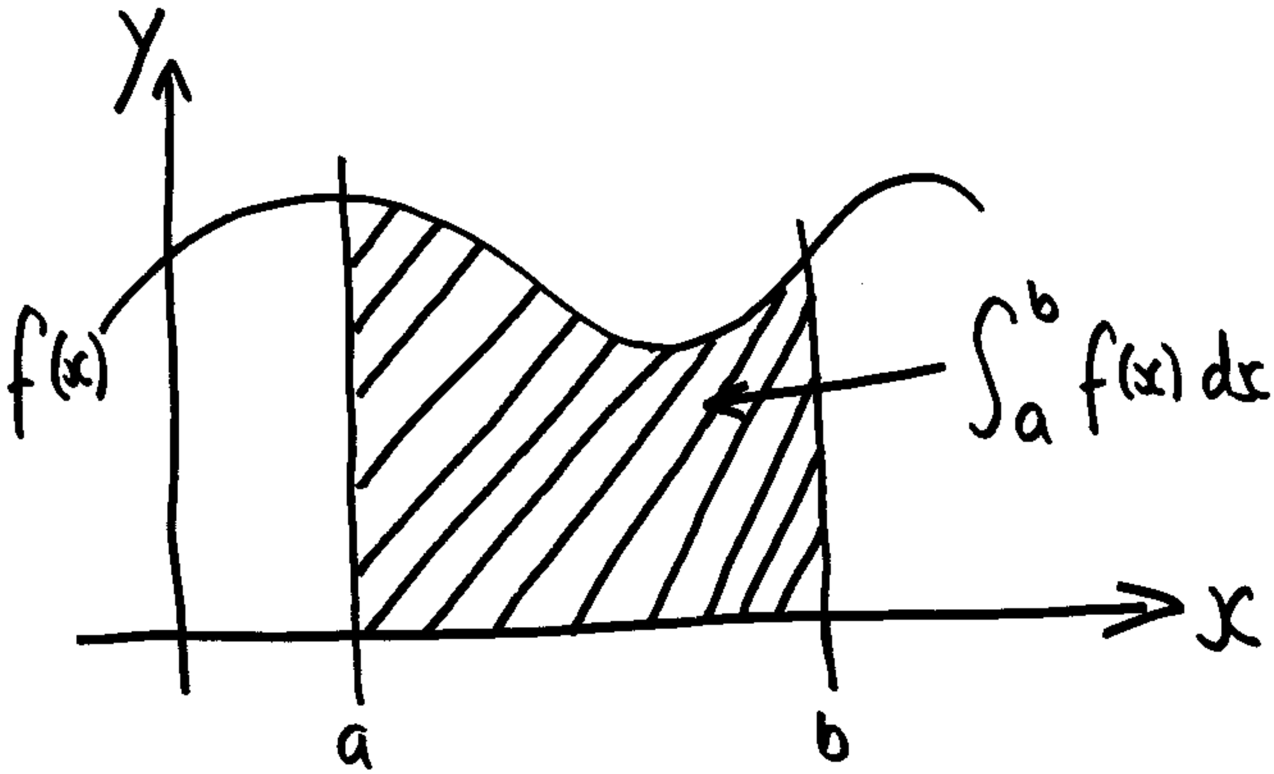
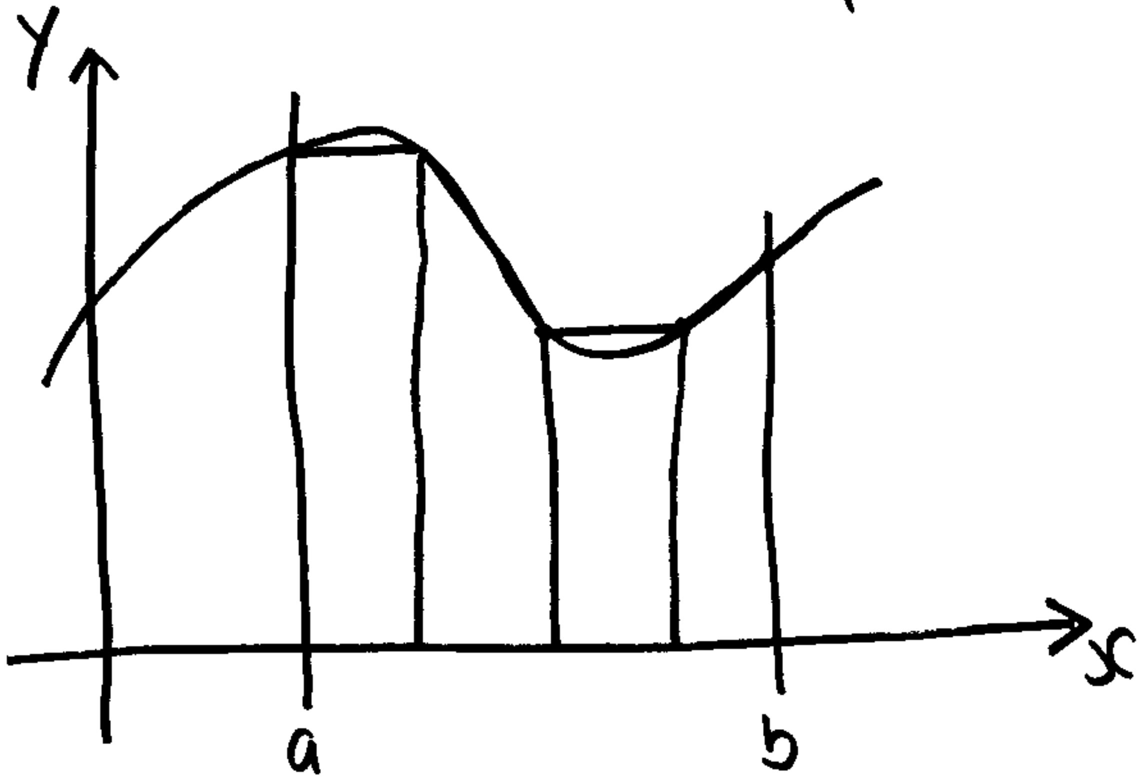


