

Parallel Programming

COMP5112

Distributed Memory Programming

With MPI (2)

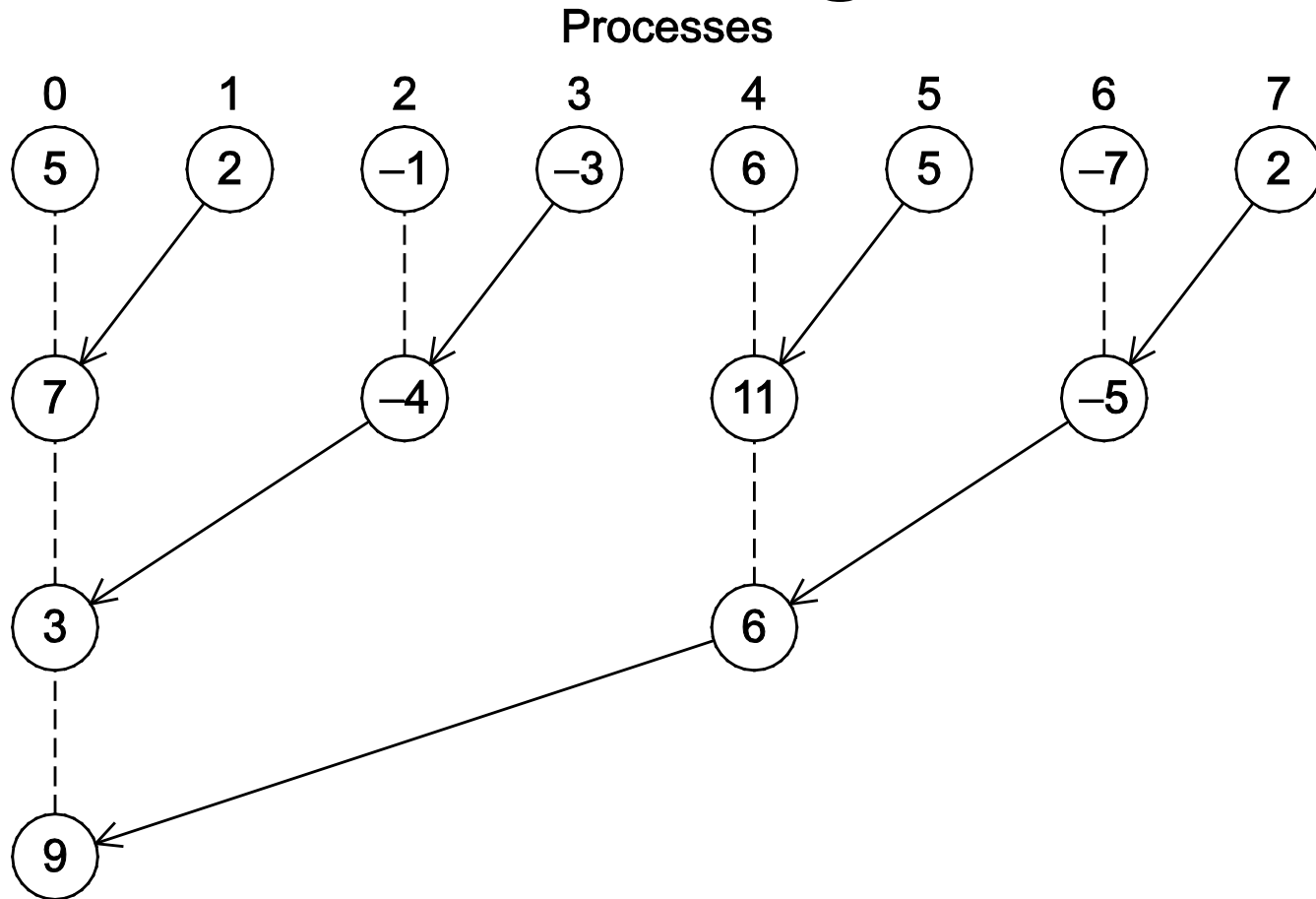
Slides adapted from the lecture notes by Peter Pacheco

Roadmap

- Writing your first MPI program.
- Using the common MPI functions.
- The Trapezoidal Rule in MPI.
- **Collective communication.**
- MPI derived datatypes.
- Performance evaluation of MPI programs.
- Parallel sorting.
- Safety in MPI programs.

