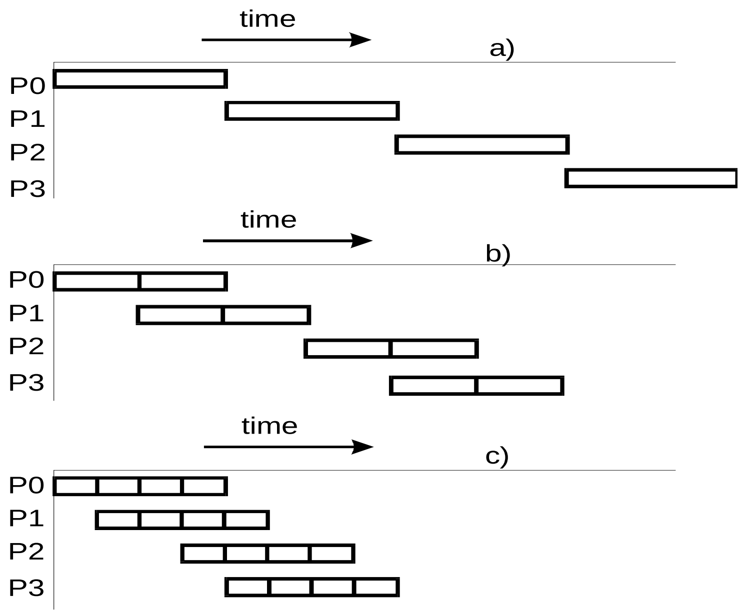
We may cut the messages into several smaller parts. In this case, there is no need to store the whole message in the intermediate processors and there is not that much delay until the whole message arrives and may be forwarded. If the message parts are small, then we may speak on **cut-through routing,** a message is forwarded immediately by the intermittent processors.

For the case of **wormhole routing** (cut-through routing) the small parts of the message are called **flow-control digits,** briefly **flits.** The flits are pipelined in the network; it looks like the whole message is waving through the path. Communication time may be defined as follows.

The source spends the *start-up time* for the message preparation; each link adds its *per-hop time*, however practically no time is needed for the store-and forward operations at the intermediate nodes. Finally the destination takes time to receive the whole message:

$$t_{comm} = t_s + t_h\, l + m\, t_w$$

Cut-through routing results in faster communication (Figure 1.3.2) and better utilization of the resources. However, there is a much higher probability of deadlocks (Figure 1.3.3).




Sending a message from processor P0 to P3
a.) through Store-and-Forward technique, then extending the concept to Cut-Through routing
b.) the message is broken into two parts
c.) the message is broken into four parts

**Figure 1.3.2**

## 1.3.4 Basic Communication Operations

Though processors in parallel computers communicate to each other in many different ways, yet we may attempt to find some basic communication categories, where the solutions may have some common methodologies.

The communication operations differ from each other in the ways:

- how many senders and receivers take part in the communication,
- the messages are originated from one or many processors,
- the destination of the message is a single processor or many processors,
- the technique is a simple message or various messages are just shifted among the processors.