Definite integral of a nonnegative function

The Trapezoidal Rule



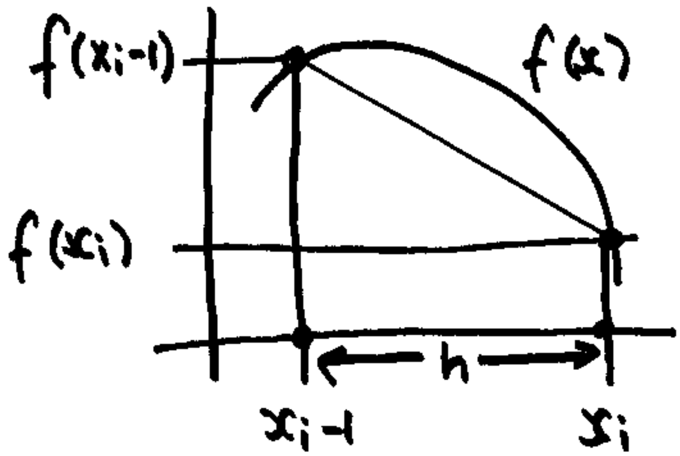
n trapezoids

Will assume all bases have the same length

the length of the base of each trapezoid

$$h = (b-a)/n$$

base of i^{th} trapezoid $[a + (i-1)h, a + ih]$, $\forall i \in 1, \ldots, n$



Let x_i denote $a+ih$, $i=0,\dots,n$

Left side of i^{th} trapezoid : $f(x_{i-1})$

Right side " : $f(x_i)$

Area of " " : $\frac{1}{2}h(f(x_{i-1})+f(x_i))$

Total area

$$\tfrac{1}{2}h[f(x_0)+f(x_1)] + \dots + \tfrac{1}{2}h[f(x_{n-1})+f(x_n)]$$

$$= \tfrac{h}{2}[f(x_0)+2f(x_1)+2f(x_2)+\dots+f(x_n)]$$

$$= h[f(x_0)/2 + f(x_n)/2 + f(x_1) + f(x_2) \dots f(x_{n-1})]$$

```c
/* Calculate definite integral using trapezoidal rule.
 * The function f(x) is hardwired.
 * Input: a, b, n.
 * Output: estimate of integral from a to b of f(x)
 *    using n trapezoids.
 */

#include <stdio.h>

main() {
    float  integral;   /* Store result in integral   */
    float  a, b;       /* Left and right endpoints   */
    int    n;          /* Number of trapezoids       */
    float  h;          /* Trapezoid base width       */
    float  x;
    int    i;

    float f(float x);  /* Function we're integrating */

    printf("Enter a, b, and n\n");
    scanf("%f %f %d", &a, &b, &n);

    h = (b-a)/n;
    integral = (f(a) + f(b))/2.0;
    x = a;
    for (i = 1; i <= n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %f\n",
        a, b, integral);
} /* main */


float f(float x) {
    float return_val;
    /* Calculate f(x).  Store calculation in return_val. */
          :
    return return_val;
} /* f */
```

Parallelizing the Trapezoidal Rule

Idea:
- Assign each process a subinterval of [a..b]
- each process estimates the integral of f over its subinterval
- global result = sum of local results.

