Introduction to MPI: Lecture 1

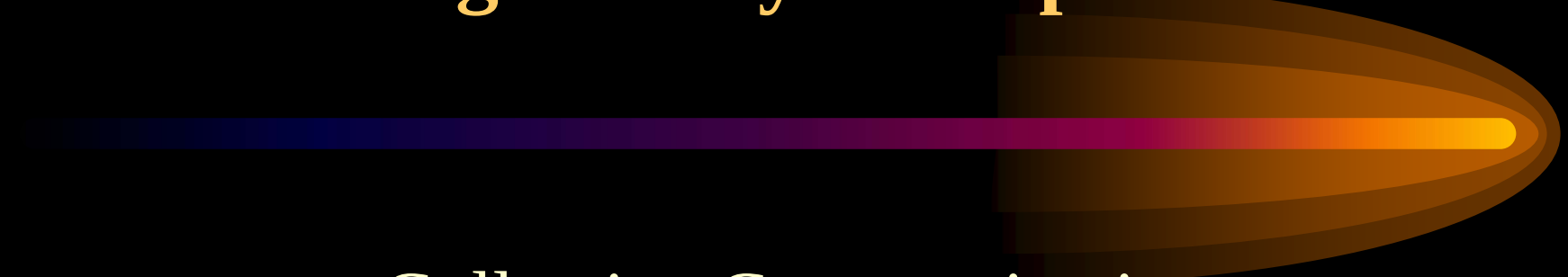
Jun Ni, Ph.D. M.E.

Associate Professor
Department of Radiology
Carver College of Medicine

Information Technology Services

The University of Iowa

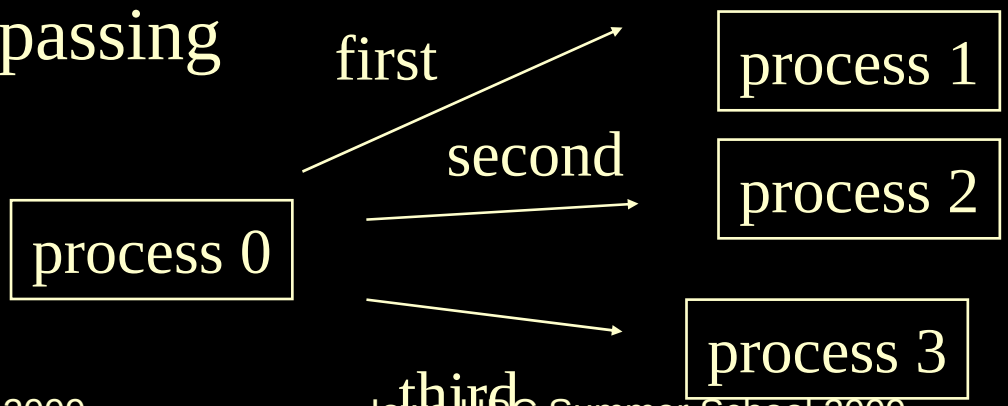
Learning MPI by Examples: Part IV



Collective Communication

Learning MPI by Examples: Part IV

- Point-to-point communication (previous examples)
  - message passing from one process to another, one by one
  - many processes are idle when message is passing

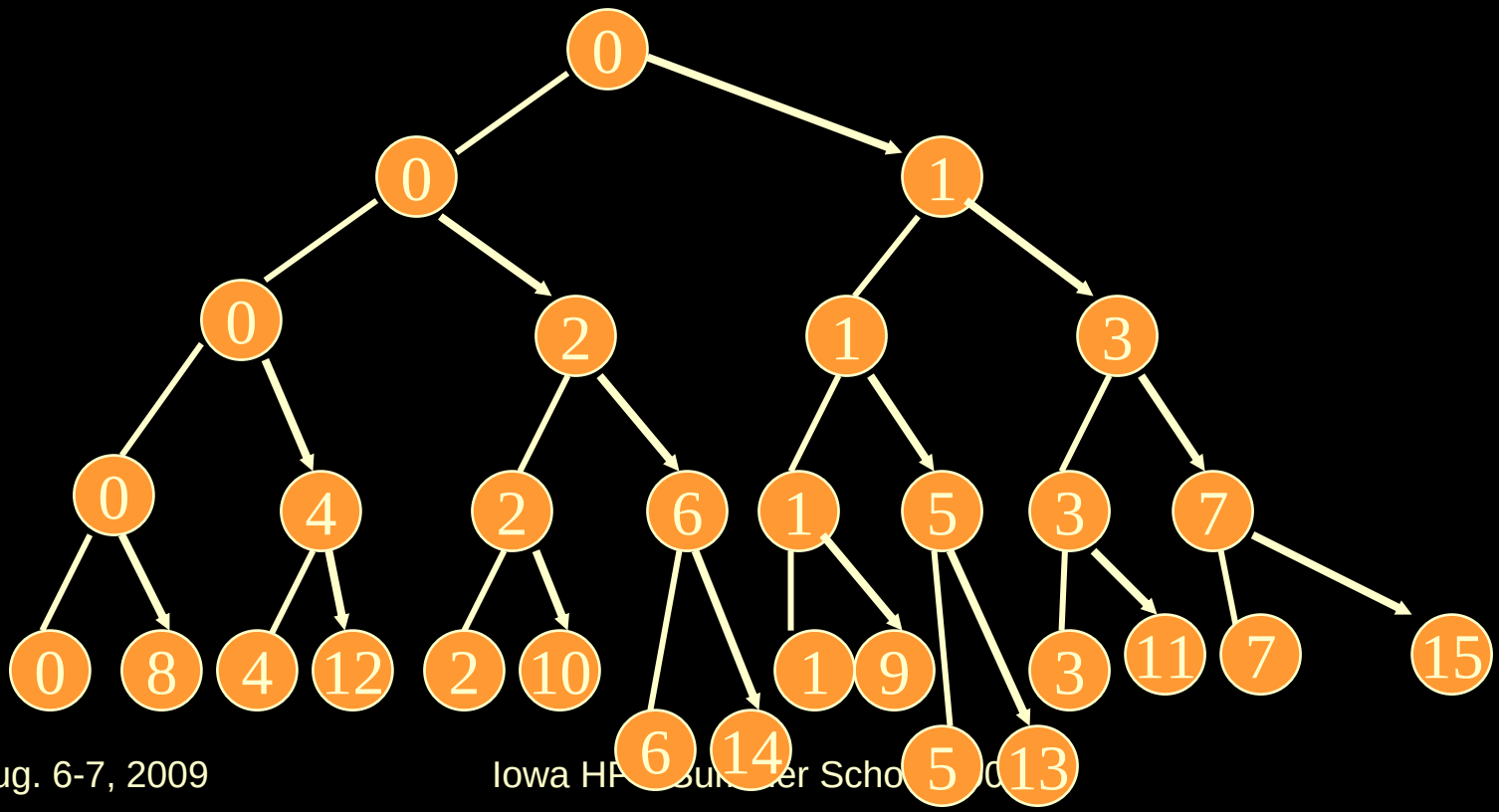


Learning MPI by Examples: Part IV

- Collective communication
  - collaborative ways to pass message among processes
    - tree-structured communication
    - broadcast communication
    - gather scatter communication

Learning MPI by Examples: Part IV

- Tree-structured communication



Learning MPI by Examples: Part IV

- Tree-structured communication
  - process 0 sends the data to process 1
  - process 0 and 1 send the data to process 2 and 3, respectively
  - process 0, 1, 2, and 3 send the data to process 4, 5, 6, and 7, respectively
  - process 0, 1,2,3,4,5,6, and 7 send the data to process 8,9,10,11,12,13,14,and 15, respectively
  - ...