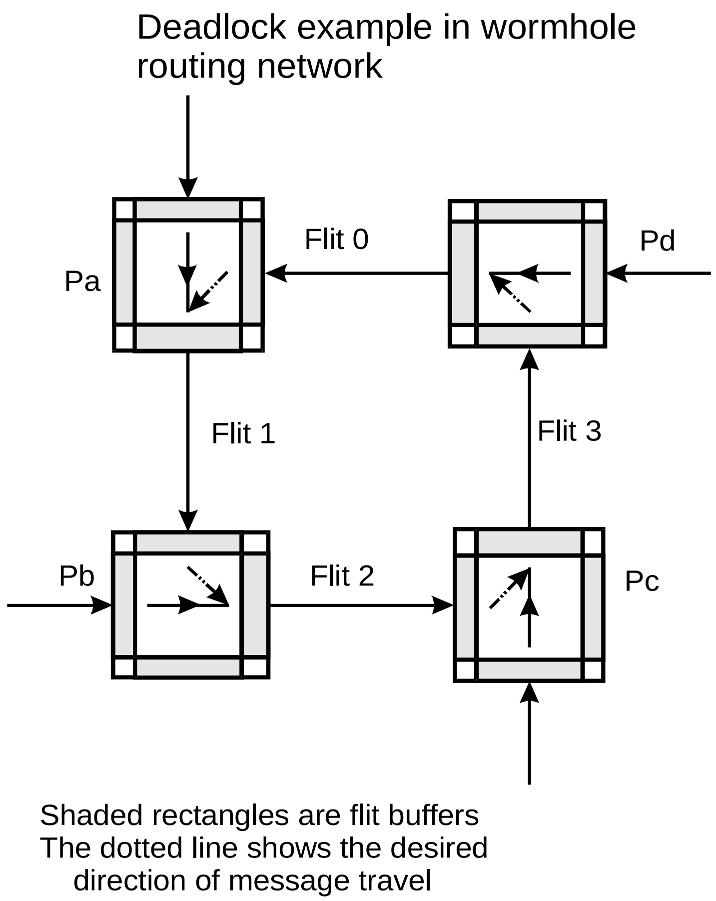



**Figure 1.3.3**

We may distinguish among the following basic classes based on the above-enlisted principles:

- simple message transfer between two processors,
- one-to-all broadcast
- all-to-all broadcast
- one-to-all personalized communication
- all-to-all personalized communication
- circular shift transmission.

## 1.3.5 Simple Message Transfer between Two Processors

This is the most frequently used operation. In the previous section we have described the time that is required when transferring simple messages. For **store and forward routing** Table 1.3.1 summarizes a few frequently used topologies, the maximum path lengths in hops in those topologies and the communication time supposing this maximum path length. When filling up the table we have taken the following equation for store and forward (SF) routing:

$$t_{comm} = t_s + m\, t_w\, l$$

For **cut-through routing** Table 1.3.2 summarizes the maximum message transfer length and transfer time data using the equation:

$$t_{comm} = t_s + m\, t_w + t_h\, l$$

**Table 1.3.1. Store-and-Forward routing**

| **Network type** | ***l max*** | **SF time max** |
|---|---|---|
| Ring | $p/2$ | $t_{comm} = t_s + m\, t_w\, p/2$ |
| 2-D torus | $\sqrt{p}$ | $t_{comm} = t_s + m\, t_w \sqrt{p}$ |
| Hypercube | $\log p$ | $t_{comm} = t_s + m\, t_w \log p$ |

**Table 1.3.2. Cut-Through routing**

| **Network type** | ***l max*** | **CT time max** |
|---|---|---|
| Ring | $p/2$ | $t_{comm} = t_s + m\, t_w + t_h\, p/2$ |
| 2-D torus | $\sqrt{p}$ | $t_{comm} = t_s + m\, t_w + t_h \sqrt{p}$ |
| Hypercube | $\log p$ | $t_{comm} = t_s + m\, t_w + t_h \log p$ |

In the case when the message is large enough, the distance between the two processors becomes unimportant in CT routing as to the value of the communication time. This time becomes approximately the same as in the case of directly connected processors.

## 1.3.6 One-to-All Broadcast

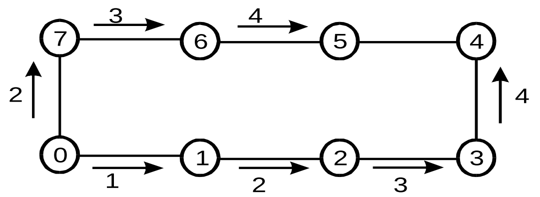
It is often required in parallel algorithms to send the same message to all (or many) other processors. This operation is called **one-to-all broadcast** or **single node broadcast.** Also, often a parallel algorithm requires the accumulation of data coming from all processors in a single processor node. This latter operation is called **single-node accumulation**. In this case data are accumulated at the receiving node. The messages are of size *m*. Their content is accumulated using some function on the data, but the message size does not change during this operation. The two operations are so called **dual operations;** they are similar to each other, only the direction of the communication is different. These two operations find their application in matrix-vector multiplication, Gaussian elimination, vector inner product, and shortest path search algorithms.

