Basic Communication Operations (Chapter 4)

Vivek Sarkar

Department of Computer Science
Rice University

vsarkar@cs.rice.edu

Outline

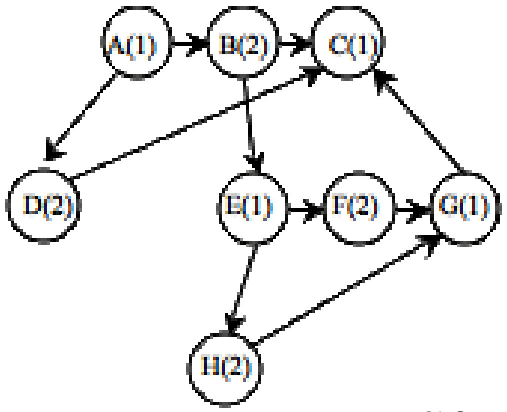
- Review of Midterm Exam
- MPI Example Program: Odd-Even Sorting (Section 6.3.5, Program 6.1)
- Basic Communication Operations (Chapter 4)
- **Acknowledgment:** today's lecture adapted from slides accompanying Chapters 4 and 9 of textbook
  - http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap4_slides.pdf
  - http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap9_slides.pdf

Midterm Exam Question #1

- Assume a computer system with a 1 GHz clock, a cache with 10ns access time, and DRAM with 100ns access time. Consider an application that accesses one 4-byte word per cycle with data hit rates of 90% and 80% for cache sizes of 32KB and 8KB respectively.
  a. What is the average ***latency*** (in nanoseconds per access) of the data accesses performed by the application, assuming a 32KB cache? Note that the latencies of a cache hit and miss are 10ns and 100ns respectively.
  b. Assuming that the latency can be completely overlapped (e.g., by prefetching), what total memory ***bandwidth*** (in bytes/s) is required to support 4 threads executing independent instances of the application on 4 CPUs (1 thread per CPU) with 8KB cache per thread?

Midterm Exam Question #2

- Consider the task graph in the figure below. Each node is labeled with the task's name and execution time e.g., B(2) refers to task B with an execution time of 2 units.
  a. Calculate the total work (T_1) and critical path length (T_∞) for this task graph.
  b. Show a 2-processor schedule for this task graph, and confirm that the completion time for your schedule, T_2, is in the range $\max(T_1/P, T_\infty) <= T_P < T_1/P + T_\infty$, for P=2.
  c. Write a Cilk program that can generate this task graph.



Midterm Exam Question #3

Consider the sequential and OpenMP versions of a matrix multiply computation shown below.

a. Is the OpenMP version a correct parallelization of the sequential version? If so, why? If not, why not?

b. Can adding static or dynamic schedule clauses to the for loops in the OpenMP version change your answer to part a? If so, how?

Sequential version:

```
for ( I = 0 ; I < N ; I++)
  for ( J = 0 ; J < N ; J++) C[I,J] = 0;
for ( I = 0 ; I < N ; I++)
  for ( J = 0 ; J < N ; J++)
    for ( K = 0 ; K < N ; K++) C[I,J] += A[I,K] * B[K,J];
```

OpenMP version:

```
#pragma omp parallel
{
  #pragma omp for nowait
  for ( I = 0 ; I < N ; I++)
    for ( J = 0 ; J < N ; J++) C[I,J] = 0;
  #pragma omp for
  for ( I = 0 ; I < N ; I++)
    for ( J = 0 ; J < N ; J++)
      for ( K = 0 ; K < N ; K++) C[I,J] += A[I,K] * B[K,J];
} // implicit barrier at end of #pragma omp parallel
```

Outline

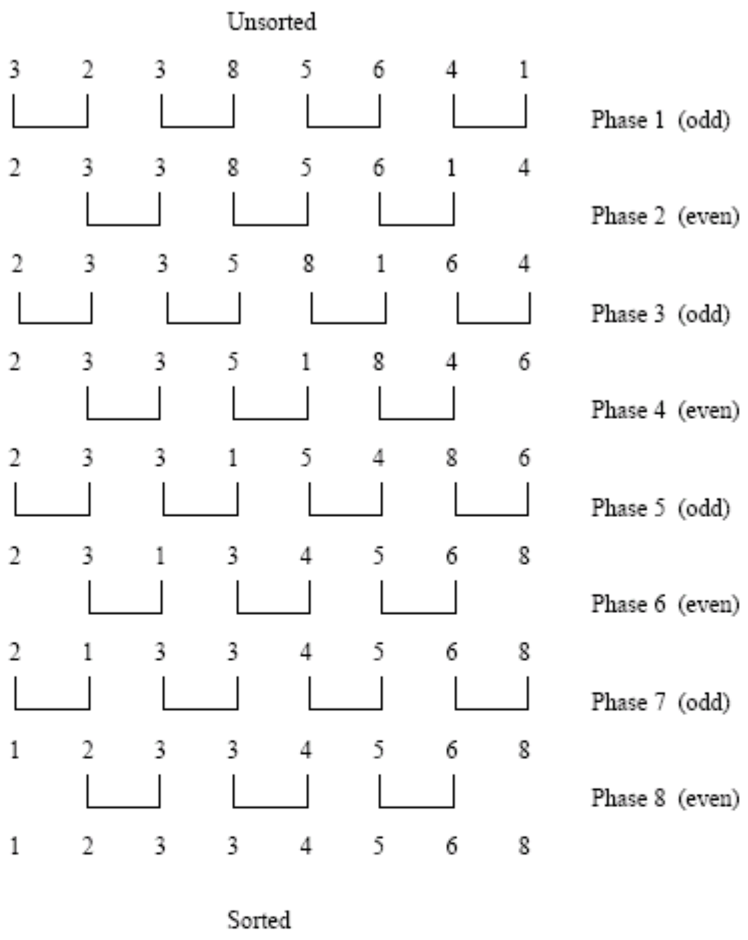
- Review of Midterm Exam
- <u>MPI Example Program: Odd-Even Sorting</u> (Section 6.3.5, Program 6.1)
- Basic Communication Operations (Chapter 4)

