CSC 770
Parallel Computing

Eugene Adjeijan
Cleo Austein
Robert Kigobe

May 20th 2021

Parallel Sum by Divide and Conquer

- What is divide and conquer

- Divide and conquer usually applies partitioning in a recursive manner by continually dividing the problem into smaller and smaller parts (MPI-scatter) and combining the results (MPI-gather).

Divide And Conquer

Divide and conquer algorithms generally have 3 steps:

- Divide: Split the sequence into subproblems.
- Conquer: Recursively solve the subproblems
- Combine: The solutions of subproblems to create the solution to the original problem.
  - There are 3 possibilities: (1) the maximum sum lies completely in the left subproblem, (2) the maximum sum lies completely in the right subproblem, and (3) the maximum sum spans across the split point. The first two cases are easy. The more interesting case is when the largest sum goes between the two subproblems. The maximum subsequence that spans the middle is equal to the largest sum of a suffix on the left and the largest sum of a prefix on the right.

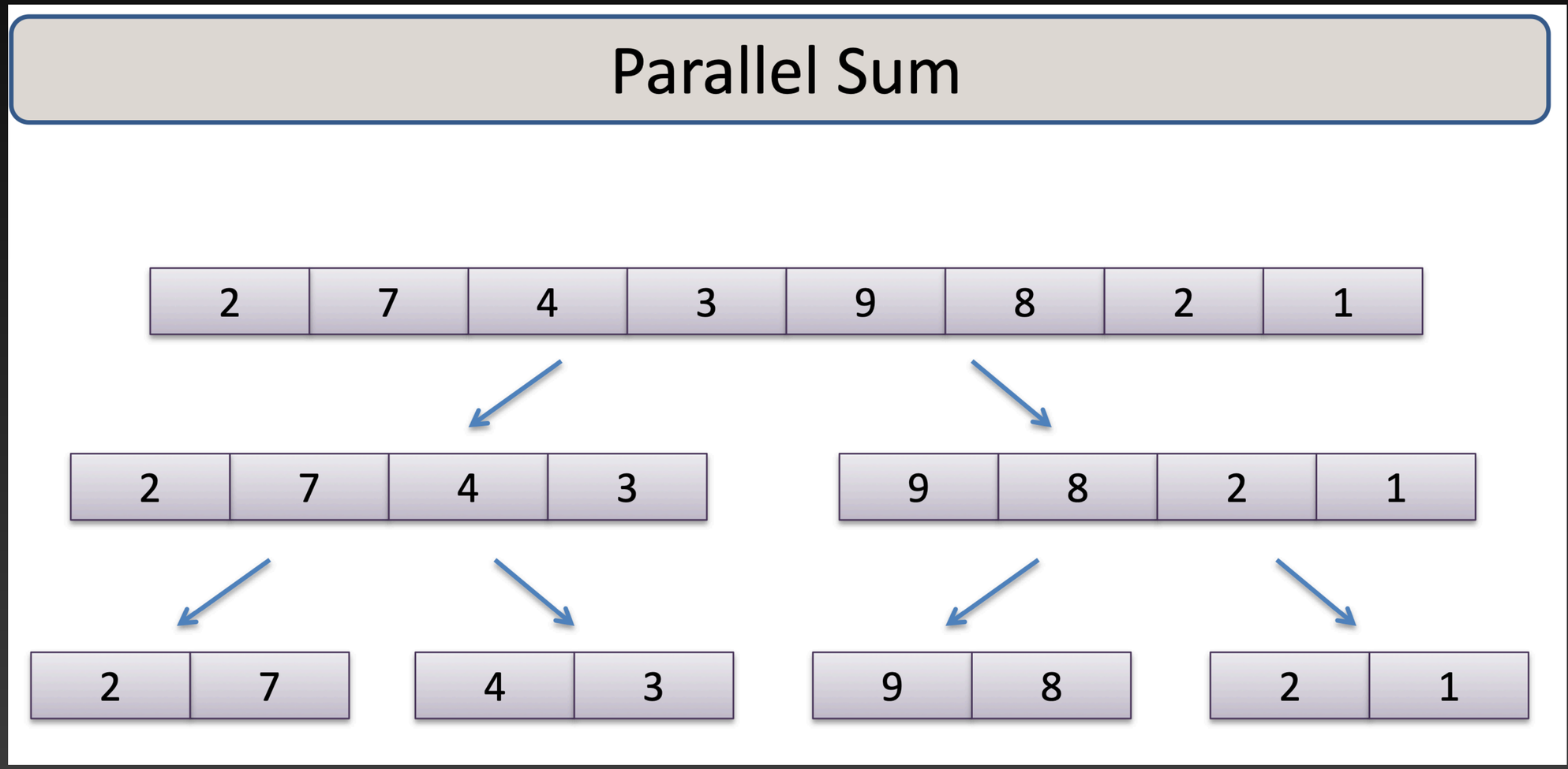
MPI Scatter and Gather

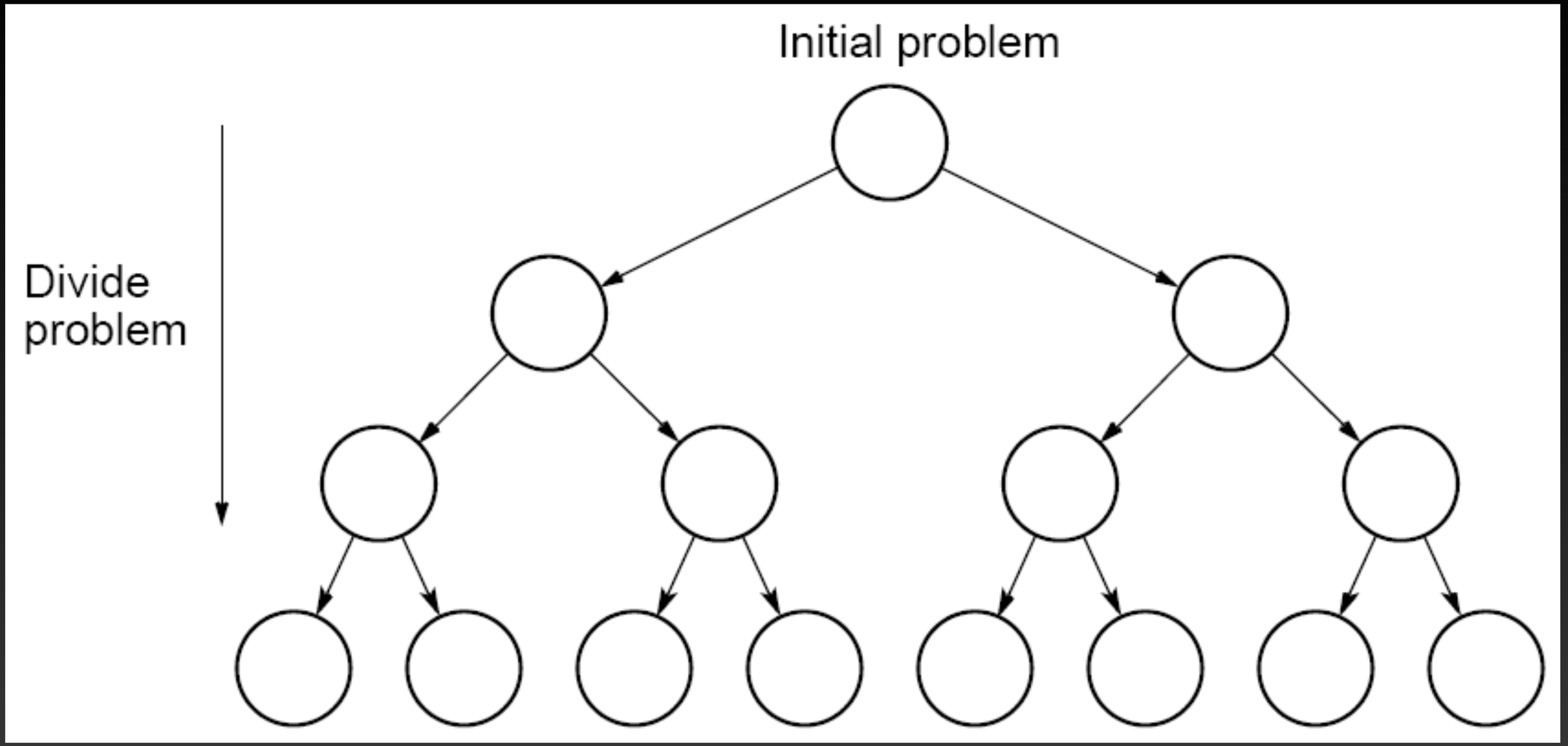
Two very important routines :