Learning MPI by Examples: Part IV

- Tree-structured communication
  - In general, tree-structured communication reduces p-1 message passing steps to $\log_2(p)$ steps.
  - The reduction radio is $\log_2(p)/(p-1)$. e.g. p=1024, p-1=1023, $\log_2(1024)$ =10
  - $\log_2(p)/(p-1)$=10/1023=0.98%.
  - That means, if we have 1024 processes, we can save 99.2 duration time when we pass the data

Learning MPI by Examples: Part IV

- Tree-structured communication
  - modify Get_data() function using tree-distribution scheme
    - use I_receive() function to check whether the process receive data or not
    - use I_send() function to check whether the process send data or not
  - Nice but complicated algorithm
  - It is strongly dependent on the architecture and topology of system.

```
/* get_data1.c -- Parallel Trapezoidal Rule; uses a hand-coded
 *     tree-structured broadcast.
 *
 * Input:
 *    a, b: limits of integration.
 *    n: number of trapezoids.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Notes:
 *    1.  f(x) is hardwired.
 *    2.  the number of processes (p) should evenly divide
 *        the number of trapezoids (n).
 *
 */
#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;          /* Left endpoint                  */
    float       b;          /* Right endpoint                 */
    int         n;          /* 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      */
```

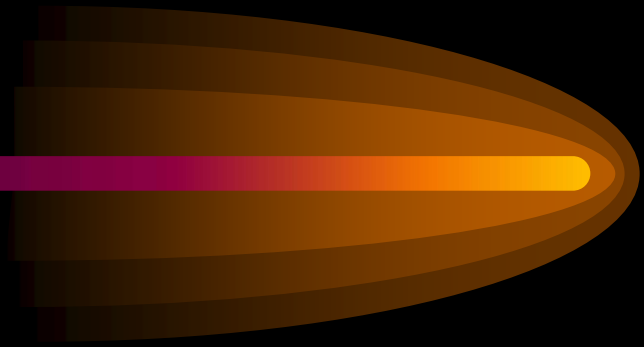
```
float       total;     /* Total integral           */
int         source;    /* Process sending integral  */
int         dest = 0;  /* All messages go to 0      */
int         tag = 0;
MPI_Status  status;
void Get_data1(float* a_ptr, float* b_ptr,
       int* n_ptr, int my_rank,  int p);
float Trap(float 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);
```

```
Get_data1(&a, &b, &n, my_rank, 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' 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  */

/*********************************************************/
/* Ceiling of log_2(x) is just the number of times
 * 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 */)
{
```

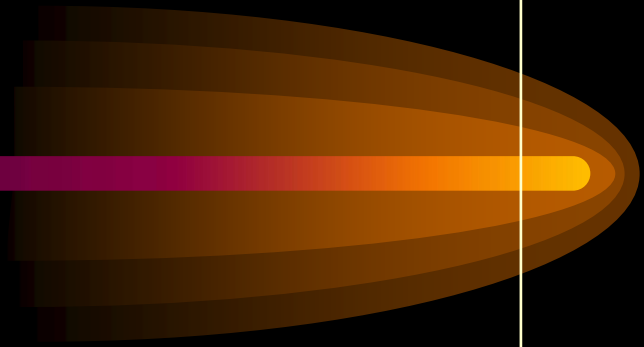
```
 MPI_Send(&a, 1, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
   MPI_Send(&b, 1, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);
   MPI_Send(&n, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
} /* Send */


/*********************************************************/
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 */


/*********************************************************/
/* Function Get_data1
 * 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:
```

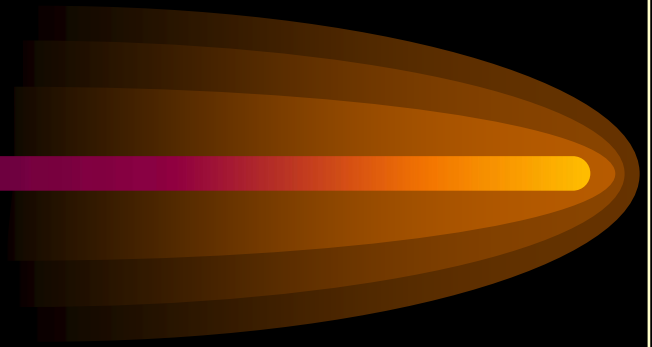
```
*     1.  Process 0 prompts user for input and
 *         reads in the values.
 *     2.  Process 0 sends input values to other
 *         processes using hand-coded tree-structured
 *         broadcast.
 */
void Get_data1(
      float*  a_ptr    /* out */,
      float*  b_ptr    /* out */,
      int*    n_ptr    /* out */,
      int     my_rank  /* in  */,
      int     p        /* in  */)
{

   int source;
   int dest;
   int stage;
```

```
    int Ceiling_log2(int  x);
    int I_receive( int stage, int my_rank, int*  source_ptr);
    int I_send(int stage, int my_rank, int p, int* dest_ptr);
    void Send(float a, float b, int n, int dest);
    void Receive(float* a_ptr, float* b_ptr, int* n_ptr, int source);

    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);
}
```

```
/*****************************************************************/
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_val = x*x;
    return return_val;
} /* f */
```

Learning MPI by Examples: Part IV

- Result:

```
%cc get_data1.c -lmpi
%mpirun -np 8 a.out
Enter a, b, and n
0 1 1024
With n = 1024 trapezoids, our estimate
```

Learning MPI by Examples: Part IV

- Collective communication using broadcasting
  - not point to point communication
  - one process sends data to every processes or broadcast to all of the processes
  - It is called group communication
  - The communication occurs within one communicator

Learning MPI by Examples

- Parallel programming with collective communication using broadcasting
  - one process share data with other processes

Learning MPI by Examples

- Using
  - MPI_Init and MPI_finalize
  - MPI_Comm_rank and MPI_Comm_size
  - MPI_Bcast()

Learning MPI by Examples

- MPI_Bcast() Syntax

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

Learning MPI by Examples

- MPI_Bcast() mechanism:
  - send a copy of the data in messageon the process with rank of root to each process in the communicator comm
  - The message is received by all the processes which are in the same communicator
  - The broadcast message can not be received with MPI_Recv()
  - no tag needed

Learning MPI by Examples

- MPI_Bcast() mechanism:
  - example: same numerical integration
  - technique: broadcast three input parameters, a, b, and n values

```
/* get_data2.c -- Parallel Trapezoidal Rule.
* Uses 3 calls to MPI_Bcast to distribute input data.
 *
 * Input:
 *    a, b: limits of integration.
 *    n: number of trapezoids.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Notes:
 *    1.  f(x) is hardwired.
 *    2.  the number of processes (p) should evenly divide
 *        the number of trapezoids (n).
 *
*/
```

```
#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;                  /* Left endpoint            */
    float       b;                  /* Right endpoint           */
    int         n;                  /* 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;

 void Get_data2(float* a_ptr, float* b_ptr, int* n_ptr, int my_rank);
 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);
```