Odd-Even Transposition

```
procedure ODD-EVEN($n$)
begin
    for $i := 1$ to $n$ do
    begin
        if $i$ is odd then
            for $j := 0$ to $n/2 - 1$ do
                compare-exchange($a_{2j+1}, a_{2j+2}$);
        if $i$ is even then
            for $j := 1$ to $n/2 - 1$ do
                compare-exchange($a_{2j}, a_{2j+1}$);
    end for
end ODD-EVEN
```

Sequential odd-even transposition sort algorithm.

Odd-Even Transposition



Sorting n = 8 elements, using the odd-even transposition sort algorithm. During each phase, n = 8 elements are compared.

Odd-Even Transposition

- After n phases of odd-even exchanges, the sequence is sorted.
- Each phase of the algorithm (either odd or even) requires $\Theta(n)$ comparisons.
- Serial complexity is $\Theta(n^2)$.

Parallel Odd-Even Transposition

- Consider the one item per processor case.
- There are n iterations, in each iteration, each processor does one compare-exchange.
- The parallel run time of this formulation is $\Theta(n)$.
- This is cost optimal with respect to the base serial algorithm but not the optimal one.

Parallel Odd-Even Transposition

```
procedure ODD-EVEN_PAR($n$)
begin
    $id$ := process's label
    for $i$ := 1 to $n$ do
    begin
        if $i$ is odd then
            if $id$ is odd then
                compare-exchange_min($id + 1$);
            else
                compare-exchange_max($id - 1$);
        if $i$ is even then
            if $id$ is even then
                compare-exchange_min($id + 1$);
            else
                compare-exchange_max($id - 1$);
    end for
end ODD-EVEN_PAR
```

Parallel formulation of odd-even transposition.