- Group functions that can manage the data distribution and collecting more easily.
- Scatter: involves a designated root process sending data to all processes in a communicator. Divides a big array into a number of smaller parts equal to the number of processes and sends each process (including the source) a piece of the array in rank order. MPI Scatter sends chunks of an array to different processes.
- Gather: MPI Gather is the inverse of MPI Scatter. Instead of spreading elements from one process to many processes, MPI Gather takes data stored from many processes (including the source or root) and gathers the elements to one single process and concatenates it in the receive array in rank order. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.

Divide into small portions

scatter: divides a big array into a number of smaller parts equal to the number of processes and sends each process (including the source) a piece of the array in rank order.



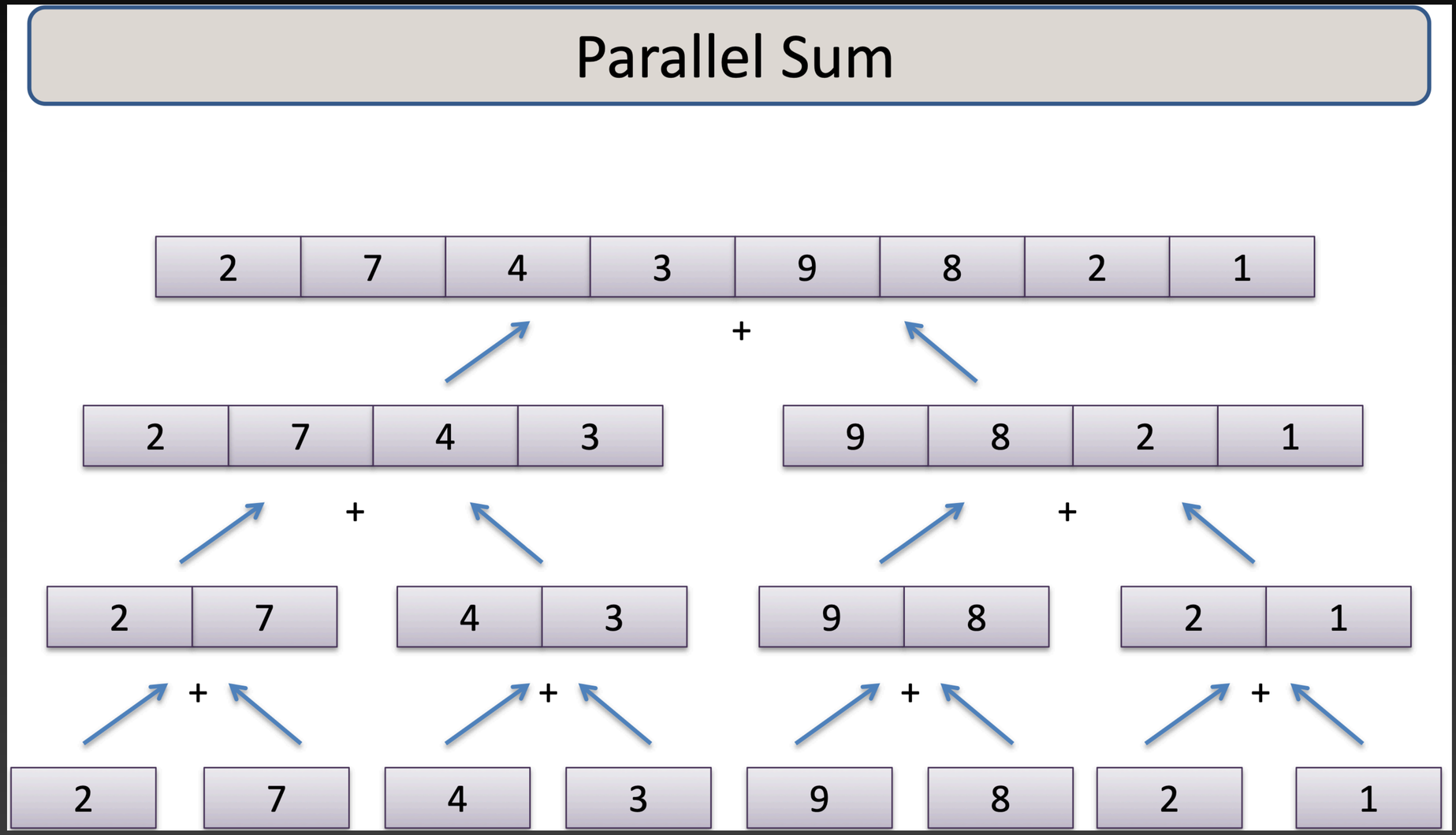

Initial problem
Divide problem

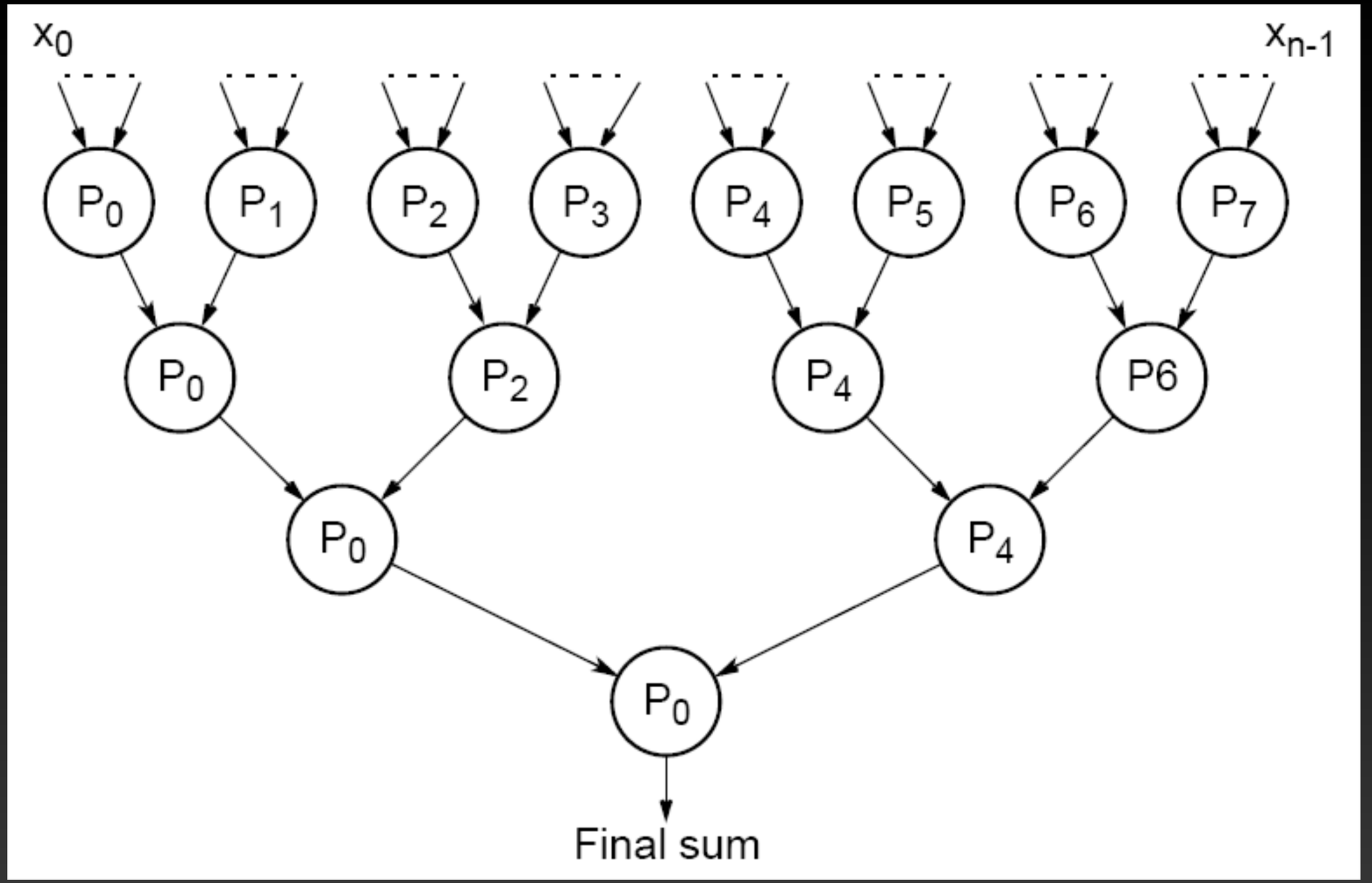
Code Snippet to scatter

```
for(int i = 1; i <= log2(size); i++){
    int dis = (int)pow(2, i-1);
    if (rank % dis == 0){

        if(rank % (2*dis) != 0){

            int recvpid = rank - dis;
            if (recvpid >= 0 && recvpid <= (size - 1)){

                MPI_Send(&parallelSum, 1, MPI_INT, recvpid, i, MPI_COMM_WORLD);
                printf("Rank %d sends %d to rank %d\n", rank, parallelSum, recvpid);

            }
        }
```