Below we examine the *one-to-all broadcast* operation (the dual operation can be examined in a very similar way) in the case of *store-and-forward* and *cut-through* routing.

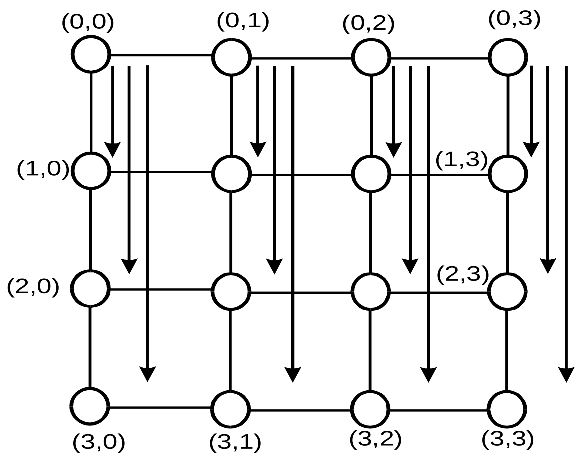
**One-to-all broadcast with SF routing**

The source processor sends its message to its neighbors, one at a time. The neighbors receive the message, store it (also for themselves) and then forward it further to their neighbors. The process differs in different network topologies.

We illustrate the process for the 1-D (ring) 2-D (torus) mesh in Figure 1.3.4 and for the 3-D hypercube in Figure 1.3.5. The time to conclude the operation is summed up in Table 1.3.3. Please note that the shortest time is obtained for the hypercube, because it has higher connectivity than the others do.




One-to-All broadcast with SF routing on an 8-processor ring




One-to-All broadcast in the multiplication of a 4x4 matrix with a 4x1 vector

**Figure 1.3.4**

**Table 1.3.3.** Communication times for the one-to-all broadcast using SF routing

| Network type | Communication time |
|---|---|
| Ring | $T_{onetoall} = (t_s + m\, t_w)\, p/2$ |
| 2-D torus | $T_{onetoall} = (t_s + m\, t_w)\, \sqrt{p}$ |
| Hypercube | $T_{onetoall} = (t_s + m\, t_w) \log p$ |

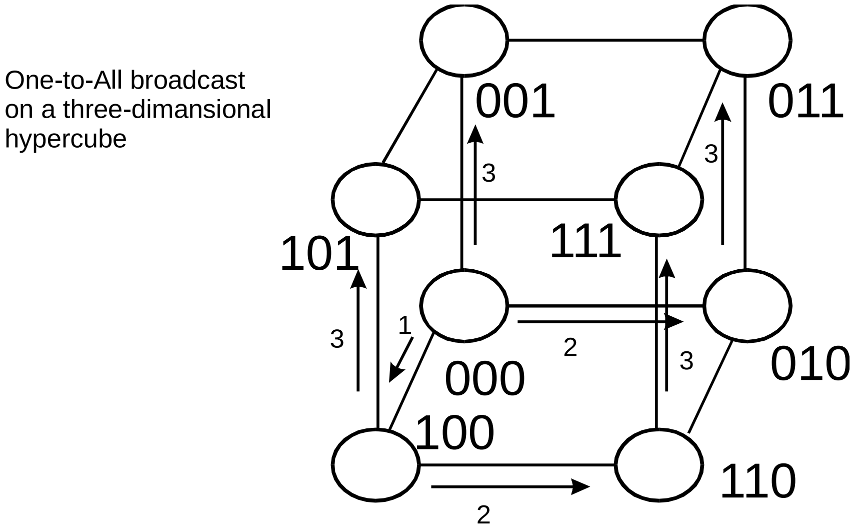



**Figure 1.3.5**

**One-to-all broadcast with CT routing**

The method naturally differs from the same method with SF routing in that the messages pass through faster, there is no need to wait for the whole message before starting to forward it (see in Figure 1.3.6. for a ring and a tree topology).