Parallel Odd-Even Transposition

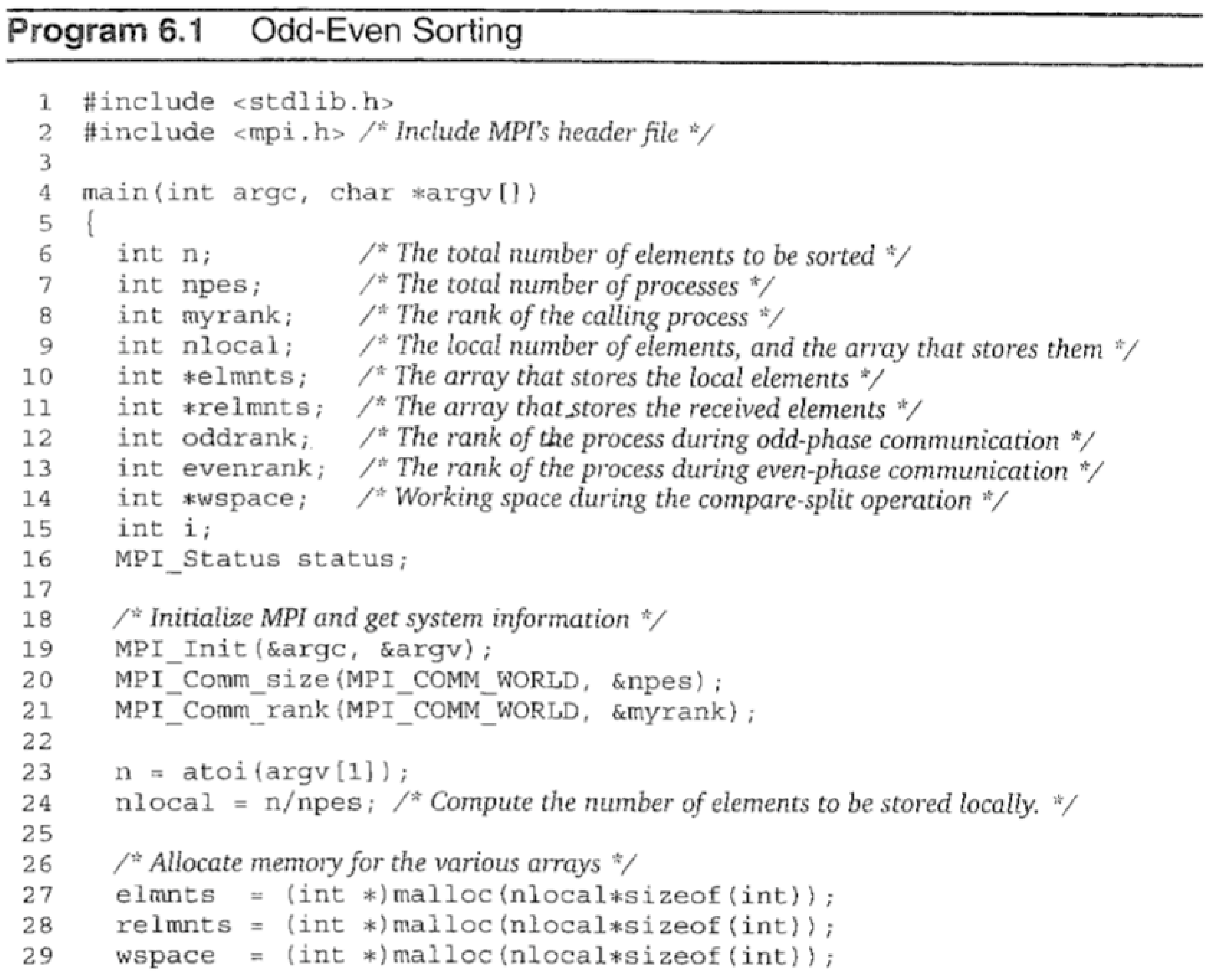
- Consider a block of n/p elements per processor.
- The first step is a local sort.
- In each subsequent step, the compare exchange operation is replaced by the compare split operation.
- The parallel run time of the formulation is

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta(n)}^{\text{comparisons}} + \overbrace{\Theta(n)}^{\text{communication}}.$$

Parallel Odd-Even Transposition

- The parallel formulation is cost-optimal for $p = O(\log n)$.
- The isoefficiency function of this parallel formulation is $\Theta(p2^p)$.