MPI Gather

Gather: the opposite, receives data stored in small arrays from all the processes (including the source or root) and concatenates it in the receive array in rank order.




x_0
x_{n-1}
P_0
P_1
P_2
P_3
P_4
P_5
P_6
P_7
P_0
P_2
P_4
P6
P_0
P_4
P_0
Final sum

Code snippet to gather

```
        if (rank % (2*dis) == 0){

            int sendpid = rank + dis;
            if (sendpid >= 0 && sendpid <= (size-1 )){

                MPI_Recv(&total, 1, MPI_INT, sendpid, i, MPI_COMM_WORLD, &status);
                printf("Rank %d recieves %d from Rank %d\n", rank, total, sendpid);
                parallelSum = parallelSum + total;
                printf("Rank sum at process %d is: %d\n", rank, parallelSum);

            }
        }
    }}

    if (rank == 0){
        printf("Total prefix sum is %d\n", parallelSum);
```

Sample Output

slurm-154591.out

```
Partial sum: 23500 received from 1
Partial sum: 39125 received from 2
Partial sum: 54750 received from 3
Rank 1 sends 23500 to rank 0
Rank 2 recieves 54750 from Rank 3
Rank sum at process 2 is: 93875
Rank 3 sends 54750 to rank 2
Rank 0 recieves 23500 from Rank 1
Rank sum at process 0 is: 31375
Rank 2 sends 93875 to rank 0
Rank 0 recieves 93875 from Rank 2
Rank sum at process 0 is: 125250
Partial sum: 70375 received from 4
Rank 4 recieves 86000 from Rank 5
Rank sum at process 4 is: 156375
Partial sum: 86000 received from 5
Rank 5 sends 86000 to rank 4
Partial sum: 101625 received from 6
Rank 6 recieves 117250 from Rank 7
Rank sum at process 6 is: 218875
Rank 6 sends 218875 to rank 4
Partial sum: 117250 received from 7
Rank 7 sends 117250 to rank 6
Rank 4 recieves 218875 from Rank 6
Rank sum at process 4 is: 375250
Rank 0 recieves 375250 from Rank 4
Rank sum at process 0 is: 500500
Total prefix sum is 500500
Rank 4 sends 375250 to rank 0
```

