Introduction to MPI

Kadin Tseng

Scientific Computing and Visualization Group

Boston University

June 2010
Parallel Computing Paradigms

- Parallel Computing Paradigms
  - Message Passing (MPI, …)
    - Distributed or shared memory
  - Directives (OpenMP, …)
    - Shared memory only
  - Multi-Level Parallel programming (MPI + OpenMP)
    - Shared (and distributed) memory
MPI Topics to Cover

- Fundamentals
- Basic MPI Functions
- Point-to-point Communications
- Compilations and Executions
- Collective Communications
- Dynamic Memory Allocations
- MPI Timer
- Cartesian Topology
What is MPI?

• MPI stands for Message Passing Interface.
• It is a library of subroutines/functions, not a computer language.
• Programmer writes fortran/C code, insert appropriate MPI subroutine/function calls, compile and finally link with MPI message passing library.
• In general, MPI codes run on shared-memory multi-processors, distributed-memory multi-computers, cluster of workstations, or heterogeneous clusters of the above.
• MPI-2 functionalities are available.
Why MPI?

• To provide efficient communication (message passing) among networks/clusters of nodes
• To enable more analyses in a prescribed amount of time.
• To reduce time required for one analysis.
• To increase fidelity of physical modeling.
• To have access to more memory.
• To enhance code portability; works for both shared- and distributed-memory.
• For “embarrassingly parallel” problems, such as many Monte-Carlo applications, parallelizing with MPI can be trivial with near-linear (or superlinear) speedup.
MPI Preliminaries

- MPI’s pre-defined constants, function prototypes, etc., are included in a header file. This file must be included in your code wherever MPI function calls appear (in “main” and in user subroutines/functions):
  - #include “mpi.h” for C codes
  - #include “mpi++.h” * for C++ codes
  - include “mpif.h” for f77 and f9x codes

- MPI_Init must be the first MPI function called.
- Terminates MPI by calling MPI_Finalize.
- These two functions must only be called once in user code.

* More on this later …
• C is case-sensitive language. MPI function names always begin with "MPI_", followed by specific name with leading character capitalized, e.g., MPI_Comm_rank. MPI pre-defined constant variables are expressed in upper case characters, e.g., MPI_COMM_WORLD.

• Fortran is not case-sensitive. No specific case rules apply.

• MPI fortran routines return error status as last argument of subroutine call, e.g., call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

• Error status is returned as “int” function value for C MPI functions, e.g.,
  int ierr = MPI_Comm_rank(MPI_COMM_WORLD, rank);
What is A Message?

- Collection of data (array) of MPI data types
  - Basic data types such as int/integer, float/real
  - Derived data types
- Message “envelope” – source, destination, tag, communicator
Modes of Communication

- Point-to-point communication
  - Blocking – returns from call when task completes
    - Several send modes; one receive mode
  - Nonblocking – returns from call without waiting for task to complete
    - Several send modes; one receive mode
- Collective communication
MPI Data Types vs C Data Types

• MPI types -- C types
  – MPI_INT – signed int
  – MPI_UNSIGNED – unsigned int
  – MPI_FLOAT – float
  – MPI_DOUBLE – double
  – MPI_CHAR – char
  – . . .
MPI vs Fortran Data Types

- `MPI_INTEGER` – INTEGER
- `MPI_REAL` – REAL
- `MPI_DOUBLE_PRECISION` – DOUBLE PRECISION
- `MPI_CHARACTER` – CHARACTER(1)
- `MPI_COMPLEX` – COMPLEX
- `MPI_LOGICAL` – LOGICAL
- ...
MPI Data Types

- MPI_PACKED
- MPI_BYTE
- User-derived types
Some MPI Implementations

There are a number of implementations:

- **MPICH** (ANL)
- LAM (UND/OSC)
- CHIMP (EPCC)
- OpenMPI (installed on Katana)
- Vendor implementations (SGI, IBM, …)
- Codes developed under one implementation should work on another without problems.
- Job execution procedures of implementations may differ.
Integrate \( \cos(x) \) by Mid-point Rule

\[
\int_{0}^{2\pi} \cos(x) \, dx
\]

- \( n \) is number of increments per partition (or processor)
- \( p \) is number of partitions
- \( h \) is increment width
Example 1 (Integration)

We will introduce some fundamental MPI function calls through the computation of a simple integral by the Mid-point rule.

\[
\int_{a}^{b} \cos(x) \, dx = \sum_{i=0}^{p-1} \sum_{j=0}^{n-1} \int_{a_i + j \cdot h}^{a_i + (j+1) \cdot h} \cos(x) \, dx
\]

\[
\approx \sum_{i=0}^{p-1} \left[ \sum_{j=0}^{n-1} \cos(a_{ij}) \cdot h \right]; \quad h = (b - a) / p / n;
\]

\[
a_i = a + i \cdot n \cdot h; \quad a_{ij} = a_i + (j + 0.5) \cdot h
\]