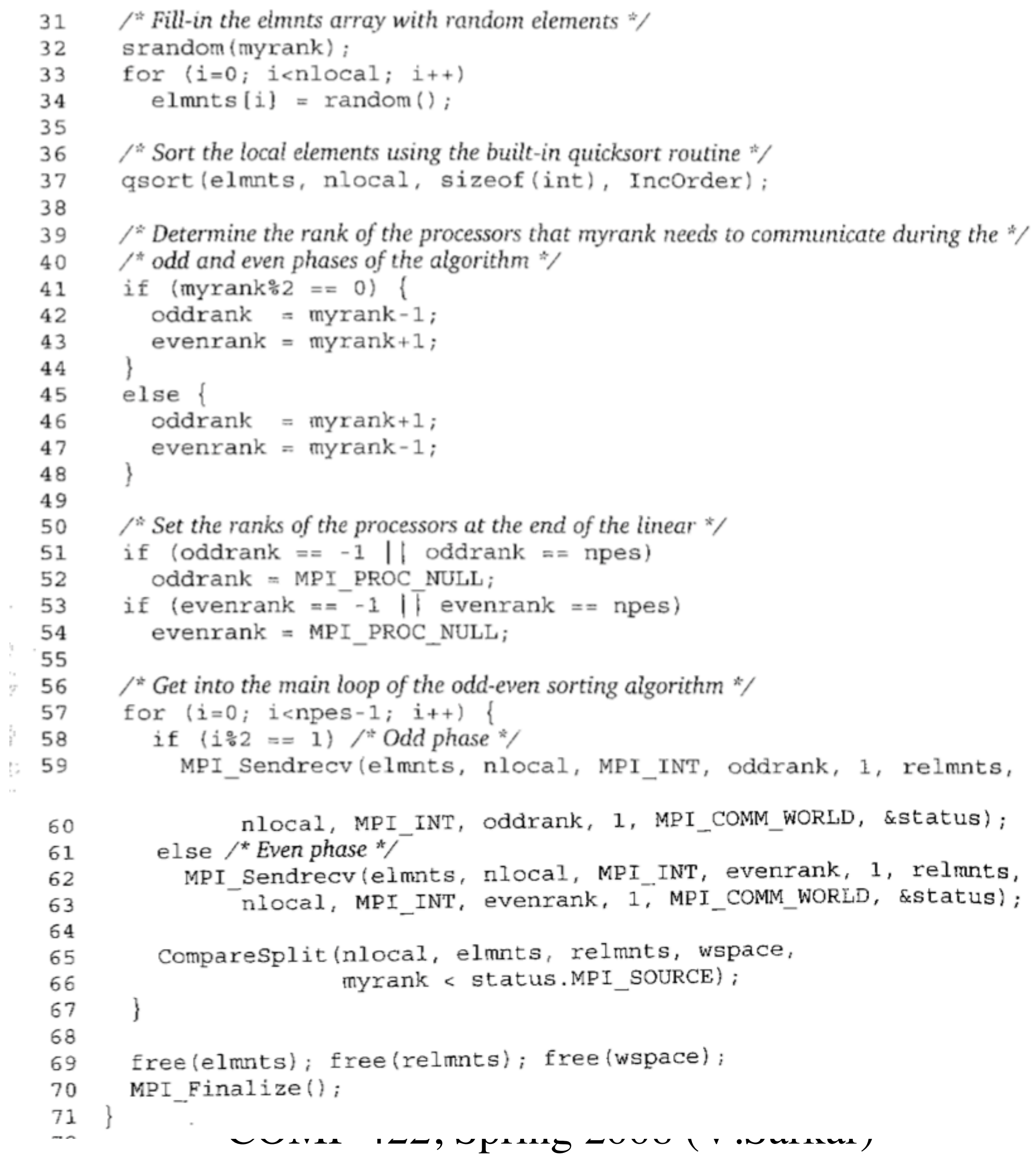
Program 6.1: Odd-Even Sorting



Program 6.1 Odd-Even Sorting

```
#include <stdlib.h>
#include <mpi.h> /* Include MPI's header file */

main(int argc, char *argv[])
{
  int n;           /* The total number of elements to be sorted */
  int npes;        /* The total number of processes */
  int myrank;      /* The rank of the calling process */
  int nlocal;      /* The local number of elements, and the array that stores them */
  int *elmnts;     /* The array that stores the local elements */
  int *relmnts;    /* The array that stores the received elements */
  int oddrank;     /* The rank of the process during odd-phase communication */
  int evenrank;    /* The rank of the process during even-phase communication */
  int *wspace;     /* Working space during the compare-split operation */
  int i;
  MPI_Status status;

  /* Initialize MPI and get system information */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  n = atoi(argv[1]);
  nlocal = n/npes; /* Compute the number of elements to be stored locally. */

  /* Allocate memory for the various arrays */
  elmnts  = (int *)malloc(nlocal*sizeof(int));
  relmnts = (int *)malloc(nlocal*sizeof(int));
  wspace  = (int *)malloc(nlocal*sizeof(int));
```

Program 6.1: Odd-Even Sorting (contd)



```
    /* Fill-in the elmnts array with random elements */
    srandom(myrank);
    for (i=0; i<nlocal; i++)
      elmnts[i] = random();

    /* Sort the local elements using the built-in quicksort routine */
    qsort(elmnts, nlocal, sizeof(int), IncOrder);

    /* Determine the rank of the processors that myrank needs to communicate during the */
    /* odd and even phases of the algorithm */
    if (myrank%2 == 0) {
      oddrank  = myrank-1;
      evenrank = myrank+1;
    }
    else {
      oddrank  = myrank+1;
      evenrank = myrank-1;
    }

    /* Set the ranks of the processors at the end of the linear */
    if (oddrank == -1 || oddrank == npes)
      oddrank = MPI_PROC_NULL;
    if (evenrank == -1 || evenrank == npes)
      evenrank = MPI_PROC_NULL;

    /* Get into the main loop of the odd-even sorting algorithm */
    for (i=0; i<npes-1; i++) {
      if (i%2 == 1) /* Odd phase */
        MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
            nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
      else /* Even phase */
        MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
            nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);

      CompareSplit(nlocal, elmnts, relmnts, wspace,
                   myrank < status.MPI_SOURCE);
    }

    free(elmnts); free(relmnts); free(wspace);
    MPI_Finalize();
}
```

Program 6.1: Odd-Even Sorting (contd)

```
/* This is the CompareSplit function */
CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace,
             int keepsmall)
{
  int i, j, k;

  for (i=0; i<nlocal; i++)
    wspace[i] = elmnts[i];   /* Copy the elmnts array into the wspace array */

  if (keepsmall) { /* Keep the nlocal smaller elements */
    for (i=j=k=0; k<nlocal; k++) {
      if (j == nlocal || (i < nlocal && wspace[i] < relmnts[j]))
        elmnts[k] = wspace[i++];
      else
        elmnts[k] = relmnts[j++];
    }
  }
  else { /* Keep the nlocal larger elements */
    for (i=k=nlocal-1, j=nlocal-1; k>=0; k--) {
      if (j == 0 || (i >= 0 && wspace[i] >= relmnts[j]))
        elmnts[k] = wspace[i--];
      else
        elmnts[k] = relmnts[j--];
    }
  }
}
```

Outline

- Review of Midterm Exam
- MPI Example Program: Odd-Even Sorting (Section 6.3.5, Program 6.1)
- Basic Communication Operations (Chapter 4)