Perfomance considerations

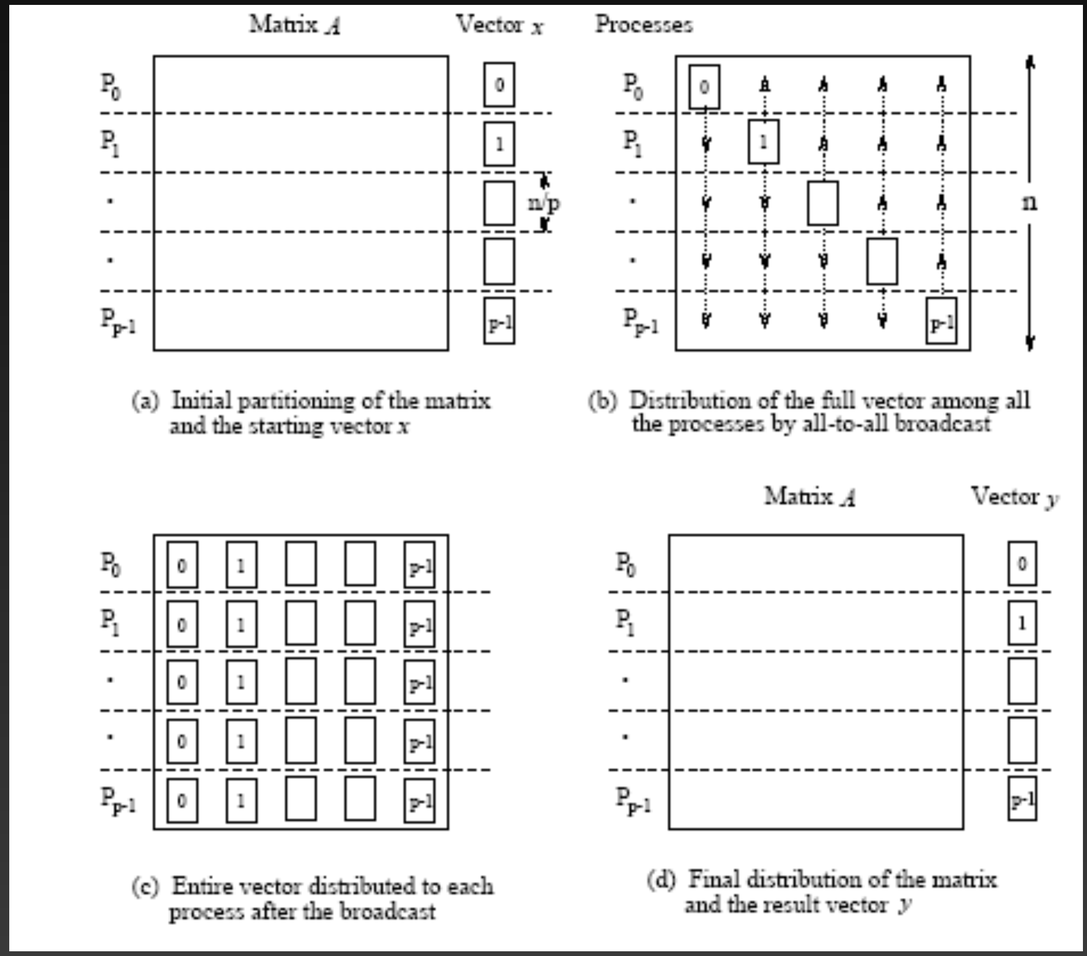
How an MPI_Scatter/Gather will work varies between implementations. Some MPI implementations may choose to use a series of MPI_Send as an underlying mechanism.

The parameters that may affect how MPI_Scatter works are:

1. Number of processes
2. Size of data
3. Interconnect For example, an implementation may avoid using a broadcast for very small number of ranks sending/receiving very large data.