COLLECTIVE COMMUNICATION

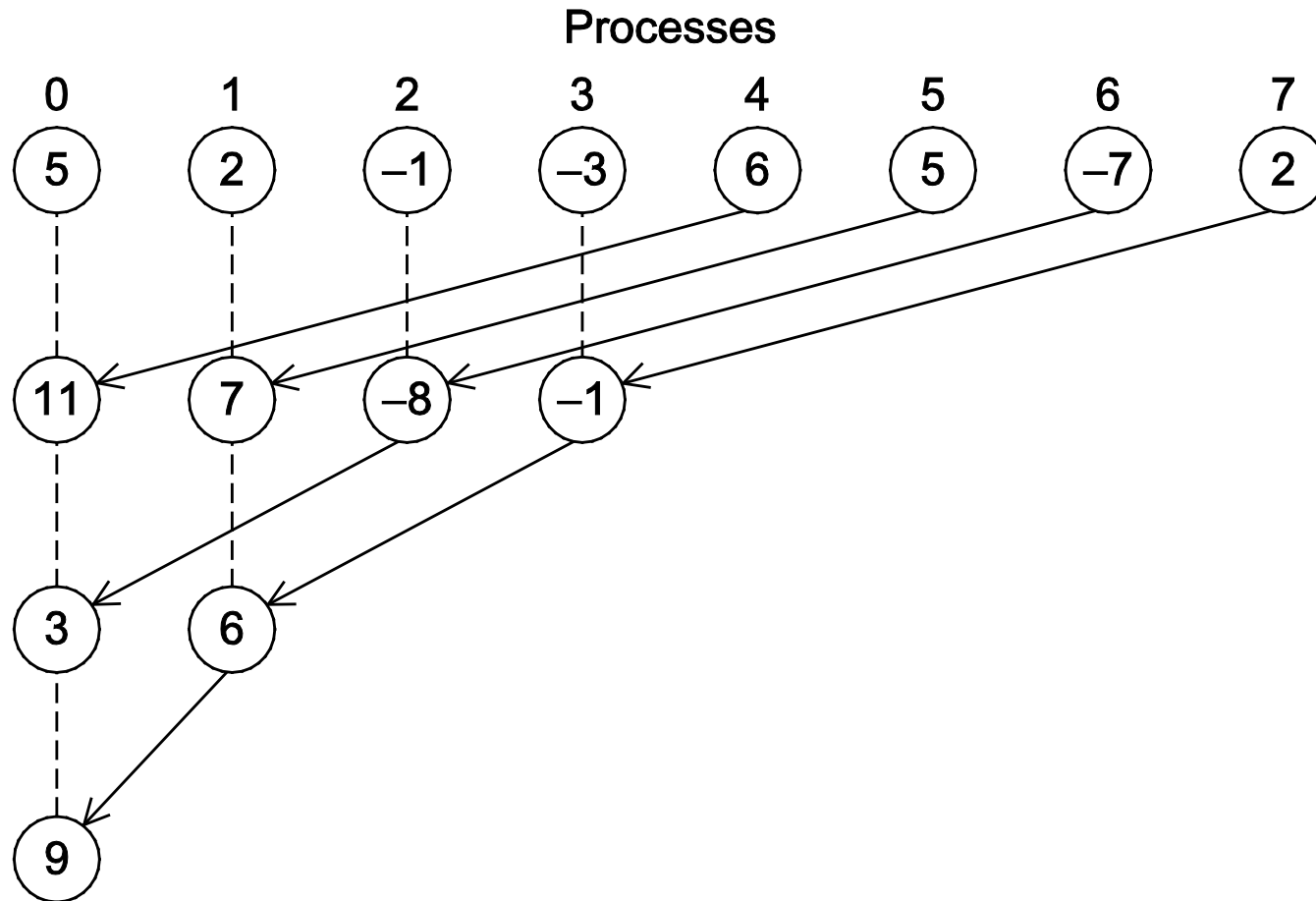
A tree-structured global sum



Tree-structured communication

1. In the first phase:
 - (a) Process 1 sends to 0, 3 sends to 2, 5 sends to 4, and 7 sends to 6.
 - (b) Processes 0, 2, 4, and 6 add in the received values.
 - (c) Processes 2 and 6 send their new values to processes 0 and 4, respectively.
 - (d) Processes 0 and 4 add the received values into their new values.
2.
 - (a) Process 4 sends its newest value to process 0.
 - (b) Process 0 adds the received value to its newest value.