Summary of MPI Communication Routines from previous lecture

`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.
`MPI_Sendrecv`	Combines send & receive.
`MPI_Isend`	Non-blocking (immediate) send
`MPI_Irecv`	Non-blocking (immediate) receive
`MPI_Test`	Tests for completion of Isend/Irecv
`MPI_Wait`	Waits for completion of Isend/Irecv

Summary of MPI Collective Communication Operations from previous lecture

`MPI_Barrier`	Barrier.
`MPI_Bcast`	One-to-all broadcast.
`MPI_Reduce`	All-to-one reduction.
`MPI_Allreduce`	Combine all-to-one reduction w/ broadcast
`MPI_Scan`	Compute prefix-sums
`MPI_Gather` `MPI_Allgather`	Gather in one process or all processes
`MPI_Scatter`	Scatter

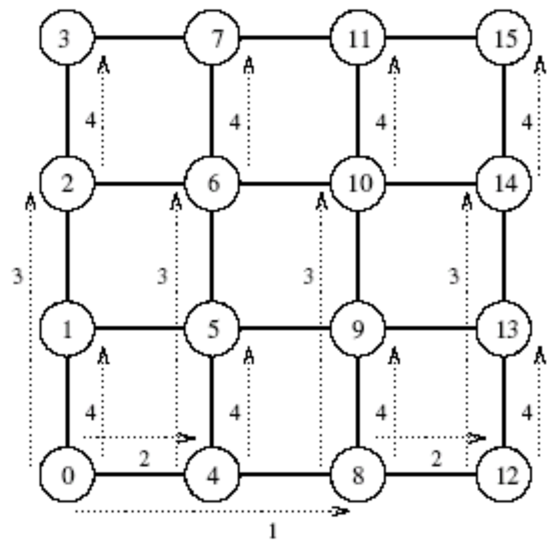
One-to-All Broadcast and All-to-One Reduction

- 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.

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



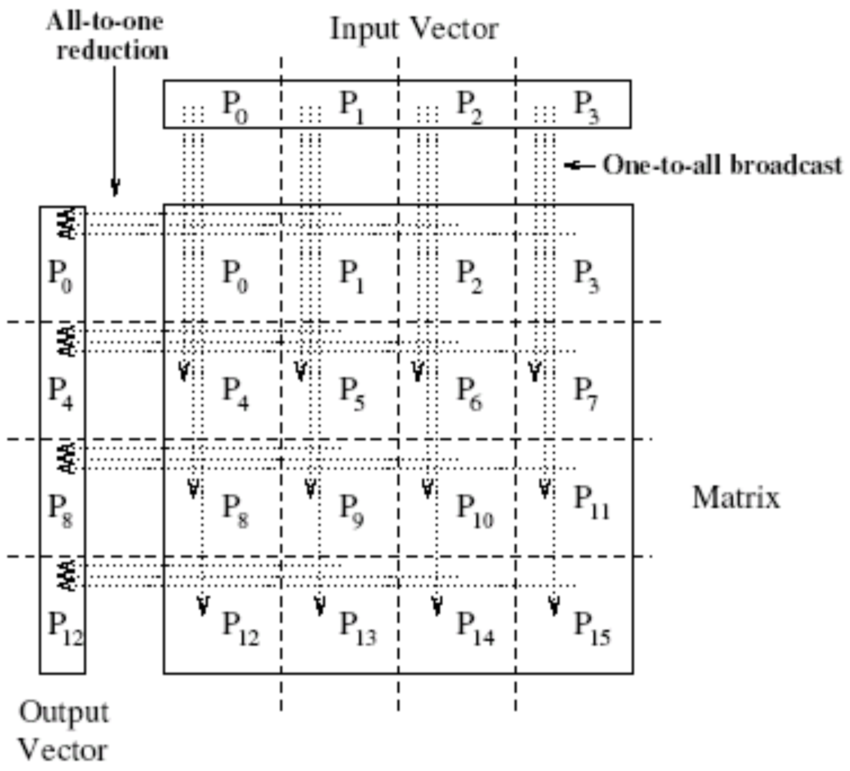
One-to-all broadcast on a 16-node mesh.