1D Row-wise matrix multiplication

- 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.



1D Row-wise matrix multiplication

- 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 .
- The all-to-all broadcast and the computation of $y[i]$ both take time $\Theta(n)$. Therefore, the parallel time is $\Theta(n)$.

1D Row-wise matrix multiplication

- 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.

Code snippet to scatter

```
//MPI_Bcast(data,row*column,MPI_INT,0,MPI_COMM_WORLD);
MPI_Scatterv(data,sendcount,displace,MPI_FLOAT,rec_data,sendcount[rank],MPI_FLOAT,0,MPI_COMM_WORLD);
count=sendcount[rank]/column;
sum=(float*)calloc(sizeof(float),count);
multi(count,sum,vec,rec_data,column);
float *result=(float *)calloc(sizeof(float),row);
int disp[size];
disp[0]=0;
reccount[0]=sendcount[0]/column;
for(i=1;i<size;i++)
{
  disp[i] = disp[i-1] + sendcount[i-1]/column;
  reccount[i]=sendcount[i]/column;
}
```

Code snippet to gather

```
MPI_Gatherv(sum,count,MPI_FLOAT,result,reccount,disp,MPI_FLOAT,0,MPI_COMM_WORLD);
if(rank==0)
{
```

2D /checkerboard matrix multiplication

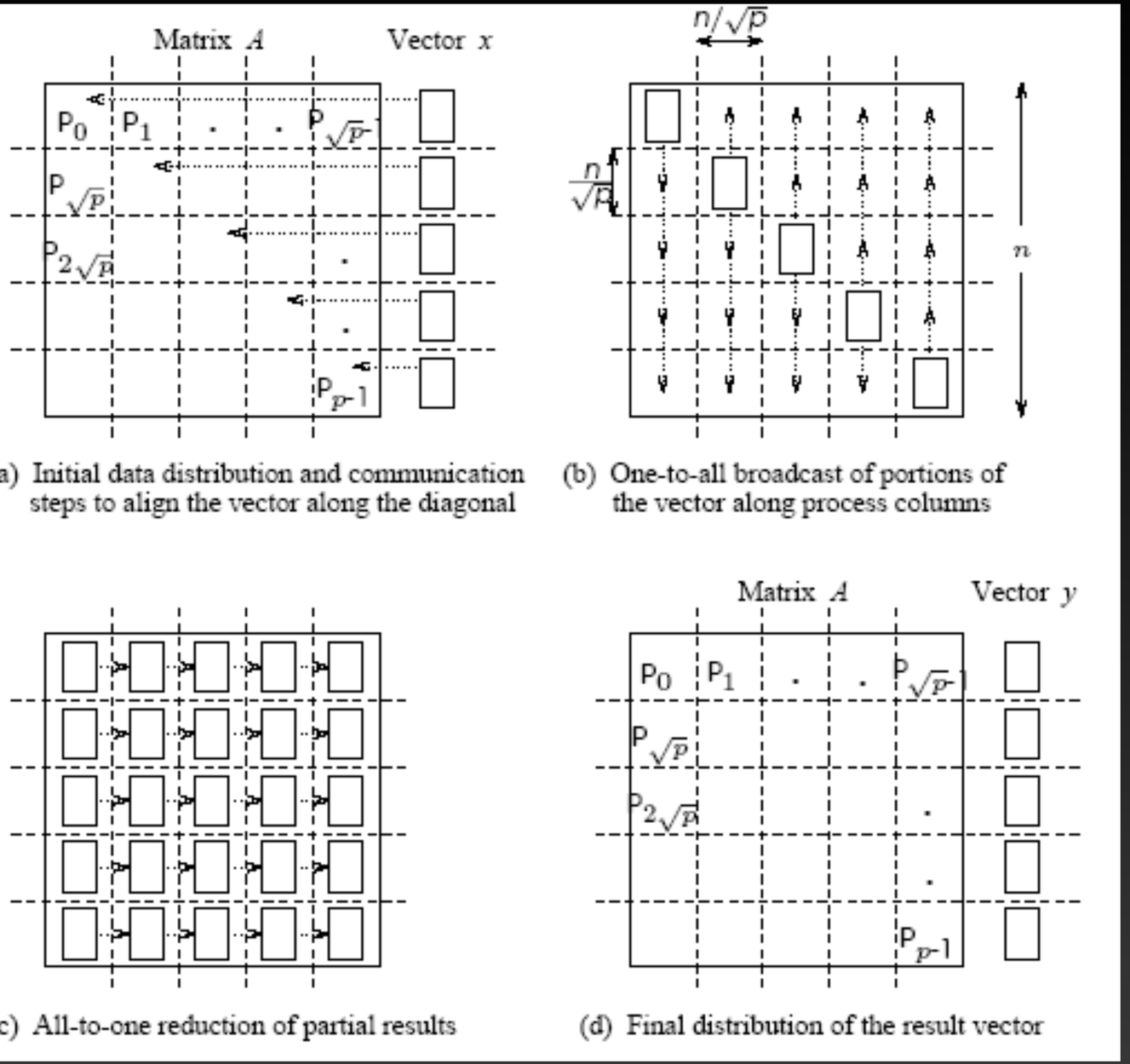
- The n x n matrix is partitioned among n^2 processors such that each processor owns a single element.
- The n x 1 vector x is distributed only in the last column of n processors.