An alternative tree-structured global sum



MPI_Reduce

```
int MPI_Reduce(  
    void*          input_data_p    /* in */,  
    void*          output_data_p   /* out */,  
    int            count           /* in */,  
    MPI_Datatype    datatype       /* in */,  
    MPI_Op          operator       /* in */,  
    int            dest_process    /* in */,  
    MPI_Comm        comm          /* in */);
```

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,  
          MPI_COMM_WORLD);
```

```
double local_x[N], sum[N];  
.  
.  
.  
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,  
          MPI_COMM_WORLD);
```

Predefined reduction operators in MPI

Operation Value	Meaning
<code>MPI_MAX</code>	Maximum
<code>MPI_MIN</code>	Minimum
<code>MPI_SUM</code>	Sum
<code>MPI_PROD</code>	Product
<code>MPI_LAND</code>	Logical and
<code>MPI_BAND</code>	Bitwise and
<code>MPI_LOR</code>	Logical or
<code>MPI_BOR</code>	Bitwise or
<code>MPI_LXOR</code>	Logical exclusive or
<code>MPI_BXOR</code>	Bitwise exclusive or
<code>MPI_MAXLOC</code>	Maximum and location of maximum
<code>MPI_MINLOC</code>	Minimum and location of minimum

Collective vs. Point-to-Point Communications

- All the processes in the communicator must call the same collective function.
- For example, a program that attempts to match a call to `MPI_Reduce` on one process with a call to `MPI_Recv` on another process is erroneous, and, in all likelihood, the program will hang or crash.

Collective vs. Point-to-Point Communications

- The arguments passed by each process to an MPI collective communication must be “compatible.”