Broadcast and Reduction: 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

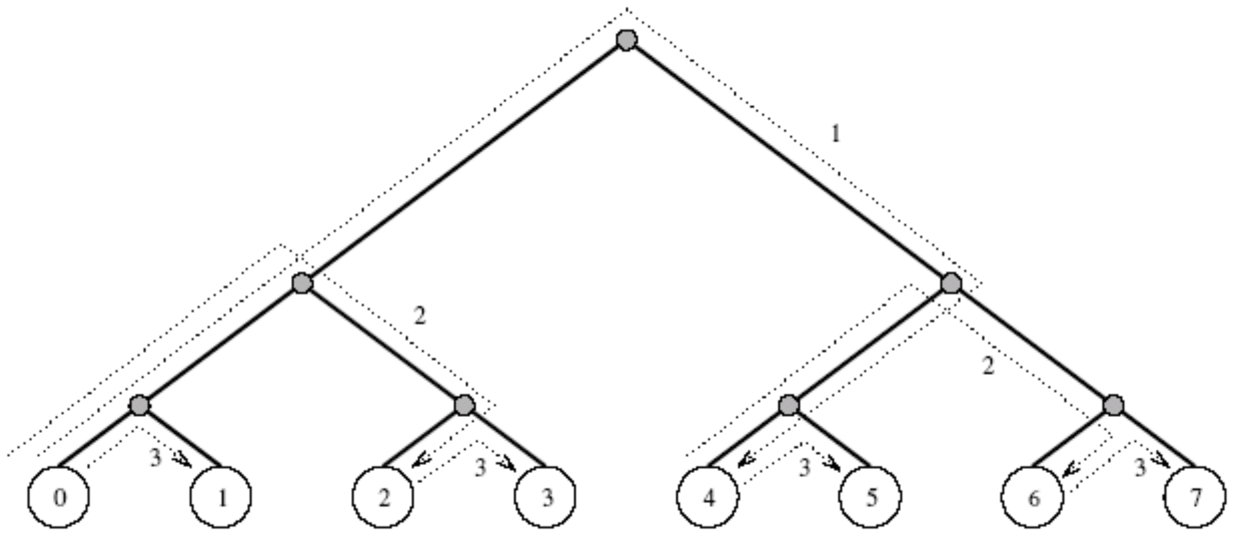


One-to-all broadcast and all-to-one reduction in the multiplication of a *4 x 4* matrix with a *4 x 1* vector.

Broadcast and Reduction on a 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.

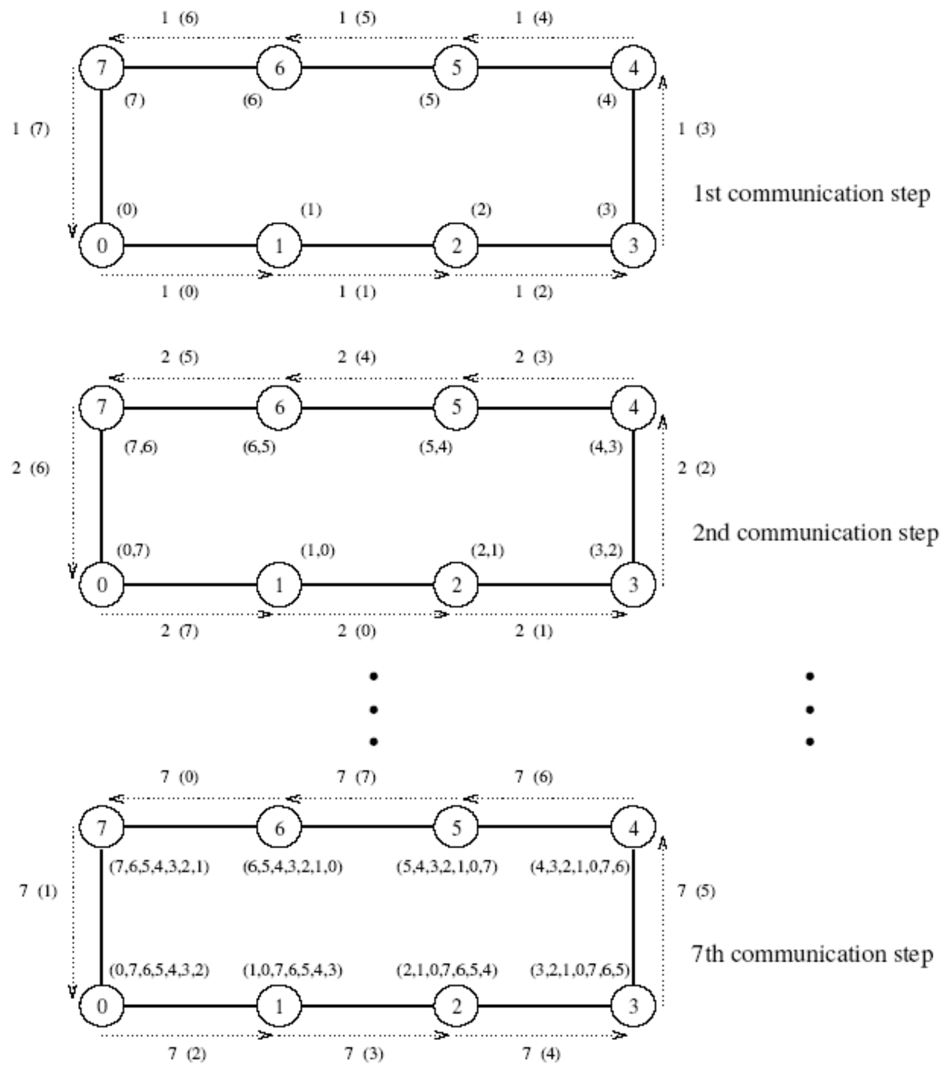
All-to-All Broadcast and Reduction

- 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 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

```
procedure ALL_TO_ALL_BC_RING($my\_id$, $my\_msg$, $p$, result)
begin
    left := $(my\_id - 1)$ mod $p$;
    right := $(my\_id + 1)$ mod $p$;
    result := $my\_msg$;
    msg := result;
    for $i := 1$ to $p - 1$ do
        send msg to right;
        receive msg from left;
        result := result $\cup$ msg;
    endfor;
end ALL_TO_ALL_BC_RING
```

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

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.