Assume n evenly divisable by P

Process	Interval
0	$[a, a + \frac{n}{p}h]$
1	$[a + \frac{n}{p}h, a + 2\frac{n}{p}h]$
⋮	
i	$[a + i\frac{n}{p}h, a + (i+1)\frac{n}{p}h]$
⋮	
p-1	$[a + (p-1)\frac{n}{p}h, b]$

```c
/* Parallel Trapezoidal Rule
 *
 * Input: None.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Algorithm:
 *    1.  Each process calculates "its" interval of
 *        integration.
 *    2.  Each process estimates the integral of f(x)
 *        over its interval using the trapezoidal rule.
 *    3a. Each process != 0 sends its integral to 0.
 *    3b. Process 0 sums the calculations received from
 *        the individual processes and prints the result.
 *
 * Note:  f(x), a, b, and n are all hardwired.
 */
#include <stdio.h>

/* We'll be using MPI routines, definitions, etc. */
#include "mpi.h"


main(int argc, char** argv) {
    int         my_rank;   /* My process rank           */
    int         p;         /* The number of processes   */
    float       a = 0.0;   /* Left endpoint             */
    float       b = 1.0;   /* Right endpoint            */
    int         n = 1024;  /* Number of trapezoids      */
    float       h;         /* Trapezoid base length     */
    float       local_a;   /* Left endpoint my process  */


    float       local_b;   /* Right endpoint my process */
    int         local_n;   /* Number of trapezoids for  */
                           /* my calculation            */
    float       integral;  /* Integral over my interval */
    float       total;     /* Total integral            */
    int         source;    /* Process sending integral  */
    int         dest = 0;  /* All messages go to 0      */
    int         tag = 0;
    MPI_Status  status;

    float Trap(float local_a, float local_b, int local_n,
              float h);    /* Calculate local integral  */
```

```
    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process's interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    if (my_rank == 0) {
        total = integral;
        for (source = 1; source < p; source++) {
            MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                MPI_COMM_WORLD, &status);
            total = total + integral;
        }
    } else {
        MPI_Send(&integral, 1, MPI_FLOAT, dest,
            tag, MPI_COMM_WORLD);
    }

    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",
            n);


        printf("of the integral from %f to %f = %f\n",
            a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /*  main  */
```

```
float Trap(
          float  local_a   /* in */,
          float  local_b   /* in */,
          int    local_n   /* in */,
          float  h         /* in */) {

    float integral;   /* Store result in integral  */
    float x;
    int i;

    float f(float x); /* function we're integrating */

    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
} /*  Trap  */


float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
           :
    return return_val;
} /* f */
```

Improvements?

Don't hardcode f(x), a, b, and n
(a, b, and n: will consider)

Idea 1:

Modify trapezoidal program so that each process attempts

```
scanf("%f %f %d", &a, &b, &n);
```

What will happen?

If user enters 0 1 1024

- All processes get 0 1 1024
- First proc. gets 0, second 1, third 1024
- ?