- For example, if one process passes in 0 as the `dest_process` and another passes in 1, then the outcome of a call to `MPI_Reduce` is erroneous, and, once again, the program is likely to hang or crash.

Collective vs. Point-to-Point Communications

- The `output_data_p` argument is only used on `dest_process`.
- However, all of the processes still need to pass in an actual argument corresponding to `output_data_p`, even if it's just `NULL`.

Collective vs. Point-to-Point Communications

- Point-to-point communications are matched on the basis of tags and communicators.
- Collective communications don't use tags.
- They're matched solely on the basis of the communicator and the order in which they're called.

Example (1)

Time	Process 0	Process 1	Process 2
0	a = 1; c = 2	a = 1; c = 2	a = 1; c = 2
1	MPI_Reduce (&a, &b, ...)	MPI_Reduce (&c, &d, ...)	MPI_Reduce (&a, &b, ...)
2	MPI_Reduce (&c, &d, ...)	MPI_Reduce (&a, &b, ...)	MPI_Reduce (&c, &d, ...)

Multiple calls to MPI_Reduce

Example (2)

- Suppose that each process calls `MPI_Reduce` with operator `MPI_SUM`, and destination process 0.
- At first glance, it might seem that after the two calls to `MPI_Reduce`, the value of `b` will be 3, and the value of `d` will be 6.

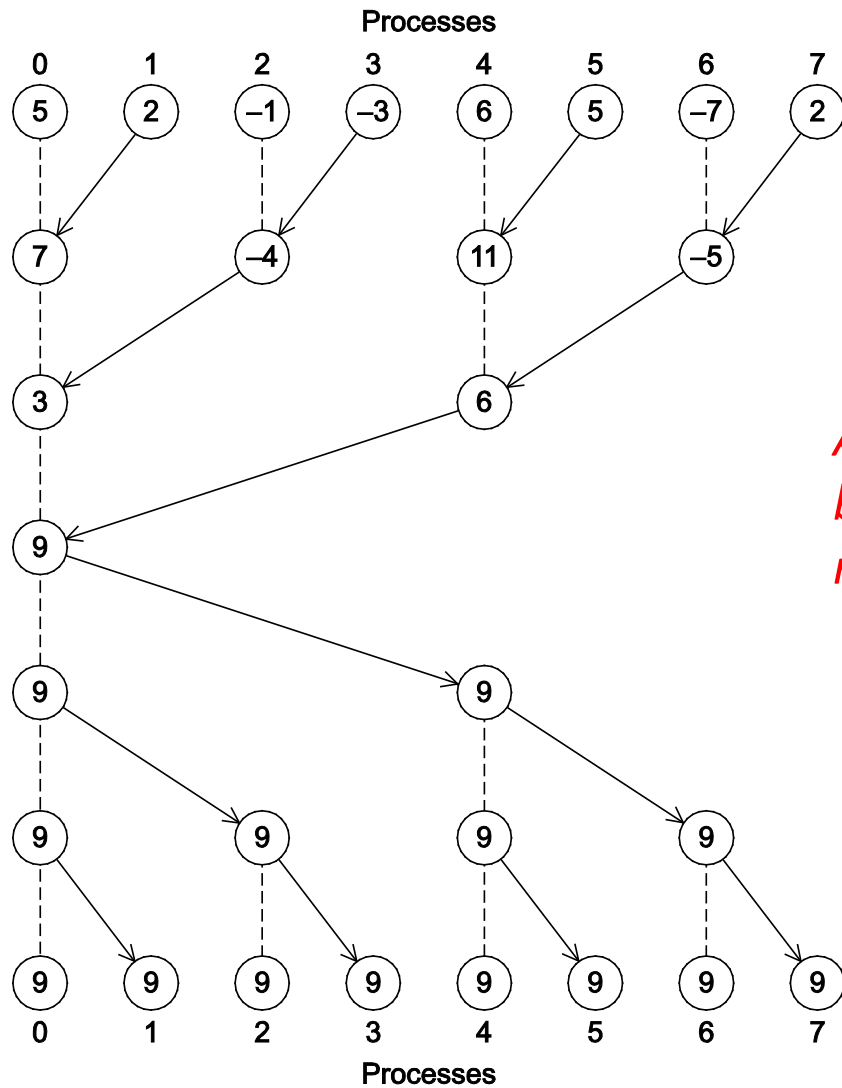
Example (3)

- However, the names of the memory locations are irrelevant to the matching of the calls to `MPI_Reduce`.
- The order of the calls will determine the matching so the value stored in b will be $1+2+1 = 4$, and the value stored in d will be $2+1+2 = 5$.

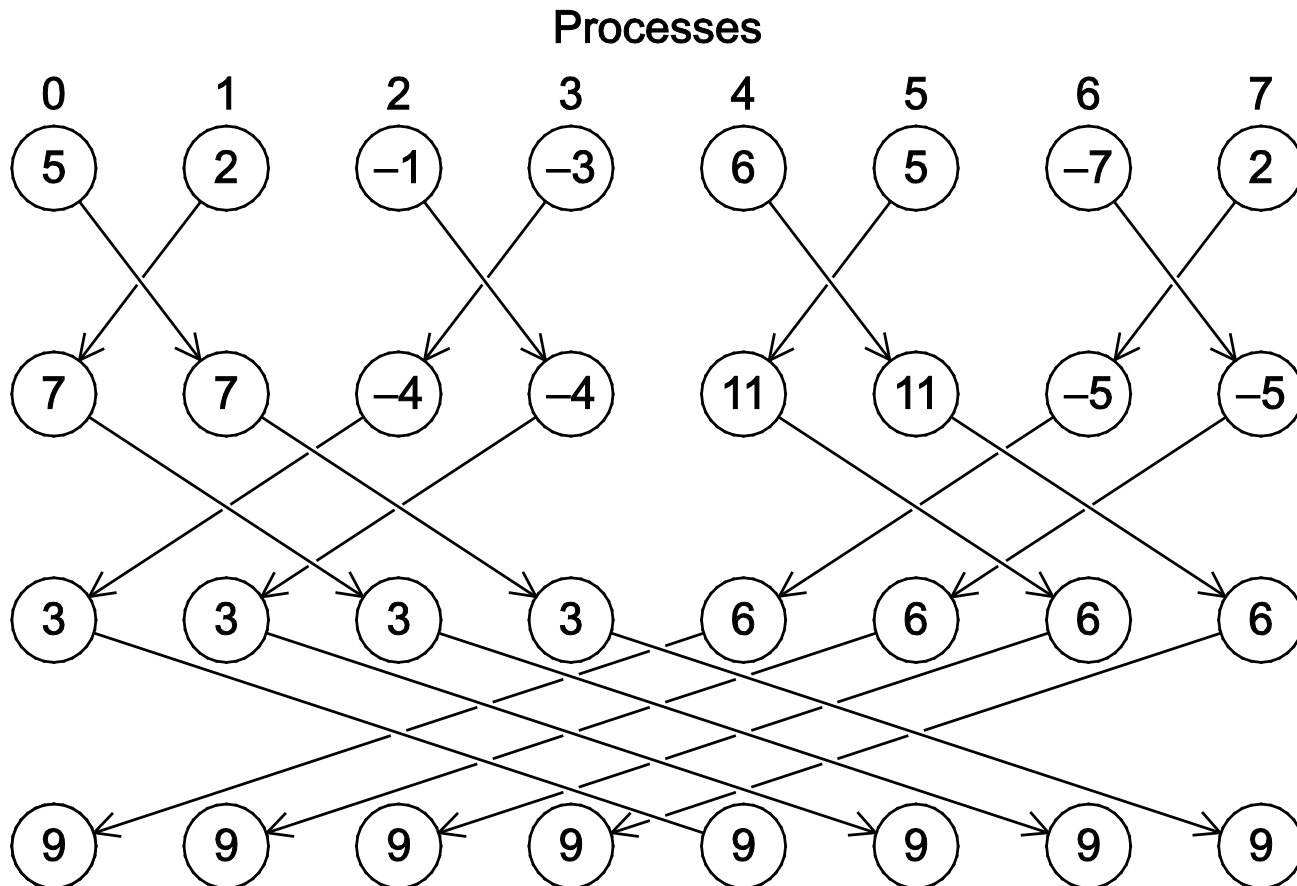
MPI_Allreduce

- Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