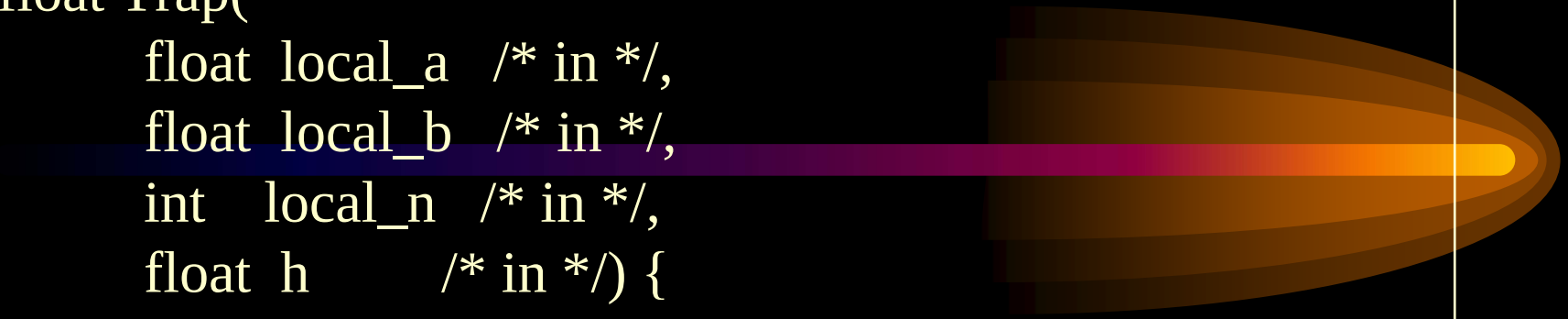
```
Get_data2(&a, &b, &n, my_rank);

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' 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);
}

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  */

/*********************************************************************/
/* Function Get_data2
 * 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 using three calls to MPI_Bcast. */
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 */
```

```
/*****************************************************/
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_val = x*x;
    return return_val;
} /* f */
```

Learning MPI by Examples

- Result:

```
Enter a, b, and n
0 1 1024
With n = 1024 trapezoids, our estimate
of the integral from 0.000000 to 1.000000 = 0.333333
```

Learning MPI by Examples

- Reduce
  - People can use reversed-order of tree structured communication pattern
  - Use MPI_Reduce()
  - a global reduction operation, all the processes in a communicator contribute data that is combined using a binary operation.

Learning MPI by Examples

- Reduce
  - reduction to perform summation by master process using MPI_Reduce () with MPI_SUM (others are MPI_MAX, MPI_MIN, MPI_MAXLOC, MPI_MINLOC)

Learning MPI by Examples

- Reduce Syntax:

```
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*/)
```

Learning MPI by Examples

- MPI_Reduce() must be called by all the processes
- MPI_Operator:

MPI_Max	Maximum
MPI_MIN	Minimum
MPI_SUM	Summation
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_MAXLOC	Maximum and location of maximum
MPI_ MINLOC	Minimum and location of minimum

You can define MPI additional operators

```
/* reduce.c -- Parallel Trapezoidal Rule.  Uses 3 calls to MPI_Bcast to
 *    distribute input.  Also uses MPI_Reduce to compute final sum.
 *
 * Input:
 *    a, b: limits of integration.
 *    n: number of trapezoids.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Notes:
 *    1.  f(x) is hardwired.
 *    2.  the number of processes (p) should evenly divide
 *        the number of trapezoids (n).
 *
 * See Chap. 5, pp. 73 & ff. in PPMPI.
 */
#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;         /* Left endpoint             */
    float       b;         /* Right endpoint            */
    int         n;         /* 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            */
```

```
void Get_data2(float* a_ptr, float* b_ptr, int* n_ptr, int my_rank);
 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);

 Get_data2(&a, &b, &n, my_rank);

 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' 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 */
MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
    MPI_SUM, 0, 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  */

/*********************************************************/
/* Function Get_data2
 * 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 using three calls to MPI_Bcast.
 */
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 */
```

```
/*****************************************************************/
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_val = x*x;
    return return_val;
} /* f */
```

Result:
mpirun -np 8 a.out
Enter a, b, and n
0 1 1024
With n = 1024 trapezoids, our estimate
of the integral from 0.000000 to 1.000000 = 0.333333

Numerical integration of Newton-Cotes methods
(Using MPI_Bcast() and MPI_Reduce()

```
/* seosl_intNC -- Parallel version of numerical integration
   with Newton-Cotes methods, which includes
     rectangle rule (one-point rule),  tprapezoidal rule (two-point rule),
     Simpson rule(three-point rule)
     using MPI_Bcast() and MPI_reduce()
*/
#include <stdio.h>
#include "mpi.h"
#include <math.h>

main(int argc, char** argv)
{
    int         my_rank;
    int         p;
    float       a = 0.0, b=1.0, h;
    int         n = 2048;
    int         mode=3;   /* mode=1,2,3
                             rectangle, trapezoidal, and Simpson */
```

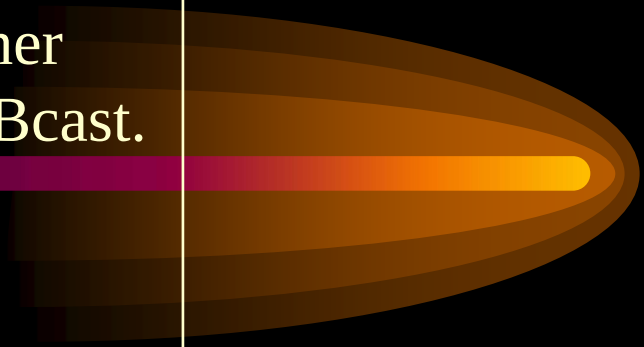
```
float      local_a, local_b, local_h;
   int        local_n;

   float      local_integral, integral;

/* function prototypes */
   void Get_data02(float* a_ptr, float* b_ptr,
                   int* n_ptr, int my_rank, int p, int *mode_ptr);
   float rect(float local_a, float local_b, int local_n, float h);
   float trap(float local_a, float local_b, int local_n, float h);
   float simp(float local_a, float local_b, int local_n, float h);

/* MPI starts   */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   Get_data02(&a, &b, &n, my_rank, p, &mode)
```

```
h = (b-a)/n;
local_n = n/p;
local_a = a + my_rank*(b-a)/p;
local_b = a + (my_rank+1)*(b-a)/p;
local_h = h;

switch(mode)
{
case(1):
  local_integral = rect(local_a, local_b, local_n, local_h);
  break;
case(2):
  local_integral = trap(local_a, local_b, local_n, local_h);
  break;
case(3):
  local_integral = simp(local_a, local_b, local_n, local_h);
}
```

```
MPI_Reduce(&local_integral,&integral,1,MPI_FLOAT,
           MPI_SUM,0,MPI_COMM_WORLD);

if(my_rank==0)
{
 if (mode==1)
   printf("Rectangle rule (0-point rule) is selected\n");
 else if (mode==2)
   printf("Trapezodial rule (2-point rule) is selected\n");
 else    /* defaulted */
   printf("Simpson rule (3-point rule) is selected\n");

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

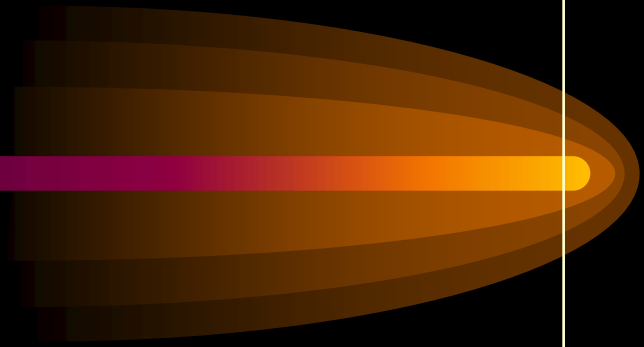
```
/* MPI finished */
   MPI_Finalize();
}

/*********************************************************/
/* Function Get_data02
 * 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.
       3.  int* mode_ptr: pointer to mode of rule of
Newton-Cotes methods
 * Algorithm:
```