Will assume only process 0 can do terminal I/O

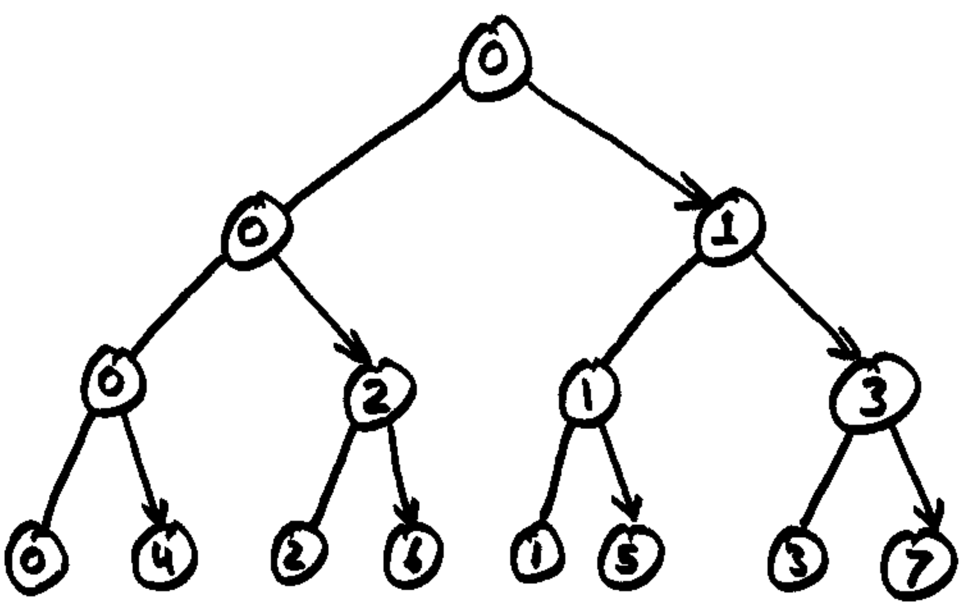
```
/* Function Get_data
 * Reads in the user input a, b, and n.
 * Input parameters:
 *     1.  int my_rank:  rank of current process.
 *     2.  int p:  number of processes.
 * Output parameters:
 *     1.  float* a_ptr:  pointer to left endpoint a.
 *     2.  float* b_ptr:  pointer to right endpoint b.
 *     3.  int* n_ptr:  pointer to number of trapezoids.
 * Algorithm:
 *     1.  Process 0 prompts user for input and
 *         reads in the values.
 *     2.  Process 0 sends input values to other
 *         processes.
 */
```

```
void Get_data(
         float*  a_ptr    /* out */,
         float*  b_ptr    /* out */,

         int*    n_ptr    /* out */,
         int     my_rank  /* in  */,
         int     p        /* in  */) {

    int source = 0;    /* All local variables used by */
    int dest;          /* MPI_Send and MPI_Recv       */
    int tag;
    MPI_Status status;

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
        for (dest = 1; dest < p; dest++){
            tag = 0;
            MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag,
                MPI_COMM_WORLD);
            tag = 1;
            MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag,
                MPI_COMM_WORLD);
            tag = 2;
            MPI_Send(n_ptr, 1, MPI_INT, dest, tag,
                MPI_COMM_WORLD);
        }
    } else {
        tag = 0;
        MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag,
            MPI_COMM_WORLD, &status);
        tag = 1;
        MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag,
            MPI_COMM_WORLD, &status);
        tag = 2;
        MPI_Recv(n_ptr, 1, MPI_INT, source, tag,
                MPI_COMM_WORLD, &status);
    }
} /* Get_data */
```

Further Improvements ?

busy	
0	1) Read input
0	2) Distribute input data
0–p-1	3) Compute integral
0	4) Collect and sum results

Idea : Configure processes as a tree



With P= 1024 # of rounds = 10
(100 fold improvement !)

```
for (stage = first; stage <= last; stage++)
    if (I_receive(stage, my_rank, &source))
        Receive(data, source);
    else if (I_send(stage, my_rank, p, &dest))
        Send(data, dest);
```

To implement this code we need to compute

- whether a proc. recieves and if so the source
- whether a proc. sends and if so the dest.

we chose

1. 0 → 1
2. 0 → 2, 1 → 3
3. 0 → 4, 1 → 5,
   2 → 6, 3 → 7

could have chosen

1. 0 → 4
2. 0 → 2, 4 → 6
3. 0 → 1, 2 → 3
   4 → 5, 6 → 7

Which is better?

Can't tell, it depend on the topology

Stage 0, Stage 1, ...

If $2^{stage} \le my\_rank < 2^{stage+1}$

then I receive from $my\_rank - 2^{stage}$.