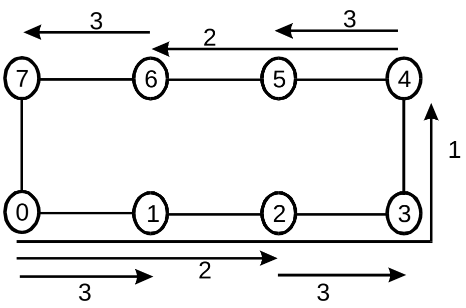
The time to conclude the operation is summed up in Table 1.3.4.

**Table 1.3.4.** Communication times for the one-to-all broadcast using CT routing

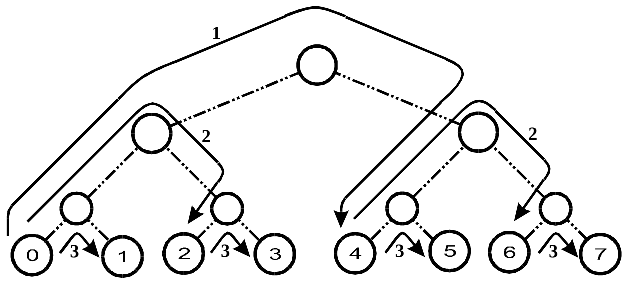
| Network type | Communication time |
|---|---|
| Ring | $T_{onetoall} = log\ p\ (t_s + m\ t_w) + t_h\ (p - 1)$ |
| 2-D torus | $T_{onetoall} = log\ p\ (t_s + m\ t_w) + t_h\ (\sqrt{p} - 1)$ |
| Hypercube | $T_{onetoall} = (t_s + m\ t_w + t_h\ (log\ p + 1))\ log\ p$ |

## 1.3.7 All-to-All Broadcast

**All-to-all broadcast** or **multinode broadcast** operation can be regarded as the generalization of the one-to-all broadcast. In this case all the processors simultaneously initialize a one-to-all broadcast operation. Each processor sends for all the other processors its message, but every processor has its own, message different from the others, to be sent. A way to do this operation is initializing *p* one-to-all broadcasts. This operation also has its dual one: the **multinode accumulation** operation, when all processors accumulate messages from all other processors.




One-to-All broadcast with CT routing on an 8-processor ring




One-to-All broadcast on an 8-processor tree

**Figure 1.3.6**

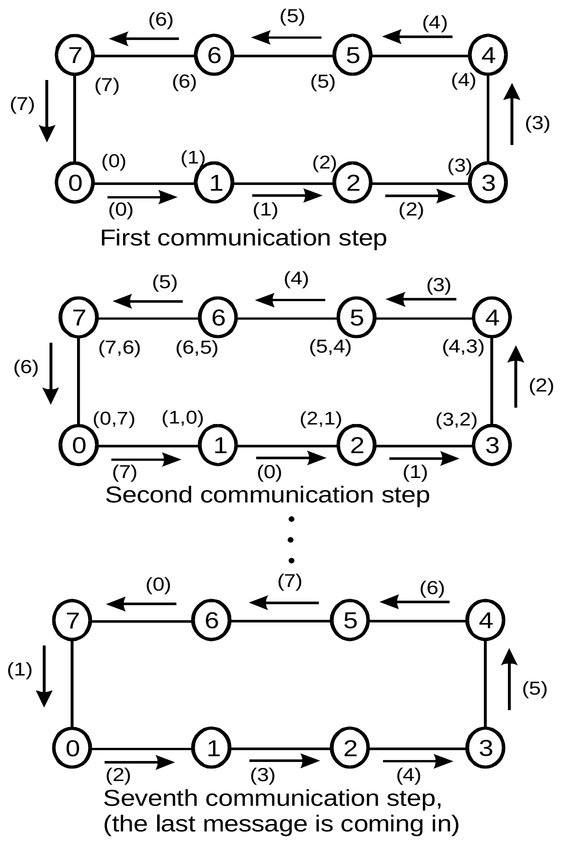
These operations may find their application in matrix operation, (matrix-matrix and matrix-vector multiplication), prefix sums or reduction operations.