```
*     1.  Process 0 prompts user for input and
 *         reads in the values.
 *     2.  Process 0 sends input values to other
 *         processes using four calls to MPI_Bcast.
 */
void Get_data02(
        float*  a_ptr    /* out */,
        float*  b_ptr    /* out */,
        int*    n_ptr    /* out */,
        int     my_rank  /* in  */,
        int     p        /* in  */,
        int*    mode_ptr /* out */)
{

    if (my_rank == 0)
    {
      do
      {
```



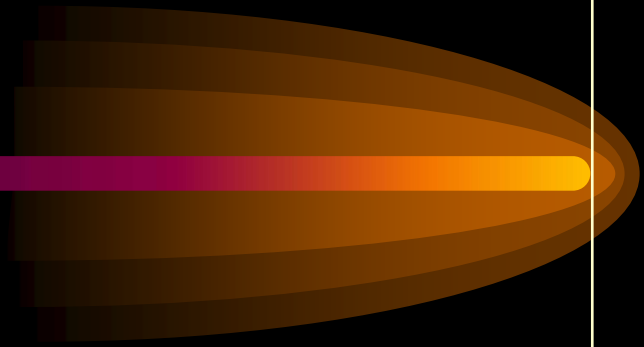
```
    printf("Enter a, b, n(1024), and mode(1--rect, 2-- trap, 3-- simp):\n"
    scanf("%f %f %d %d", a_ptr, b_ptr, n_ptr, mode_ptr);
    } while (*mode_ptr<1 || *mode_ptr>3);

  }

  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);
  MPI_Bcast(mode_ptr,1,MPI_INT,0,MPI_COMM_WORLD);

} /* Get_data02*/
```

```
float rect( float local_a, float local_b, int local_n, float local_h )
{
    float local_integral;
    float x;
    int i;

    float f(float x);

    local_integral = f(local_a);
    x = local_a;
    for (i = 1; i <= local_n-1; i++)
    {
        x = x + local_h;
        local_integral += f(x);
    }
    local_integral *=local_h;
    return local_integral;
}
```

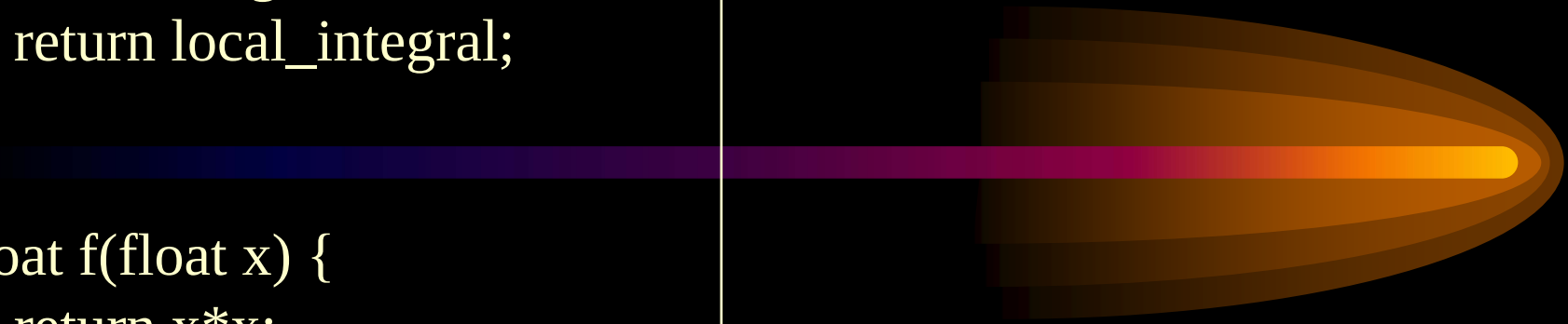


```
float trap( float local_a, float local_b, int local_n, float local_h )
{
    float local_integral;
    float x;
    int i;

    float f(float x);
    local_integral = f(local_a) + f(local_b);
    x = local_a;
    for (i = 1; i <= local_n-1; i++)
    {
        x = x + local_h;
        local_integral += 2.0*f(x);
    }
    local_integral *=local_h/2.0;
    return local_integral;
}
```

```
float simp( float local_a, float local_b, int local_n, float local_h )
{
    float local_integral;
    float x;
    int i;

    float f(float x);

    local_integral = f(local_a) + f(local_b);
    x = local_a;
    for (i = 1; i < local_n; i++)
    {
      x = x + local_h;
      if (i % 2 == 0)              /* if i is even */
         local_integral = local_integral + 2 * f(x);
      else                         /* if i is odd  */
         local_integral = local_integral + 4 * f(x);
    }
```



```
    local_integral *=local_h/3.0;
    return local_integral;
}

float f(float x) {
    return x*x;
}
```



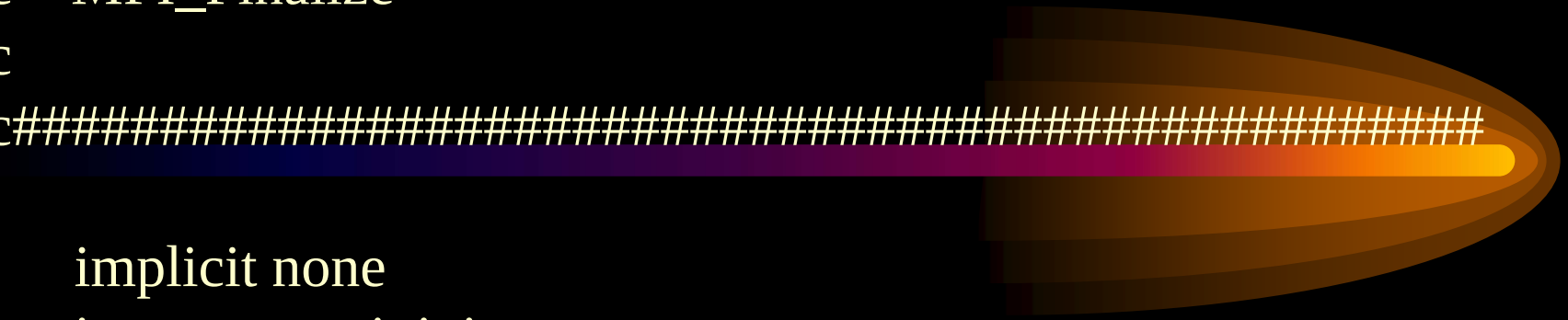
```
mpirun -np 8 a.out
Enter a, b, n(1024), and mode(1--rect, 2-- trap, 3-- simp):
0 1 1024 2
Trapezodial rule (2-point rule) is selected
With n = 1024, the total integral from 0.000000 to 1.000000 =
0.333333
```

Learning MPI by Examples

- Example of parallel programming using collective communication (F77)

```
 Program Example1_3
c##############################################################
c  This is an MPI example on parallel integration
c  It demonstrates the use of :
c * MPI_Init
c * MPI_Comm_rank
c * MPI_Comm_size
c * MPI_Bcast
c * MPI_Reduce
```

```
c * MPI_SUM
c * MPI_Finalize
c
c#############################################################

      implicit none
      integer n, p, i, j, ierr, master
      real h, result, a, b, integral, pi
      include "mpif.h"  !! This brings in pre-defined MPI constants, ...
      integer Iam, source, dest, tag, status(MPI_STATUS_SIZE)
      real my_result
     data master/0/

c**Starts MPI processes ...
       call MPI_Init(ierr)                          !! starts MPI
      call MPI_Comm_rank(MPI_COMM_WORLD, Iam, ierr)
                                                   !! get current process id
```



```
   call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
              !! get number of processes
    pi = acos(-1.0)   !!  = 3.14159...
    a = 0.0           !! lower limit of integration
    b = pi*1./2.      !! upper limit of integration
    dest = 0          !! define the process that computes the final result
    tag = 123         !! set the tag to identify this particular job
    if(Iam .eq. master) then
       print *,'The requested number of processors =',p
       print *,'enter number of increments within each process'
       read(*,*)n
       endif


c**Broadcast "n" to all processes
 call MPI_Bcast(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
      h = (b-a)/n/p     !! length of increment
      my_result = integral(a,Iam,h,n)
```