If $my\_rank < 2^{stage}$,

then I send to $my\_rank + 2^{stage}$

```
/* Ceiling of log_2(x) is just the number of
 * times x-1 can be divided by 2 until the quotient
 * is 0.  Dividing by 2 is the same as right shift.
 */
int Ceiling_log2(int  x  /* in */) {
    /* Use unsigned so that right shift will fill
     * leftmost bit with 0
     */
    unsigned temp = (unsigned) x - 1;
    int result = 0;

    while (temp != 0) {
         temp = temp >> 1;
         result = result + 1 ;
    }

    return result;
} /* Ceiling_log2 */

int I_receive(
        int    stage       /* in  */,
        int    my_rank     /* in  */,
        int*   source_ptr  /* out */) {

    int   power_2_stage;

    /* 2^stage = 1 << stage */
    power_2_stage = 1 << stage;
    if ((power_2_stage <= my_rank) &&
            (my_rank < 2*power_2_stage)){
        *source_ptr = my_rank - power_2_stage;
        return 1;
    } else return 0;
} /* I_receive */
```

```
int I_send(
        int   stage    /* in  */,
        int   my_rank  /* in  */,
        int   p        /* in  */,
        int*  dest_ptr /* out */) {
    int power_2_stage;

    /* 2^stage = 1 << stage */
    power_2_stage = 1 << stage;
    if (my_rank < power_2_stage){
        *dest_ptr = my_rank + power_2_stage;
        if (*dest_ptr >= p) return 0;
        else return 1;
    } else return 0;
} /* I_send */

void Send(
        float  a      /* in */,
        float  b      /* in */,
        int    n      /* in */,
        int    dest   /* in */) {

    MPI_Send(&a, 1, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
    MPI_Send(&b, 1, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);
    MPI_Send(&n, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
} /* Send */

void Receive(
        float*  a_ptr  /* out */,
        float*  b_ptr  /* out */,
        int*    n_ptr  /* out */,
        int     source /* in  */) {

    MPI_Status status;

    MPI_Recv(a_ptr, 1, MPI_FLOAT, source, 0,
        MPI_COMM_WORLD, &status);
    MPI_Recv(b_ptr, 1, MPI_FLOAT, source, 1,
        MPI_COMM_WORLD, &status);
    MPI_Recv(n_ptr, 1, MPI_INT, source, 2,
        MPI_COMM_WORLD, &status);
} /* Receive */
```

```
void Get_data1(
        float*  a_ptr    /* out */,
        float*  b_ptr    /* out */,
        int*    n_ptr    /* out */,
        int     my_rank  /* in  */,
        int     p        /* in  */) {

    int source;
    int dest;
    int stage;

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }
    for (stage = 0; stage < Ceiling_log2(p); stage++)
        if (I_receive(stage, my_rank, &source))
            Receive(a_ptr, b_ptr, n_ptr, source);
        else if (I_send(stage, my_rank, p, &dest))
            Send(*a_ptr, *b_ptr, *n_ptr, dest);
} /* Get_data1*/
```

The Better way: MPI Collective Comm.

```
int MPI_Bcast(
        void*           message    /* in/out */,
        int             count      /* in     */,
        MPI_Datatype    datatype   /* in     */,
        int             root       /* in     */,
        MPI_Comm        comm       /* in     */)
```

```
void Get_data2(
        float*  a_ptr    /* out */,
        float*  b_ptr    /* out */,
        int*    n_ptr    /* out */,
        int     my_rank  /* in  */) {

        if (my_rank == 0) {
            printf("Enter a, b, and n\n");
            scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
        }
        MPI_Bcast(a_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
        MPI_Bcast(b_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
        MPI_Bcast(n_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
    } /* Get_data2 */
```

Broadcast times (times are in milliseconds; version 1 uses a linear loop of sends from process 0, version 2 uses `MPI_Bcast`; all systems running `mpich`)

	nCUBE2		*Paragon*		*SP2*	
Processes	*Version 1*	*Version 2*	*Version 1*	*Version 2*	*Version 1*	*Version 2*
2	0.59	0.69	0.21	0.43	0.15	0.16
8	4.7	1.9	0.84	0.93	0.55	0.35
32	19.0	3.0	3.2	1.3	2.0	0.57

MPI Collective Communication Operations

- All processes in communicator involved
- No tags!
- No barrier at end but, ...
  1. May be point of synchronization if there is not enough buffer space
  2. Relative order of data communicated is maintained!

MPI Reduce

- "Scan", apply binary associative op

```
int MPI_Reduce(
        void*          operand    /* in  */,
        void*          result     /* out */,
        int            count      /* in  */,
        MPI_Datatype   datatype   /* in  */,
        MPI_Op         operator   /* in  */,
        int            root       /* in  */,
        MPI_Comm       comm       /* in  */)
```

- What Operations?