In a 2-D mesh all processors send and receive messages first in one dimension (in the rows), then all messages obtained from this dimension are sent by all processors via the other dimension (in columns). A hypercube works in a similar way in *d* dimensions. Table 1.3.5 contains the timing data for this type of broadcasting.

**Table 1.3.5.** Communication times for the all-to-all broadcast using (SF routing)

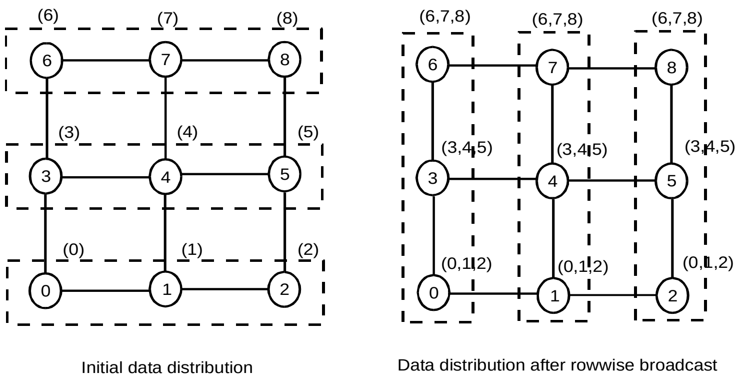
| Network type | Communication time |
|---|---|
| Ring | $T_{alltoall} = ( t_s + m\, t_w) ( p - 1 )$ |
| 2-D torus | $T_{alltoall} = 2\, t_s (\sqrt{p} - 1) + ( p - 1 )\, m\, t_w$ |
| Hypercube | $T_{alltoall} = t_s \log p + m\, t_w ( p - 1 )$ |

Figure 1.3.7 represents this type of broadcast operation for an 8-processor ring, Figure 1.3.8 for a 3 by 3 mesh, while Figure 1.3.9 for the hypercube.




All-to-All broadcast on on 8-processor ring

**Figure 1.3.7**




All-to-all bradcast on a mesh. Dotted boundaries show processors communicating with each other. By the end of the second phase all processors get (0,1,2,3,4,5,6,7), i.e. all messages