```
write(*,"('Process ',i2,' has the partial result of',f10.6)")
&          Iam,my_result
 call MPI_Reduce(my_result, result, 1, MPI_REAL, MPI_SUM,
&     dest, MPI_COMM_WORLD, ierr)

 if(Iam .eq. master) then
   print *,'The result =',result
 endif

 call MPI_Finalize(ierr)                    !! let MPI finish up ...
 stop
 end

 real function integral(a,i,h,n)
 implicit none
 integer n, i, j
```

```
real h, h2, aij, a
real fct, x
fct(x) = cos(x)              !! kernel of the integral
integral = 0.0               !! initialize integral
h2 = h/2.
do j=0,n-1                   !! sum over all "j" integrals
  aij = a + (i*n +j)*h       !! lower limit of "j" integral
  integral = integral + fct(aij+h2)*h
enddo
return
end
```

Result:

```
% /bin/time mpirun -np 8 example1_3
 The requested number of processors =           8
 enter number of increments within each process
20
Process  0 has the partial result of  0.195091
Process  7 has the partial result of  0.019215
Process  1 has the partial result of  0.187594
Process  4 has the partial result of  0.124363
Process  5 has the partial result of  0.092410
Process  6 has the partial result of  0.056906
Process  2 has the partial result of  0.172887
Process  3 has the partial result of  0.151537
 The result =    1.000004
real 24.721
user 0.005
sys  0.053
```

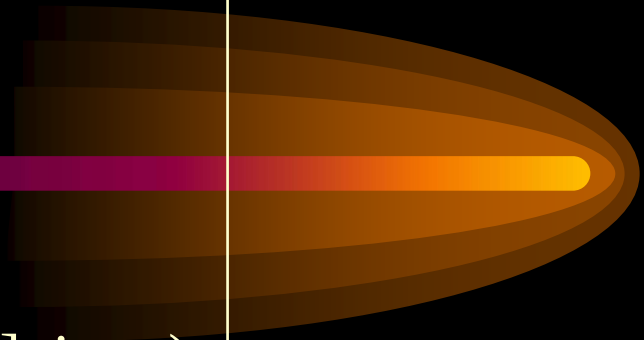
```
% /bin/time mpirun -np 8 example1_3
 The requested number of processors =          8
 enter number of increments within each process
40
Process  0 has the partial result of  0.195091
Process  1 has the partial result of  0.187593
Process  4 has the partial result of  0.124363
Process  5 has the partial result of  0.092410
Process  6 has the partial result of  0.056906
Process  7 has the partial result of  0.019215
Process  3 has the partial result of  0.151537
Process  2 has the partial result of  0.172887
 The result =    1.000001
real 4.381
user 0.005
sys  0.047
```

Learning MPI by Examples

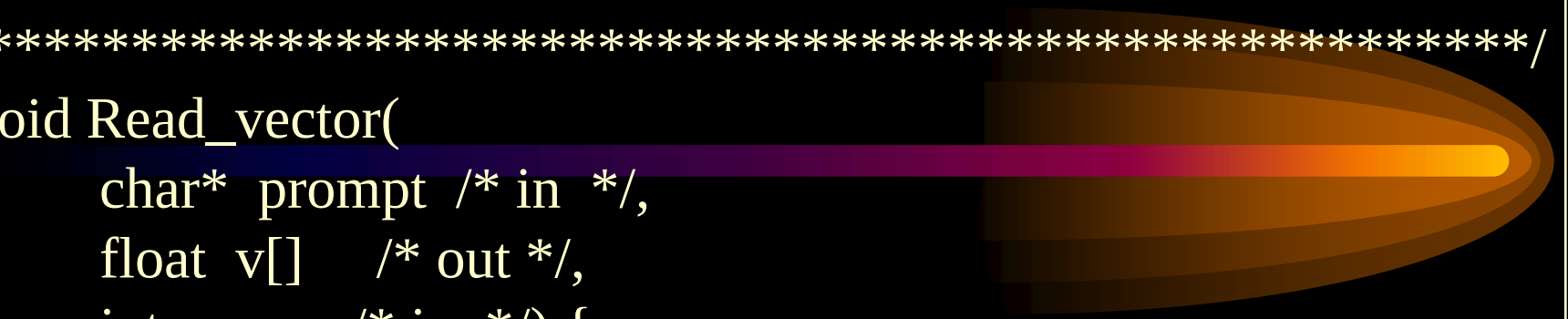
- Serial Dot Production

```
/* serial_dot.c -- compute a dot product on a single processor.
 *
 * Input:
 *     n: order of vectors
 *     x, y:  the vectors
 *
 * Output:
 *     the dot product of x and y.
 *
 * Note:  Arrays containing vectors are statically allocated.
 *
 * See Chap 5, p. 75 in PPMPI.
 */
#include <stdio.h>

#define MAX_ORDER 100
```

```
main() {
    float  x[MAX_ORDER];
    float  y[MAX_ORDER];
    int    n;
    float  dot;

    void Read_vector(char* prompt, float v[], int n);
    float Serial_dot(float x[], float y[], int n);

    printf("Enter the order of the vectors\n");
    scanf("%d", &n);
    Read_vector("the first vector", x, n);
    Read_vector("the second vector", y, n);
    dot = Serial_dot(x, y, n);
    printf("The dot product is %f\n", dot);
}  /* main */
```

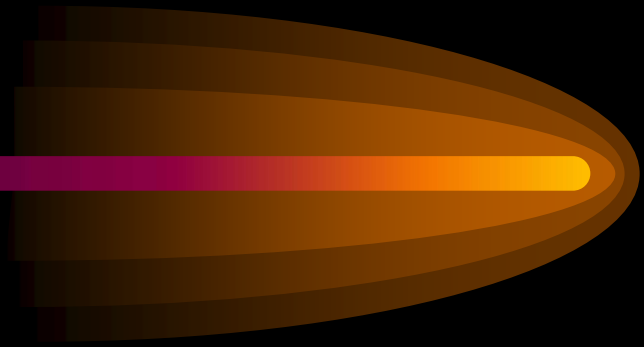


```
/*********************************************************/
void Read_vector(
      char*  prompt  /* in  */,
      float  v[]     /* out */,
      int    n       /* in  */) {
   int i;

   printf("Enter %s\n", prompt);
   for (i = 0; i < n; i++)
      scanf("%f", &v[i]);
}  /* Read_vector */
```



```
/*********************************************************/
float Serial_dot(
        float  x[]  /* in */,
        float  y[]  /* in */,
        int    n    /* in */)
{

   int    i;
   float  sum = 0.0;