Predefined reduction operators in MPI

Operation Name	*Meaning*
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Using MPI_Reduce in Trapezoidal

```
        ⋮
/* Add up the integrals calculated by each process */
MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
    MPI_SUM, 0, MPI_COMM_WORLD);

/* Print the result */
        ⋮
```

Notes

- only "root" processor gets "total"
- all processors must execute call
- operand & result must <u>Not</u> be aliases

```
/* Attempt to store the result in the same
 * location as the operand.  Illegal call.
 */
MPI_Reduce(&integral, &integral, 1, MPI_FLOAT,
    MPI_SUM, 0, MPI_COMM_WORLD);
```

What if all processes need the result?

```
int MPI_Allreduce(
        void*          operand   /* in  */,
        void*          result    /* out */,
        int            count     /* in  */,
        MPI_Datatype   datatype  /* in  */,
        MPI_Op         operator  /* in  */,
        MPI_Comm       comm      /* in  */)
```

What about a partial sum?

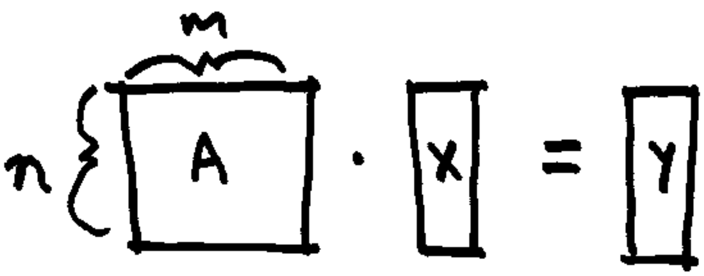
	5	8	2	1	3	6
psum	5	13	15	16	19	25

See

MPI_Scan

Gather & Scatter

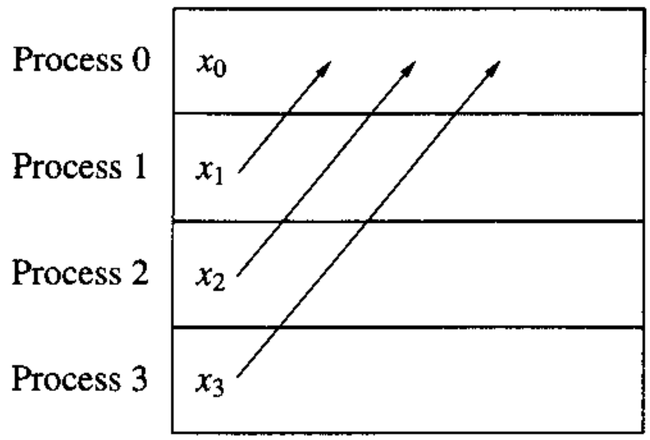
Consider Matrix Vector Mult.



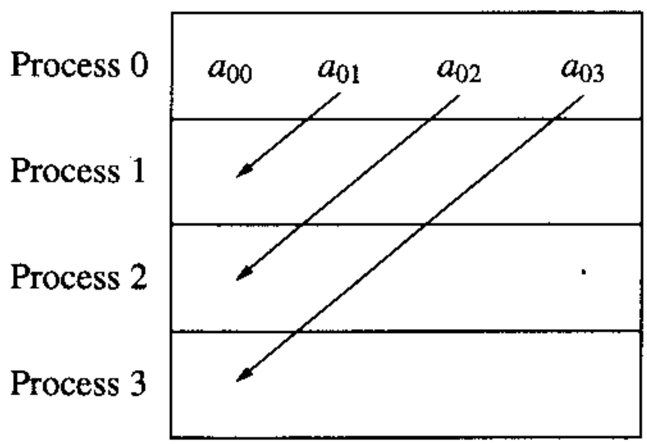
Sequential

```
/* MATRIX_T is a two-dimensional array of floats */
void Serial_matrix_vector_prod(
        MATRIX_T  A    /* in  */,
        int       m    /* in  */,
        int       n    /* in  */,
        float     x[]  /* in  */,
        float     y[]  /* out */) {

    int k, j;

    for (k = 0; k < m; k++) {
        y[k] = 0.0;
        for (j = 0; j < n; j++)
            y[k] = y[k] + A[k][j]*x[j];
    }
}  /* Serial_matrix_vector_prod */
```