```
int MPI_Allreduce(  
    void*          input_data_p    /* in */ ,  
    void*          output_data_p  /* out */ ,  
    int            count           /* in */ ,  
    MPI_Datatype    datatype       /* in */ ,  
    MPI_Op          operator       /* in */ ,  
    MPI_Comm        comm           /* in */ );
```

*A global sum followed
by distribution of the
result.*

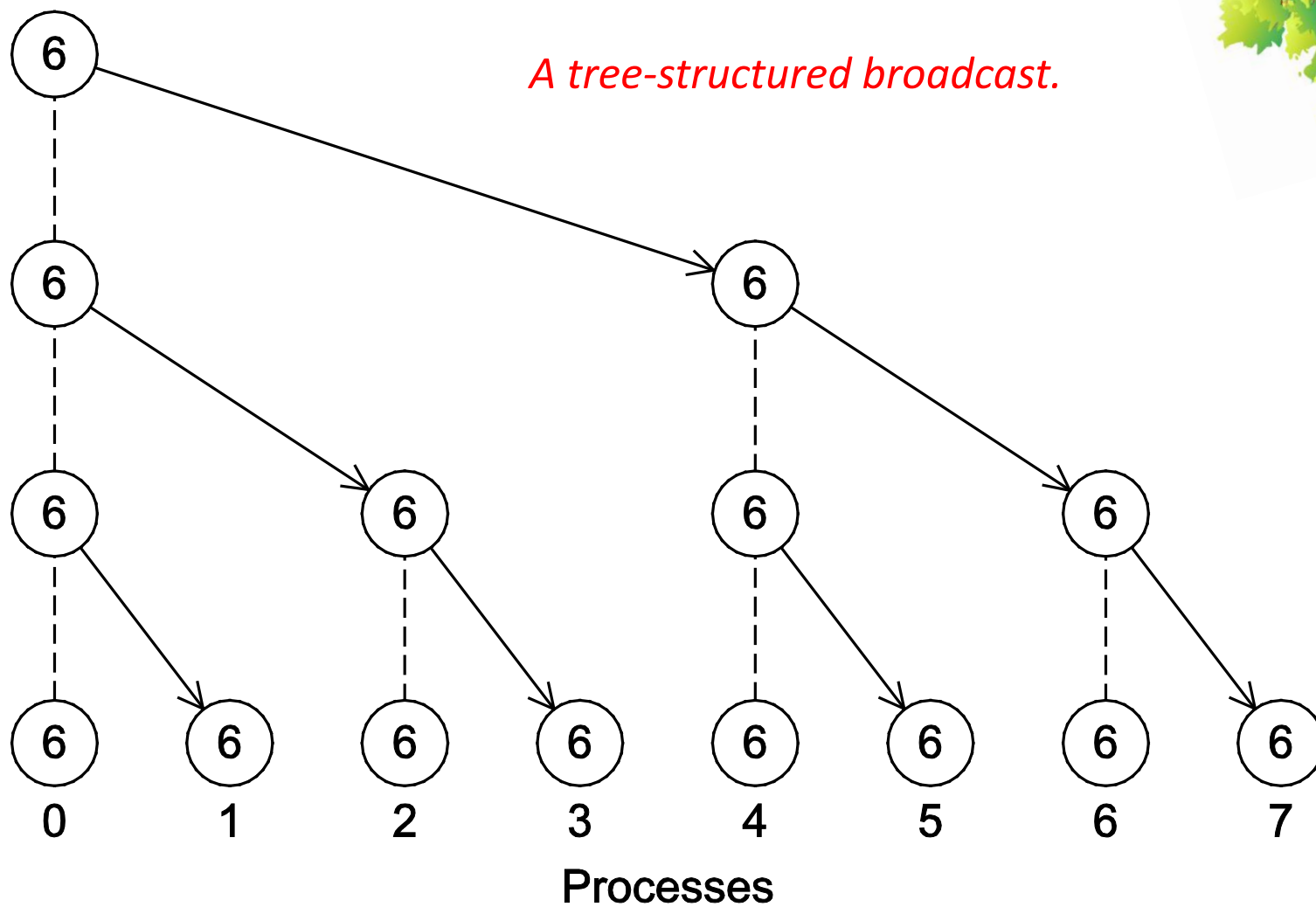


A butterfly-structured global sum.

Broadcast

- Data belonging to a single process is sent to all of the processes in the communicator.