   for (i = 0; i < n; i++)
       sum = sum + x[i]*y[i];
   return sum;
} /* Serial_dot */
```

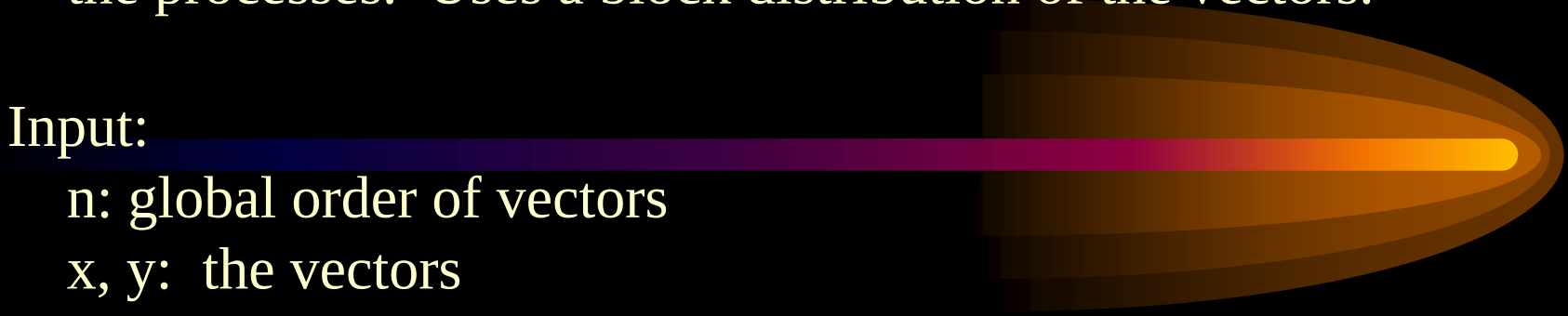
```
% cc serial_dot.c
% a.out
Enter the order of the vectors
4
Enter the first vector
2
3
4
3
Enter the second vector
4
3
2
4
The dot product is 37.000000
```



Learning MPI by Examples

- Parallel _dot.c

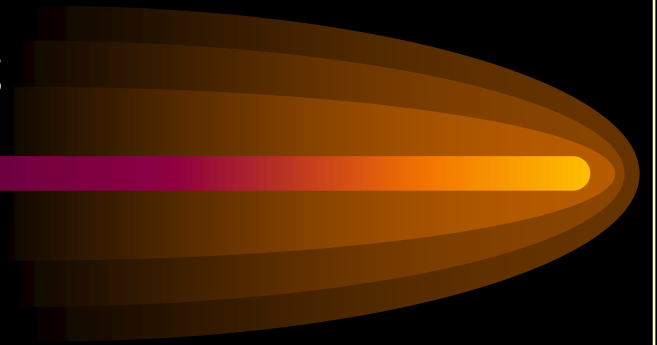
```
/* parallel_dot.c -- compute a dot product of a vector distributed among
 *     the processes.  Uses a block distribution of the vectors.
 *
 * Input:
 *     n: global order of vectors
 *     x, y:  the vectors
 *
 * Output:
 *     the dot product of x and y.
 *
 * Note:  Arrays containing vectors are statically allocated.  Assumes
 *     n, the global order of the vectors, is divisible by p, the number
 *     of processes.
*/
```



```
#include <stdio.h>
#include "mpi.h"
#define MAX_LOCAL_ORDER 100

main(int argc, char* argv[]) {
    float  local_x[MAX_LOCAL_ORDER];
    float  local_y[MAX_LOCAL_ORDER];
    int    n;
    int    n_bar;  /* = n/p */
    float  dot;
    int    p;
    int    my_rank;

    void Read_vector(char* prompt, float local_v[], int n_bar, int p,
          int my_rank);
    float Parallel_dot(float local_x[], float local_y[], int n_bar);
```

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

if (my_rank == 0)
{
    printf("Enter the order of the vectors (n>= %d):\n", p);
    scanf("%d", &n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
n_bar = n/p;

Read_vector("the first vector", local_x, n_bar, p, my_rank);
Read_vector("the second vector", local_y, n_bar, p, my_rank);

dot = Parallel_dot(local_x, local_y, n_bar);
```

```
    if (my_rank == 0)
        printf("The dot product is %f\n", dot);

    MPI_Finalize();
}  /* main */


/*********************************************************************/
void Read_vector(
        char*  prompt     /* in  */,
        float  local_v[]  /* out */,
        int    n_bar      /* in  */,
        int    p          /* in  */,
        int    my_rank    /* in  */)
```



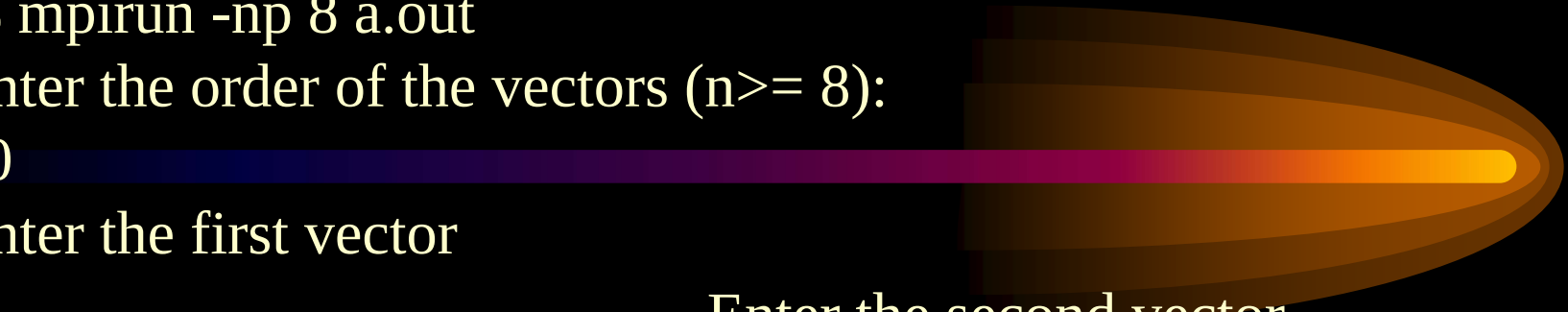
```
{
   int i, q;
   float temp[MAX_LOCAL_ORDER];
   MPI_Status status;
   if (my_rank == 0)
   {
     printf("Enter %s\n", prompt);
     for (i = 0; i < n_bar; i++)
       scanf("%f", &local_v[i]);
     for (q = 1; q < p; q++)
     {
       for (i = 0; i < n_bar; i++)
         scanf("%f", &temp[i]);
       MPI_Send(temp, n_bar, MPI_FLOAT, q, 0,
                                          MPI_COMM_WORLD);
     }
   }
```

```
 else
    {
      MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0,
                        MPI_COMM_WORLD,
          &status);
    }
}  /* Read_vector */


/*********************************************************/
float Serial_dot(
          float  x[]  /* in */,
          float  y[]  /* in */,
          int    n    /* in */)
{
```

```
  int    i;
  float  sum = 0.0;

  for (i = 0; i < n; i++)
        sum = sum + x[i]*y[i];
  return sum;
} /* Serial_dot */


/*********************************************************/
float Parallel_dot(
          float  local_x[]  /* in */,
          float  local_y[]  /* in */,
          int  n_bar        /* in */)
{
  float  local_dot;
```

```
 float  dot = 0.0;
  float  Serial_dot(float x[], float y[], int m);

  local_dot = Serial_dot(local_x, local_y, n_bar);
  MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
      MPI_SUM, 0, MPI_COMM_WORLD);
  return dot;
} /* Parallel_dot */
```

```
% cc parallel_dot.c -lmpi
% mpirun -np 8 a.out
Enter the order of the vectors (n>= 8):
10
Enter the first vector
1
2
3
4
3
5
6
5
Enter the second vector
4
5
6
4
3
5
6
5
The dot product is 143.000000
```