\(p\) is number of partitions and \(n\) is increments per partition.
Program Example1

implicit none
integer n, p, i, j
real h, integral_sum, a, b, integral, pi, ai
pi = acos(-1.0) ! = 3.14159...
a = 0.0 ! lower limit of integration
b = pi/2. ! upper limit of integration
p = 4 ! number of partitions (processes)
n = 500 ! number of increments in each partition
h = (b-a)/p/n ! length of increment
ai = a + i*n*h
integral_sum = 0.0 ! Initialize solution to the integral
do i=0,p-1 ! Integral sum over all partitions
  integral_sum = integral_sum + integral(ai,h,n)
enddo
print *, 'The Integral =', integral_sum
stop
end
example1.f  continues . . .

real function integral(ai, h, n)
! This function computes the integral of the ith partition
  implicit none
  integer n, i, j     ! i is partition index; j is increment index
  real h, h2, aij, ai

  integral = 0.0             ! initialize integral
  h2 = h/2.           
  do j=0,n-1         ! sum over all "j" integrals
    aij = ai+ (j+0.5)*h     ! lower limit of integration of “j”
    integral = integral + cos(aij)*h    ! contribution due “j”
  enddo

  return
  end
Example 1 - Serial C code

```c
#include <math.h>
#include <stdio.h>

float integral(float a, int i, float h, int n);

void main() {
    int n, p, i, j, ierr;
    float h, integral_sum, a, b, pi, ai;
    pi = acos(-1.0);  /* = 3.14159... * 
    a = 0.;               /* lower limit of integration */
    b = pi/2.;           /* upper limit of integration */
    p = 4;                /* # of partitions */
    n = 500;            /* increments in each process */
    h = (b-a)/n/p;    /* length of increment */
    ai = a + i*n*h;    /* lower limit of int. for partition i */
    integral_sum = 0.0;
    for (i=0; i<p; i++) { /* integral sum over partitions */
        integral_sum += integral(ai,h,n);
    }
    printf("The Integral =%f\n", integral_sum);
}
```
example1.c continues . . .

```c
float integral(float ai, float h, int n) {
    int j;
    float aij, integ;
    integ = 0.0;               /* initialize integral */
    for (j=0; j<n; j++) {     /* sum over integrals in partition i*/
        aij = ai + (j+0.5)*h;  /* lower limit of integration of j*/
        integ += cos(aij)*h;    /* contribution due j */
    }
    return integ;
}
```
Example 1_1 - Parallel f77 code

Two main styles of programming: SPMD, MPMD. The following demonstrates SPMD, which is more frequently used than MPMD,

MPI functions used in this example:

- `MPI_Init, MPI_Comm_rank, MPI_Comm_size`
- `MPI_Send, MPI_Recv, MPI_Finalize`

PROGRAM Example1_1

  implicit none
  integer n, p, i, j, ierr, master, myid
  real h, integral_sum, a, b, integral, pi, ai
  include "mpif.h" ! pre-defined MPI constants, ...
  integer source, tag, status(MPI_STATUS_SIZE)
  real my_int

  data master/0/ ! 0 is the master processor responsible
  ! for collecting integral sums ...
! Starts MPI processes ...
   call MPI_Init(ierr)
! Get current process id
   call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
! Get number of processes from command line
   call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)

! executable statements before MPI_Init is not
! advisable; side effect implementation-dependent  (historical)

pi = acos(-1.0)   ! = 3.14159...
 a = 0.0           ! lower limit of integration
 b = pi/2.         ! upper limit of integration
 n = 500           ! number of increments in each process
 h = (b - a)/ p / n  ! (uniform) increment size
 tag = 123         ! set tag for job
 ai = a + myid*n*h  ! Lower limit of integration for partition myid
my_int = integral(ai, h, n)  ! compute local sum due
write(*,"(\'Process \',i2,\' has the partial integral of\',
& \ f10.6\")\"myid,my_int
    call MPI_Send(my_int, 1, MPI_REAL, master, tag,
& \ MPI_COMM_WORLD, ierr)  ! send my_int to master

if(myid .eq. master) then
    do source=0,p-1  ! loop on all procs to collect local sum (serial !)
        call MPI_Recv(my_int, 1, MPI_REAL, source, tag,
& \ MPI_COMM_WORLD, status, ierr)  ! not safe
        integral_sum = integral_sum + my_int
    enddo
    print *,\'The Integral =\', integral_sum
endif
    call MPI_Finalize(ierr)  ! let MPI finish up
end
Message Passing to Self

• It is valid to send/recv message to/from itself
• On IBM pSeries, env variable MP_EAGER_LIMIT may be used to control buffer memory size.
• Above example hangs if MP_EAGER_LIMIT set to 0
• Good trick to use to see if code is “safe”
• Not available with MPICH
Example 1_2 - Parallel C code