.

Checkerboard Partitioning

Checkerboard partitioning: a matrix is divided into small square or rectangular blocks and each of them is mapped to a processor. Maximum n^2 processors

More concurrency may be exploited



(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

Operations

- One to one: the processors holding the vector row value [i] send it to the [i][i] element in the matrix
- One to all: the diagonal matrix formed from the step above broadcasts all the values in the matrix[I][I] to all the other processors to allow multiplication
- The processors all reduce the values back to the master processor to hold the final multiplication vector

Code snippet of one to one and all to all

```
}
MPI_Bcast (Vector, SIZE, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatter (MatrixA, 1, MPI_INT, MatrixA[start], 1, MPI_INT, 0, MPI_COMM_WORLD);

outPut[start] = 0;
outPut[start] += MatrixA[start][0]* Vector[(int)pid%SIZE];
```

Code snippet to reduce

```
MPI_Reduce(&outPut,&final,SIZE,MPI_INT, MPI_SUM,0,MPI_COMM_WORLD);
if (pid == 0) {
    printf("\nInput Matrix ");
for (int i = 0; i < SIZE; i++) {
    printf("\n\t|");
    for (int j = 0; j < SIZE; j++)
        printf("%2d ", MatrixA[i][j]);
    printf("|");
}
printf("\n\nVector\n");
for (int i = 0; i < SIZE; i++) {
    printf("\n\t|%2d ", Vector[i]);
    printf("|");
}
printf("\n\nMultiplication\n");
for (int i = 0; i < SIZE; i++) {
    printf("\n\t|%2d ", final[i]);
    printf("|");
}
MPI_Finalize();
return 0;
```

Sample results

slurm-154596.out

```
Java module loaded - the system's JAVA is replaced by JDK 1.8.0_211

Input Matrix
        | 1  2  3  4  5  6  7  8 |
        | 9 10 11 12 13 14 15 16 |
        |17 18 19 20 21 22 23 24 |
        |25 26 27 28 29 30 31 32 |
        |33 34 35 36 37 38 39 40 |
        |41 42 43 44 45 46 47 48 |
        |49 50 51 52 53 54 55 56 |
        |57 58 59 60 61 62 63 64 |

Vector

        | 2 |
        | 3 |
        | 4 |
        | 5 |
        | 6 |
        | 7 |
        | 8 |
        | 9 |

Multiplication

        | 2 |
        | 6 |
        |12 |
        |20 |
        |30 |
        |42 |
        |56 |
        |72 |
```