Learning MPI by Examples

- AllReduce()

```
/* parallel_dot1.c -- Computes a parallel dot product.
Uses MPI_Allreduce.
 *
 * Input:
 *     n: order of vectors
 *     x, y:  the vectors
 * Output:
 *     the dot product of x and y as computed by each process.
 * Note:  Arrays containing vectors are statically allocated.
Assumes that
 *     n, the global order of the vectors, is evenly divisible by p, the
 *     number of processes.
 *
 * See Chap 5, pp. 76 & ff in PPMPI.
 */
#include <stdio.h>
#include "mpi.h"
```

```
#define MAX_LOCAL_ORDER 100

main(int argc, char* argv[]) {
    float  local_x[MAX_LOCAL_ORDER];
    float  local_y[MAX_LOCAL_ORDER];
    int    n;
    int    n_bar;  /* = n/p */
    float  dot;
    int    p;
    int    my_rank;

    void Read_vector(char* prompt, float local_v[], int n_bar, int p,
          int my_rank);
    float Parallel_dot(float local_x[], float local_y[], int n_bar);
    void Print_results(float dot, int my_rank, int p);

    MPI_Init(&argc, &argv);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank == 0) {
      printf("Enter the order of the vectors (n>= %d):\n", p);
      scanf("%d", &n);
   }
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   n_bar = n/p;

   Read_vector("the first vector", local_x, n_bar, p, my_rank);
   Read_vector("the second vector", local_y, n_bar, p, my_rank);

   dot = Parallel_dot(local_x, local_y, n_bar);
   Print_results(dot, my_rank, p);
   MPI_Finalize();
}  /* main */
```

```
/*********************************************************/
void Read_vector(
      char*  prompt     /* in  */,
      float  local_v[]  /* out */,
      int    n_bar      /* in  */,
      int    p          /* in  */,
      int    my_rank    /* in  */) {
   int i, q;
   float temp[MAX_LOCAL_ORDER];
   MPI_Status status;

   if (my_rank == 0) {
      printf("Enter %s\n", prompt);
      for (i = 0; i < n_bar; i++)
         scanf("%f", &local_v[i]);
```

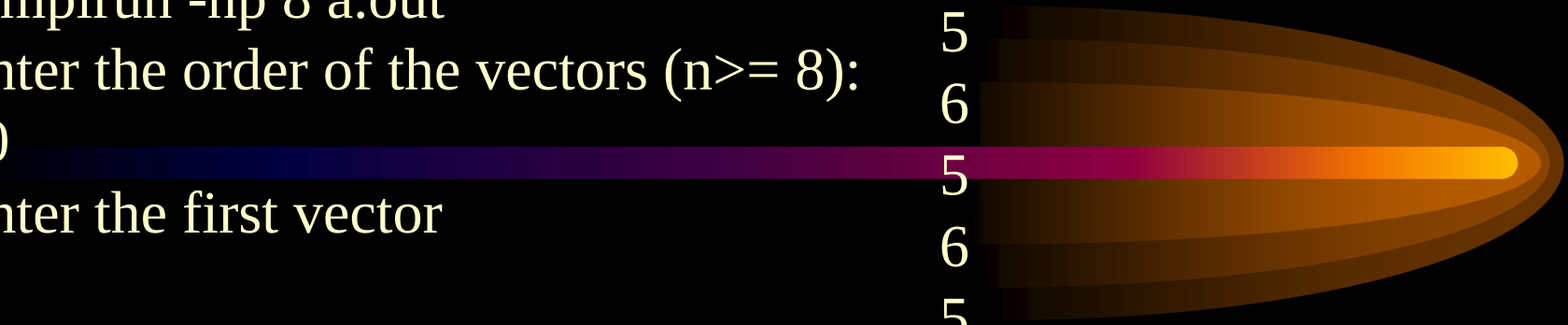
```
  for (q = 1; q < p; q++) {
          for (i = 0; i < n_bar; i++)
             scanf("%f", &temp[i]);
          MPI_Send(temp, n_bar, MPI_FLOAT, q, 0,
                             MPI_COMM_WORLD);
       }
    } else {
       MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0,
                          MPI_COMM_WORLD,
          &status);
    }
}  /* Read_vector */
```

```
/**********************************************************/
float Serial_dot(
        float  x[]  /* in */,
        float  y[]  /* in */,
        int    n    /* in */) {

    int    i;
    float  sum = 0.0;

    for (i = 0; i < n; i++)
        sum = sum + x[i]*y[i];
    return sum;
} /* Serial_dot */
```

```
/*********************************************************/
float Parallel_dot(
        float  local_x[]  /* in */,
        float  local_y[]  /* in */,
        int  n_bar        /* in */) {

    float  local_dot;
    float  dot = 0.0;
    float  Serial_dot(float x[], float y[], int m);

    local_dot = Serial_dot(local_x, local_y, n_bar);
    MPI_Allreduce(&local_dot, &dot, 1, MPI_FLOAT,
        MPI_SUM, MPI_COMM_WORLD);
    return dot;
} /* Parallel_dot */
```

```
void Print_results(float  dot      /* in */,
        int    my_rank  /* in */,
        int    p        /* in */) {
        int         q;          float       temp;
    MPI_Status  status;
    if (my_rank == 0) {
        printf("dot = \n");
        printf("Process 0 > %f\n", dot);
        for (q = 1; q < p; q++) {
            MPI_Recv(&temp, 1, MPI_FLOAT, q, 0,
                        MPI_COMM_WORLD,&status);
            printf("Process %d > %f\n", q, temp);
        }
    } else {MPI_Send(&dot, 1, MPI_FLOAT, 0, 0,
                                                MPI_COMM_WORLD);
    }
}
```

```
% cc parallel_dot1.c -lmpi
%mpirun -np 8 a.out
Enter the order of the vectors (n>= 8):
10
Enter the first vector
1
2
3
4
5
6
5
4
Enter the second vector
3
5
6
5
6
5
4
5
dot =
Process 0 > 151.000000
Process 1 > 151.000000
Process 2 > 151.000000
Process 3 > 151.000000
Process 4 > 151.000000
Process 5 > 151.000000
Process 6 > 151.000000
Process 7 > 151.000000
```



Learning MPI by Examples

- Parallel programming with collective communication using pack and unpack broadcasting
  - master process packs data before broadcasting
  - after other processed receive packed data, they should unpack the data in order to recover the actual data
  - multiple inputs
  - using MPI_Pack and MPI_Unpack

```
Program Example1_4
c##################################################################
c#
c# This is an MPI example on parallel integration
c# It demonstrates the use of :
c#
c# * MPI_Init
c# * MPI_Comm_rank
c# * MPI_Comm_size
c# * MPI_Pack
c# * MPI_Unpack
c# * MPI_Reduce
c# * MPI_SUM, MPI_MAXLOC, and MPI_MINLOC
c# * MPI_Finalize
c#
c##################################################################
```

```
      implicit none
      integer n, p, i, j, ierr, m, master
      real h, result, a, b, integral, pi
      include "mpif.h"   !! This brings in pre-defined MPI constants,
...
      integer Iam, source, dest, tag, status(MPI_STATUS_SIZE)
      real my_result(2), min_result(2), max_result(2)
      integer Nbytes
      parameter (Nbytes=1000, master=0)
      character scratch(Nbytes)   !! needed for
MPI_pack/MPI_unpack; counted in bytes
      integer index, minid, maxid

c**Starts MPI processes ...
      call MPI_Init(ierr)                          !! starts MPI
      call MPI_Comm_rank(MPI_COMM_WORLD, Iam, ierr)
                                   !! get current process id
```