```
int MPI_Bcast(  
    void*      data_p      /* in/out */,  
    int        count       /* in      */,  
    MPI_Datatype datatype   /* in      */,  
    int        source_proc  /* in      */,  
    MPI_Comm    comm       /* in      */);
```



A version of Get_input that uses MPI_Bcast

```
void Get_input(  
    int      my_rank    /* in */,  
    int      comm_sz    /* in */,  
    double*  a_p        /* out */,  
    double*  b_p        /* out */,  
    int*     n_p        /* out */) {  
  
    if (my_rank == 0) {  
        printf("Enter a, b, and n\n");  
        scanf("%lf %lf %d", a_p, b_p, n_p);  
    }  
    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);  
} /* Get_input */
```

Data distributions

$$\begin{aligned}\mathbf{x} + \mathbf{y} &= (x_0, x_1, \dots, x_{n-1}) + (y_0, y_1, \dots, y_{n-1}) \\ &= (x_0 + y_0, x_1 + y_1, \dots, x_{n-1} + y_{n-1}) \\ &= (z_0, z_1, \dots, z_{n-1}) \\ &= \mathbf{z}\end{aligned}$$

Compute a vector sum.

Serial implementation of vector addition

```
void Vector_sum(double x[], double y[], double z[], int n) {  
    int i;  
  
    for (i = 0; i < n; i++)  
        z[i] = x[i] + y[i];  
} /* Vector_sum */
```

Different partitions of a 12-component vector among 3 processes

Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Partitioning options

- Block partitioning
 - Assign blocks of consecutive components to each process.
- Cyclic partitioning
 - Assign components in a round robin fashion.
- Block-cyclic partitioning
 - Use a cyclic distribution of blocks of components.

Parallel implementation of vector addition

```
void Parallel_vector_sum(  
    double  local_x[]  /* in  */,  
    double  local_y[]  /* in  */,  
    double  local_z[]  /* out */,  
    int     local_n    /* in  */) {  
    int local_i;  
  
    for (local_i = 0; local_i < local_n; local_i++)  
        local_z[local_i] = local_x[local_i] + local_y[local_i];  
} /* Parallel_vector_sum */
```

Scatter

- MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

```
int MPI_Scatter(  
    void*      send_buf_p  /* in */,  
    int        send_count  /* in */,  
    MPI_Datatype send_type  /* in */,  
    void*      recv_buf_p  /* out */,  
    int        recv_count  /* in */,  
    MPI_Datatype recv_type  /* in */,  
    int        src_proc     /* in */,  
    MPI_Comm    comm        /* in */);
```

Reading and distributing a vector

```
void Read_vector(  
    double    local_a[]    /* out */,  
    int       local_n      /* in  */,  
    int       n            /* in  */,  
    char      vec_name[]   /* in  */,  
    int       my_rank      /* in  */,  
    MPI_Comm  comm        /* in  */) {  
  
    double* a = NULL;  
    int i;  
  
    if (my_rank == 0) {  
        a = malloc(n*sizeof(double));  
        printf("Enter the vector %s\n", vec_name);  
        for (i = 0; i < n; i++)  
            scanf("%lf", &a[i]);  
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,  
                    0, comm);  
        free(a);  
    } else {  
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,  
                    0, comm);  
    }  
} /* Read_vector */
```

Gather

- Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.