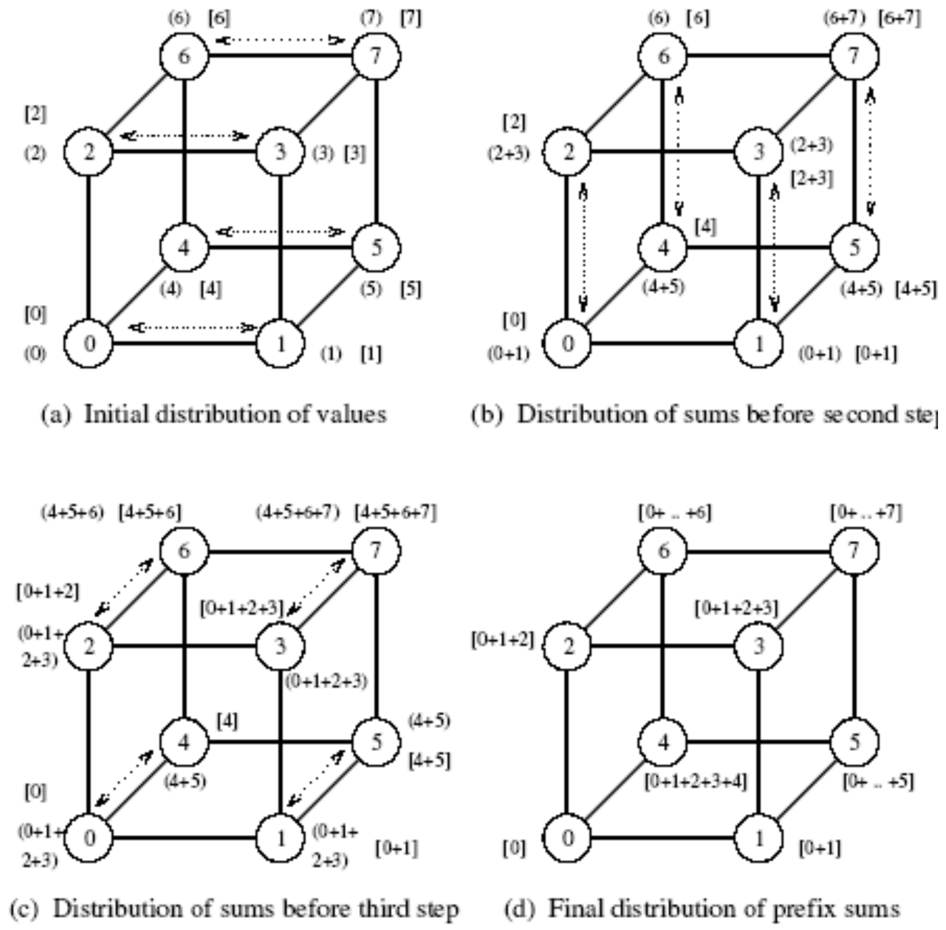
All-Reduce and Prefix-Sum Operations

- In all-reduce, each node starts with a buffer of size *m* and the final results of the operation are identical buffers of size *m* on each node that are formed by combining the original *p* buffers using an associative operator.
- Identical to all-to-one reduction followed by a one-to-all broadcast. This formulation is not the most efficient. Uses the pattern of all-to-all broadcast, instead. The only difference is that message size does not increase here. Time for this operation is $(t_s + t_w m) \log p$.
- Different from all-to-all reduction, in which *p* simultaneous all-to-one reductions take place, each with a different destination for the result.

The Prefix-Sum 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.

The Prefix-Sum Operation


(a) Initial distribution of values
(b) Distribution of sums before second step
(c) Distribution of sums before third step
(d) Final distribution of prefix sums