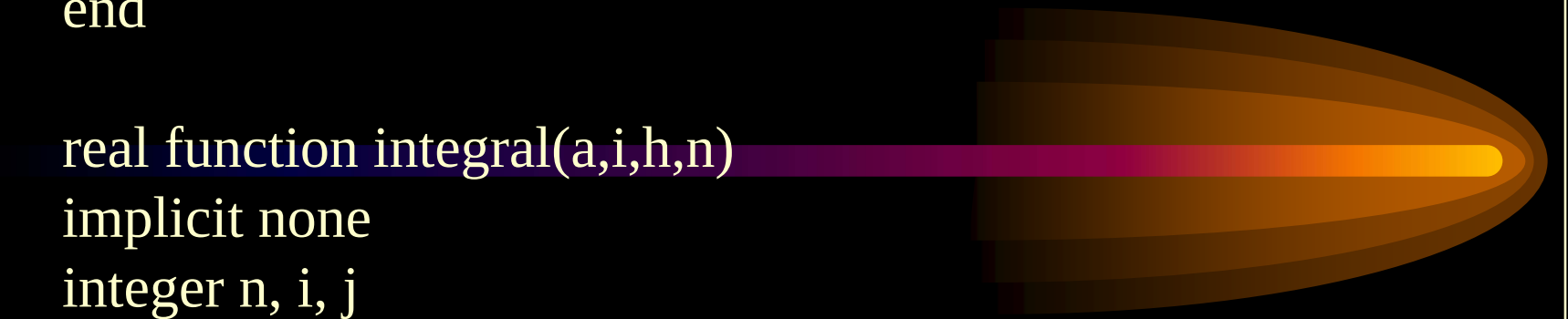
```
call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
                                        !! get number of processes
 pi = acos(-1.0)   !!  = 3.14159...
 dest = 0       !! define the process that computes the final result
 tag = 123      !! set the tag to identify this particular job

 if(Iam .eq. 0) then
   print *,'The requested number of processors =',p
   print *,'enter number of increments within each process'
   read(*,*)n
   print *,'enter a & m'
   print *,' a = lower limit of integration'
   print *,' b = upper limit of integration'
   print *,'   = m * pi/2'
   read(*,*)a,m
   b = m * pi / 2.
```

```
c**to be efficient, pack all things into a buffer for broadcast
      index = 1
      call MPI_Pack(n, 1, MPI_INTEGER, scratch, Nbytes, index,
     &              MPI_COMM_WORLD, ierr)
      call MPI_Pack(a, 1, MPI_REAL, scratch, Nbytes, index,
     &              MPI_COMM_WORLD, ierr)
      call MPI_Pack(b, 1, MPI_REAL, scratch, Nbytes, index,
     &              MPI_COMM_WORLD, ierr)
      call MPI_Bcast(scratch, Nbytes, MPI_PACKED, 0,
&        MPI_COMM_WORLD, ierr)
     else

      call MPI_Bcast(scratch, Nbytes, MPI_PACKED, 0,
     &    MPI_COMM_WORLD, ierr)
c* things received have been packed, unpack into expected
c*  locations
```

```
    index = 1
   call MPI_Unpack(scratch, Nbytes, index, n, 1,
& MPI_INTEGER, MPI_COMM_WORLD, ierr)
   call MPI_Unpack(scratch, Nbytes, index, a, 1, MPI_REAL,
&                  MPI_COMM_WORLD, ierr)
   call MPI_Unpack(scratch, Nbytes, index, b, 1, MPI_REAL,
&                  MPI_COMM_WORLD, ierr)
 endif
 h = (b-a)/n/p     !! length of increment
 my_result(1) = integral(a,Iam,h,n)
 my_result(2) = Iam
 write(*,"('Process ',i2,' has the partial result of',f10.6)")
&         Iam,my_result(1)

 call MPI_Reduce(my_result, result, 1, MPI_REAL,
& MPI_SUM, dest, MPI_COMM_WORLD, ierr)
    !! data reduction by way of MPI_SUM
```

```
 call MPI_Reduce(my_result, min_result, 1, MPI_2REAL,
& MPI_MINLOC, dest, MPI_COMM_WORLD, ierr)
                  !! data reduction by way of MPI_MINLOC
 call MPI_Reduce(my_result, max_result, 1, MPI_2REAL,
& MPI_MAXLOC, dest, MPI_COMM_WORLD, ierr)
               !! data reduction by way of MPI_MAXLOC
 if(Iam .eq. master) then
   print *,'The result =',result
   maxid = max_result(2)
   print *,'Proc',maxid,' has  largest integrated value of',
&     max_result(1)
   minid = min_result(2)
   print *,'Proc',minid,' has smallest integrated value of',
&     min_result(1)
 endif
 call MPI_Finalize(ierr)          !! let MPI finish up ...
```

```
 stop
 end

 real function integral(a,i,h,n)
 implicit none
 integer n, i, j
 real h, h2, aij, a
 real fct, x
fct(x) = cos(x)              !! kernel of the integral
 integral = 0.0              !! initialize integral
 h2 = h/2.
 do j=0,n-1                  !! sum over all "j" integrals
   aij = a + (i*n +j)*h      !! lower limit of "j" integral
   integral = integral + fct(aij+h2)*h
 enddo
 return
 end
```



Learning MPI by Examples

- Parallel programming with collective communication using gather and scatter
  - Gather and scatter functions can be used to transfer data from different processes to one root process, or transfer data from root process to others
  - Using MPI_Gather and MPI_Scatter

Program Example1_5

```
c##############################################################
c#
c# This is an MPI example on parallel integration
c# It demonstrates the use of :
c#
c# * MPI_Init
c# * MPI_Comm_rank
c# * MPI_Comm_size
c# * MPI_Bcast
c# * MPI_Gather
c# * MPI_Scatter
c# * MPI_Finalize
c#
c##############################################################
```

```
      implicit none
      integer n, p, i, j, ierr, master
      real h, result, a, b, integral, pi
      include "mpif.h"  !! This brings in pre-defined MPI constants, ...
      integer Iam
      real my_result, buf(50), tmp
      parameter (master=0)
c**Starts MPI processes ...
      call MPI_Init(ierr)                          !! starts MPI
      call MPI_Comm_rank(MPI_COMM_WORLD, Iam, ierr)
                !! get current process id
      pi = acos(-1.0)   !!  = 3.14159...
      a = 0.0           !! lower limit of integration
      b = pi*1./2.      !! upper limit of integration
      n = 500
      h = (b-a)/n/p     !! length of increment
      my_result = integral(a,Iam,h,n)
```

```
write(*,"('Process ',i2,' has the partial result of',f10.6)")
&          Iam,my_result
 call MPI_Gather(my_result, 1, MPI_REAL, buf, 1, MPI_REAL, 0,
&          MPI_COMM_WORLD, ierr)
 call MPI_Scatter(buf, 1, MPI_REAL, tmp, 1, MPI_REAL, 0,
&                 MPI_COMM_WORLD, ierr)
 print *,'Result sent back from buf =', tmp
 if(Iam .eq. master) then
   result = 0.0
   do i=1,p
     result = result + buf(i)
   enddo
   print *,'The result =',result
 endif
 call MPI_Finalize(ierr)                    !! let MPI finish up ...
 stop
 end
```

```
real function integral(a,i,h,n)
implicit none
integer n, i, j
real h, h2, aij, a
real fct, x
fct(x) = cos(x)                  !! kernel of the integral
integral = 0.0                   !! initialize integral
h2 = h/2.
do j=0,n-1                        !! sum over all "j" integrals
  aij = a + (i*n +j)*h            !! lower limit of "j" integral
  integral = integral + fct(aij+h2)*h
enddo
return
end
```

Topics in Next MPI workshop

- Advanced point-to-point communication
- Advanced collective communication
- User-defined data-type and package
- Function of communicator
- Process Topologies
- Performance
- Parallel environment and debugging

Reference

- MPI Forum (http://www.mpi-forum.org/) and MPI documents (http://www.mpi-forum.org/docs/docs.html)
- MPI standards (http://www.unix.mcs.anl.gov/mpi/)
- Mississippi State University NSF Engineering Research Center (http://www.erc.msstate.edu/labs/hpcl/projects/mpi/)

Reference

- "MPI, the Complete Reference," MIT Press, 1995, by Marc Snir Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra (http://www.netlib.org/utk/papers/mpi-book/mpi-book.html)
- MPI SoftTech Inc. (http://www.mpi-softtech.com/)