**Figure 1.3.8**

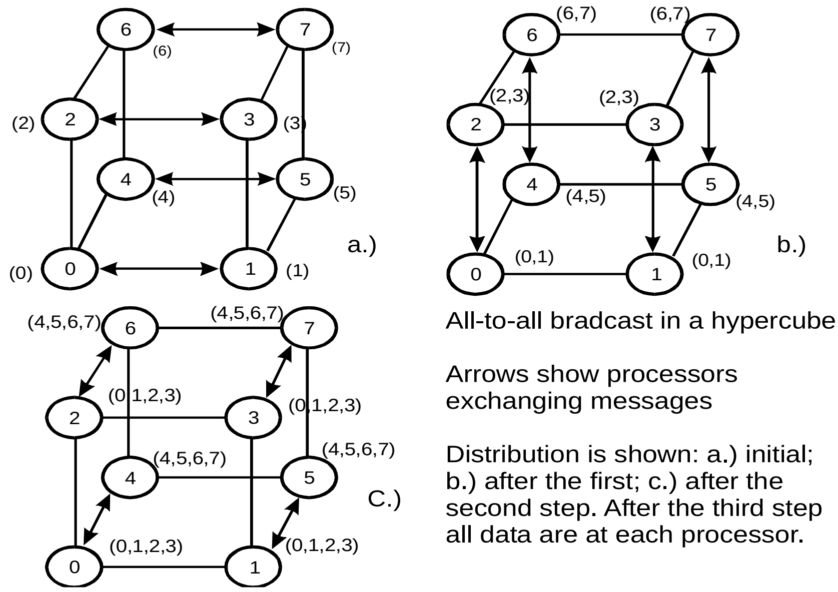



**Figure 1.3.9**

## 1.3.8 One-to-All Personalized Communication

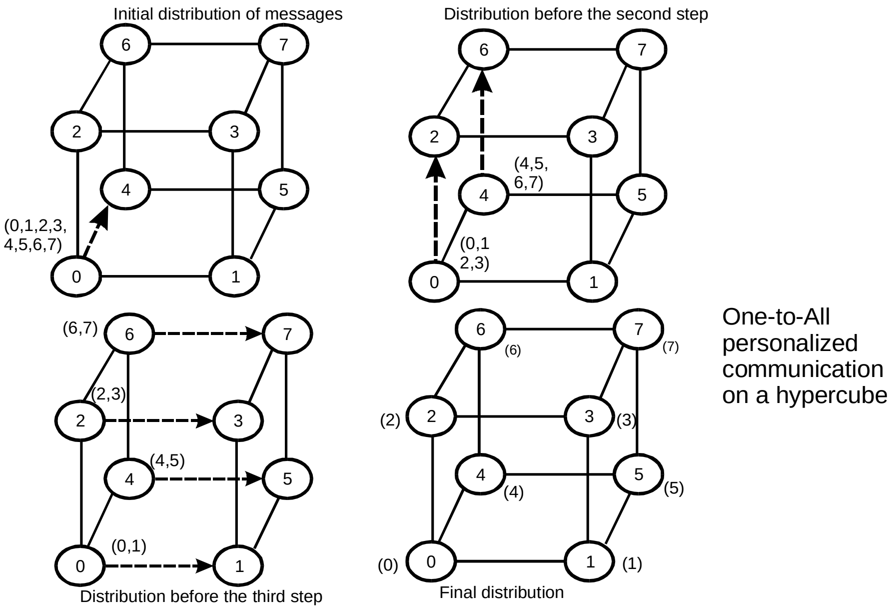



**Figure 1.3.10**

In this case a single node sends a unique message (size *m*) to each of the other processors. It is also called **single node scatter** operation. The source processor starts *p-1 unique* messages one destined to each processor (Figure 1.3.10.). This operation has its dual one called **Single node gather** operation. In the latter case one processor gathers *p-1* different messages from the other processors.

The communication complexity of this operation is similar to the *all-to-all broadcast.*

## 1.3.9. All-to-All Personalized Communication

This operation is also called **total exchange.** In the case of total exchange each processor sends a distinct message (size *m*) to every other processor. Messages are different to different processors.

Total exchange operation appears in algorithms like *Fast Fourier Transformation (FFT), Matrix transpose, and Parallel database* operations. The communication patterns are identical to those in *all-to-all broadcast* operations. However the content and the size of the messages are different.

In Table 1.3.5 the communication time estimation is summarized.

**Table 1.3.5. All-to-All personalized communication**

| **Network type** | **Communication time max** |
| --- | --- |
| Ring | $t_{alltoallper} = ( t_s + 1/2\ p\ m\ t_w ) (p - 1 )$ |
| 2-D torus | $t_{alltoallper} = ( 2\ t_s + p\ m\ t_w ) (\sqrt{p} - 1 )$ |
| Hypercube | $t_{alltoallper} = ( t_s + 1/2\ p\ m\ t_w ) \log p$ |

## 1.3.10. Circular shift

Circular shift belongs to the class of permutation operations. In a **circular q-shift** within a *p* processor parallel computer, processor *i* sends a data packet to the processor *(i+q) mod p.*

Applications for the circular shift operation are *matrix operations, string matching, and image pattern-matching operation.*

The implementation for the circular shift operation may be very intuitive, very much depends on the *q* parameter (the shift parameter) and the architecture of the system.