```
int MPI_Gather(  
    void*          send_buf_p    /* in    */,  
    int           send_count    /* in    */,  
    MPI_Datatype   send_type     /* in    */,  
    void*          recv_buf_p    /* out   */,  
    int           recv_count    /* in    */,  
    MPI_Datatype   recv_type     /* in    */,  
    int           dest_proc      /* in    */,  
    MPI_Comm       comm         /* in    */);
```

Print a distributed vector (1)

```
void Print_vector(  
    double    local_b[]    /* in */,  
    int       local_n      /* in */,  
    int       n            /* in */,  
    char      title[]      /* in */,  
    int       my_rank      /* in */,  
    MPI_Comm  comm         /* in */) {  
  
    double* b = NULL;  
    int i;
```

Print a distributed vector (2)

```
if (my_rank == 0) {
    b = malloc(n*sizeof(double));
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
               0, comm);
    printf("%s\n", title);
    for (i = 0; i < n; i++)
        printf("%f ", b[i]);
    printf("\n");
    free(b);
} else {
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
               0, comm);
}
} /* Print_vector */
```

Allgather

- Concatenates the contents of each process' `send_buf_p` and stores this in each process' `recv_buf_p`.
- As usual, `recv_count` is the amount of data being received from each process.

```
int MPI_Allgather(  
    void*      send_buf_p    /* in */,  
    int        send_count    /* in */,  
    MPI_Datatype send_type    /* in */,  
    void*      recv_buf_p    /* out */,  
    int        recv_count    /* in */,  
    MPI_Datatype recv_type    /* in */,  
    MPI_Comm   comm          /* in */);
```


Matrix-vector multiplication

$A = (a_{ij})$ is an $m \times n$ matrix

\mathbf{x} is a vector with n components

$\mathbf{y} = A\mathbf{x}$ is a vector with m components

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}$$

i -th component of \mathbf{y}

*Dot product of the i th
row of A with \mathbf{x} .*

Matrix-vector multiplication

a_{00}	a_{01}	\cdots	$a_{0,n-1}$
a_{10}	a_{11}	\cdots	$a_{1,n-1}$
\vdots	\vdots		\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$
\vdots	\vdots		\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$

x_0
x_1
\vdots
x_{n-1}

=

y_0
y_1
\vdots
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
\vdots
y_{m-1}

Multiply a matrix by a vector

```
/* For each row of A */  
for (i = 0; i < m; i++) {  
    /* Form dot product of ith row with x */  
    y[i] = 0.0;  
    for (j = 0; j < n; j++)  
        y[i] += A[i][j]*x[j];  
}
```

Serial pseudo-code

C style arrays

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

stored as

0 1 2 3 4 5 6 7 8 9 10 11

Serial matrix-vector multiplication

```
void Mat_vect_mult(  
    double  A[] /* in */,  
    double  x[] /* in */,  
    double  y[] /* out */,  
    int      m /* in */,  
    int      n /* in */) {  
    int i, j;  
  
    for (i = 0; i < m; i++) {  
        y[i] = 0.0;  
        for (j = 0; j < n; j++)  
            y[i] += A[i*n+j]*x[j];  
    }  
} /* Mat_vect_mult */
```

An MPI matrix-vector multiplication function (1)

```
void Mat_vect_mult(  
    double    local_A[]    /* in    */,  
    double    local_x[]    /* in    */,  
    double    local_y[]    /* out   */,  
    int        local_m      /* in    */,  
    int        n            /* in    */,  
    int        local_n      /* in    */,  
    MPI_Comm   comm        /* in    */) {  
    double* x;  
    int local_i, j;  
    int local_ok = 1;
```

An MPI matrix-vector multiplication function (2)

```
x = malloc(n*sizeof(double));
MPI_Allgather(local_x, local_n, MPI_DOUBLE,
              x, local_n, MPI_DOUBLE, comm);

for (local_i = 0; local_i < local_m; local_i++) {
    local_y[local_i] = 0.0;
    for (j = 0; j < n; j++)
        local_y[local_i] += local_A[local_i*n+j]*x[j];
}
free(x);
} /* Mat_vect_mult */
```