Two approaches



A gather

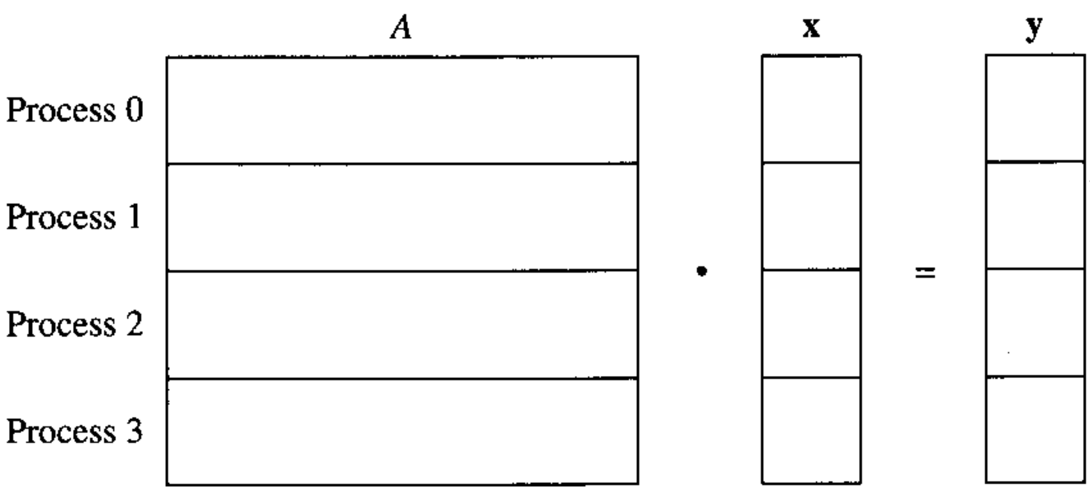


A scatter

How in Parallel?

Block-row distribution

Process	*Elements of A*			
0	a_{00}	a_{01}	a_{02}	a_{03}
	a_{10}	a_{11}	a_{12}	a_{13}
1	a_{20}	a_{21}	a_{22}	a_{23}
	a_{30}	a_{31}	a_{32}	a_{33}
2	a_{40}	a_{41}	a_{42}	a_{43}
	a_{50}	a_{51}	a_{52}	a_{53}
3	a_{60}	a_{61}	a_{62}	a_{63}
	a_{70}	a_{71}	a_{72}	a_{73}



Mappings of A, $\mathbf{x}$, and $\mathbf{y}$ for matrix-vector product

Note

- All processors need a copy of X, or

Gather

```
/* Space allocated in calling program     */
float local_x[];  /* local storage for x  */
float global_x[]; /* storage for all of x */

/* Assumes n is divisible by p */
MPI_Gather(local_x, n/p, MPI_FLOAT,
           global_x, n/p, MPI_FLOAT,
           0, MPI_COMM_WORLD);
```

The exact syntax of `MPI_Gather` is

```
int MPI_Gather(
        void*         send_data    /* in  */,
        int           send_count   /* in  */,
        MPI_Datatype  send_type    /* in  */,
        void*         recv_data    /* out */,
        int           recv_count   /* in  */,
        MPI_Datatype  recv_type    /* in  */,
        int           root         /* in  */,
        MPI_Comm      comm         /* in  */)
```

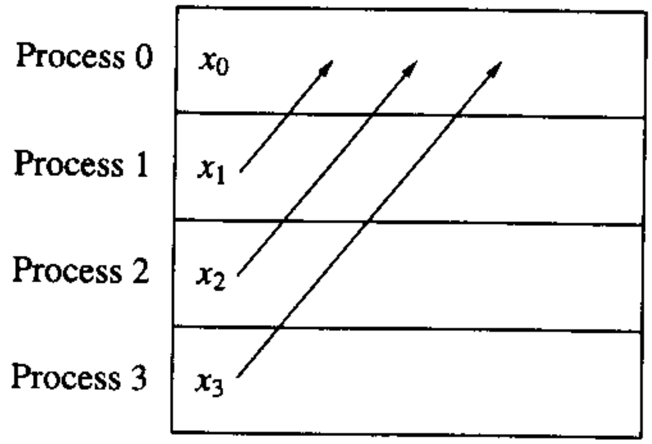
Scatter

```
/* Both arrays allocated by calling program      */
LOCAL_MATRIX_T local_A; /* A 2-dimensional array  */
float          row_segment[];
                        /* An array containing    */
                        /* storage for n/p floats */

/* Assumes n is divisible by p */
MPI_Scatter(&(local_A[0][0]), n/p, MPI_FLOAT,
            row_segment, n/p, MPI_FLOAT,
            0, MPI_COMM_WORLD);
```