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

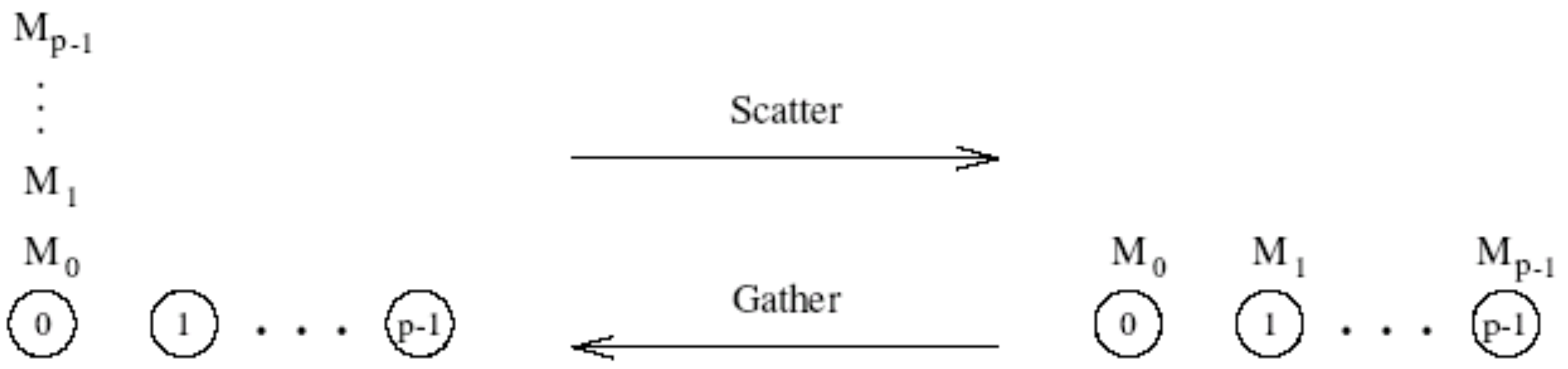
```
procedure PREFIX_SUMS_HCUBE($my\_id$, $my\_number$, $d$, result)
begin
    result := $my\_number$;
    msg := result;
    for $i := 0$ to $d - 1$ do
        partner := $my\_id$ XOR $2^i$;
        send msg to partner;
        receive number from partner;
        msg := msg + number;
        if (partner $< my\_id$) then result := result + number;
    endfor;
end PREFIX_SUMS_HCUBE
```

Prefix sums on a *d*-dimensional hypercube.

Scatter and Gather

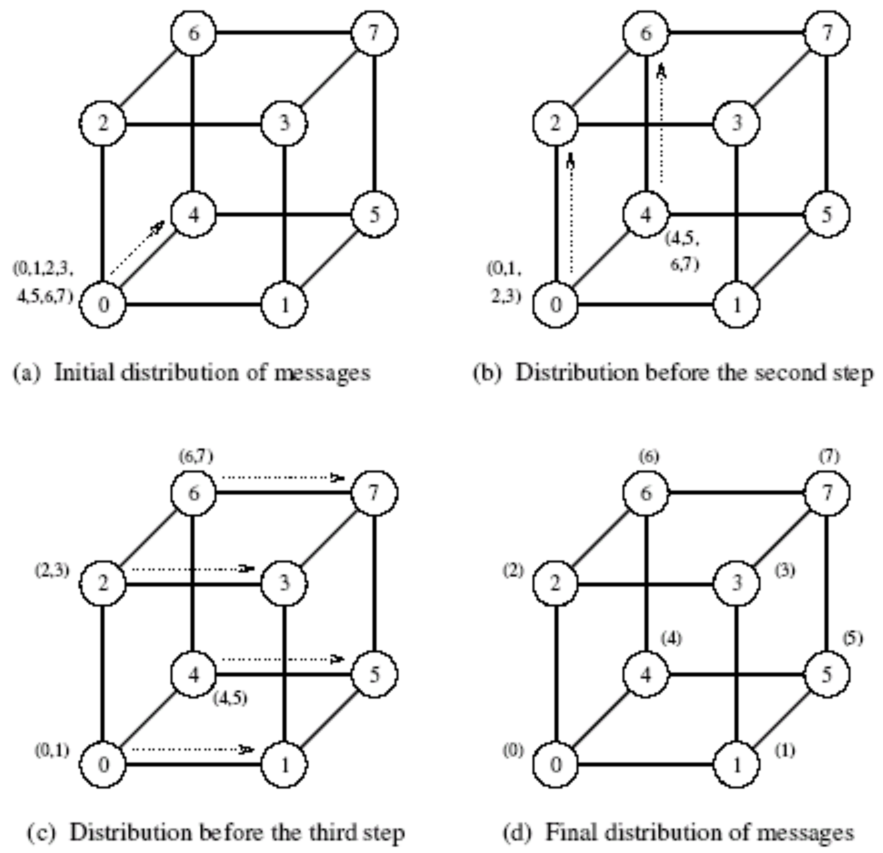
- In the *scatter* operation, a single node sends a unique message of size *m* to every other node (also called a one-to-all personalized communication).
- In the *gather* operation, a single node collects a unique message from each node.
- While the scatter operation is fundamentally different from 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



(0,1,2,3,
4,5,6,7)
(a) Initial distribution of messages
(0,1,
2,3)
(4,5,
6,7)
(b) Distribution before the second step
(6,7)
(2,3)
(4,5)
(0,1)
(c) Distribution before the third step
(6)
(7)
(2)
(3)
(4)
(5)
(0)
(1)
(d) Final distribution of messages

The scatter operation on an eight-node hypercube.