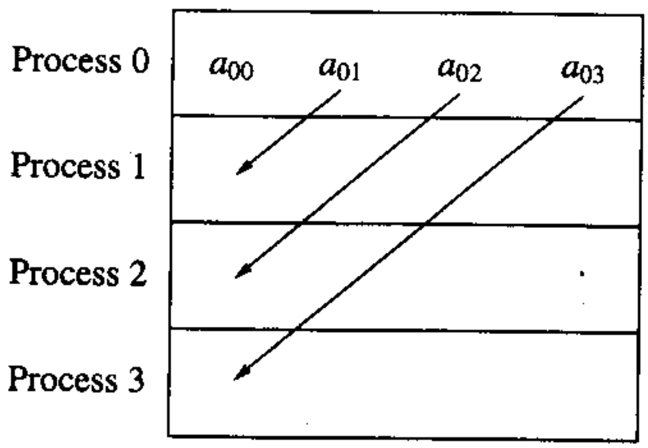
The syntax of `MPI_Scatter` is

```
int MPI_Scatter(
        void*          send_data     /* in  */,
        int            send_count    /* in  */,
        MPI_Datatype   send_type     /* in  */,
        void*          recv_data     /* out */,
        int            recv_count    /* in  */,
        MPI_Datatype   recv_type     /* in  */,
        int            root          /* in  */,
        MPI_Comm       comm          /* in  */)
```

For Matrix Vector Mult. Should we Scatter or Gather?



A gather



A scatter

Issues

- Need to minimize communication
- How does each processor compute its part of y?

Ansewer

Use gather to distrubute copies of X. Compute Y locally

p Gathers

```
for (root = 0; root < p; root++)
    MPI_Gather(local_x, n/p, MPI_FLOAT,
        global_x, n/p, MPI_FLOAT,
        root, MPI_COMM_WORLD);
```

or

```
int MPI_Allgather(
        void*          send_data    /* in  */,
        int            send_count   /* in  */,
        MPI_Datatype   send_type    /* in  */,
        void*          recv_data    /* out */,
        int            recv_count   /* in  */,
        MPI_Datatype   recv_type    /* in  */,
        MPI_Comm       comm         /* in  */)
```

Final Code

```
/* All arrays are allocated in calling program */
void Parallel_matrix_vector_prod(
        LOCAL_MATRIX_T  local_A     /* in  */,
        int             m           /* in  */,
        int             n           /* in  */,
        float           local_x[]   /* in  */,
        float           global_x[]  /* in  */,
        float           local_y[]   /* out */,
        int             local_m     /* in  */,
        int             local_n     /* in  */) {

    /* local_m = n/p, local_n = n/p */

    int i, j;

    MPI_Allgather(local_x, local_n, MPI_FLOAT,
                  global_x, local_n, MPI_FLOAT,
                  MPI_COMM_WORLD);
    for (i = 0; i < local_m; i++) {
        local_y[i] = 0.0;
        for (j = 0; j < n; j++)
            local_y[i] = local_y[i] +
                         local_A[i][j]*global_x[j];
    }
}  /* Parallel_matrix_vector_prod */
```

What if you need to "personalize" the data sent to each processor?

```
int MPI_Alltoall(
        void*          send_buffer  /* in  */,
        int            send_count   /* in  */,
        MPI_Datatype   send_type    /* in  */,
        void*          recv_buffer  /* out */,
        int            recv_count   /* in  */,
        MPI_Datatype   recv_type    /* in  */,
        MPI_Comm       comm         /* in  */)
```

what if the size of the data varies?

```
int MPI_Alltoallv(
        void*          send_buffer            /* in  */,
        int            send_counts[]          /* in  */,
        int            send_displacements[]   /* in  */,
        MPI_Datatype   send_type              /* in  */,
        void*          recv_buffer            /* out */,
        int            recv_counts[]          /* in  */,
        int            recv_displacements[]   /* in  */,
        MPI_Datatype   recv_type              /* in  */,
        MPI_Comm       comm                   /* in  */)
```