#include <mpi.h>
#include <math.h>
#include <stdio.h>
float integral(float a, int i, float h, int n); /* prototype */
void main(int argc, char *argv[]) {
    int n, p, i;
    float h, result, a, b, pi, my_int, ai;
    int myid, source, master, tag;
    MPI_Status status; /* MPI data type */
    MPI_Init(&argc, &argv); /* start MPI processes */
    MPI_Comm_rank(MPI_COMM_WORLD, &myid); /* current proc. id */
    MPI_Comm_size(MPI_COMM_WORLD, &p); /* # of processes */
... Parallel C code (continued)

\[
\begin{align*}
\pi &= \text{acos}(-1.0); \quad /* = 3.14159... */ \\
a &= 0.; \quad /* lower limit of integration */ \\
b &= \pi/2.; \quad /* upper limit of integration */ \\
n &= 500; \quad /* number of increment within each process */ \\
master &= 0; \\
/* define the process that computes the final result */ \\
tag &= 123; \quad /* set the tag to identify this particular job */ \\
h &= (b-a)/n/p; \quad /* length of increment */ \\
ai &= a + myid*n*h; \quad /* lower limit of int. for partition myid */ \\
my\_int &= \text{integral}(ai,h,n); \quad /* local sum due process myid */ \\
\text{printf}("Process %d has the partial integral of %f\n", myid,my\_int); 
\end{align*}
\]
if(myid == 0) {
    integral_sum = my_int;
    for (source=1; source<p; i++) {
        MPI_Recv(&my_int, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status); /* safe */
        integral_sum += my_int;
    }
    printf("The Integral =\%f\n", integral_sum);
} else {
    MPI_Send(&my_int, 1, MPI_FLOAT, master, tag, MPI_COMM_WORLD); /* send my_int to "master" */
}
MPI_Finalize(); /* let MPI finish up ... */
Essentials of Communication

- Sender must specify valid destination.
- Sender and receiver data type, tag, communicator must match.
- Receiver can receive from non-specific (but valid) source.
- Receiver returns extra (status) parameter to report info regarding message received.
- Sender specifies size of sendbuf; receiver specifies *upper bound* of recvbuf.
In the following slides, the compilation and job running procedures will be outlined for the computer systems maintained by SCV:

• Katana Cluster
• IBM pSeries 655 and 690
• IBM Bluegene/L
• Linux Cluster
How To Compile On Katana

On Katana Cluster:
• katana % mpif77 example.f (F77)
• katana % mpif90 example.f (F90)
• katana % mpicc example.c (C)
• katana % mpiCC example.C (C++)

• The above scripts should be used for MPI code compilation as they automatically include appropriate include files (–I) and library files (–L) for successful compilations.
• Above script names are generic. Compilers available are: Gnu and Portland Group.
• Two MPI implementations are available: MPICH and OpenMPI.
• See http://scv.bu.edu/computation/bladecenter/programming.html
How To Run Jobs On Katana

Interactive jobs:
• katana % mpirun -np 4 a.out

Batch jobs (via Sun GridEngine):
• katana % qsub myscript

See
http://scv.bu.edu/computation/bladecenter/runningjobs.html
Output of Example1_1

katana% mpirun -np 4 example1_1
Process 1 has the partial result of 0.324423
Process 2 has the partial result of 0.216773
Process 0 has the partial result of 0.382683
Process 3 has the partial result of 0.076120
The Integral = 1.000000

Processing out of order!
On AIX:

- Twister % mpxlf example.f    (F77)
- Twister % mpxlf90 example.f  (F90)
- Twister % mpxlf90 example.f90 (F90)
- Twister % mpcc example.c     (C)
- Twister % mpCC -D_MPI_CPP_BINDINGS example.C (C++)

See
http://scv.bu.edu/computation/pseries/programming.html

The above compiler scripts should be used for MPI code compilation as they automatically include appropriate include files (–I) and library files (–L) for successful compilations.
How To Run Jobs On pSeries

Interactive jobs:
- Twister % a.out -procs 4 or
- Twister % poe a.out -procs 4

LSF batch jobs:
- Twister % bsub -q queue-name "a.out -procs 4"

See
http://scv.bu.edu/computation/pseries/runningjobs.html
BGL consists of front-end and back-end. Compilation is performed on the FE but job is run on the BE. A cross compiler is required to achieve this:

- Lee % blrts_xlf example.f ...
- Lee % blrts_xlf90 example.f ...
- Lee % blrts_xlf90 example.f90 ...
- Lee % blrts_xlc example.c ...
- Lee % blrts_xlc -D_MPI_CPP_BINDINGS example.C ...

Need to link-in a handful of libraries, include files, etc., compilation is best handled with a makefile. For details, consult http://scv.bu.edu/computation/bluegene/programming.html

Many of the compiler switches are the same as for AIX. However, DO NOT use the -qarch=auto.
Interactive job: Not permitted

Loadleveler batch:

- Lee % llsubmit user-batch-script
- Lee % bglsub nprocs CWD EXE [“more MPI args”]

(A user script file called bglsub.$USER will also be generated. You can also use that along with llsubmit to run job)

Example:
Lee % bglsub 32 $PWD $PWD/example1_4 “< mystdin”

For details, see
http://scv.bu.edu/computation/bluegene/runningjobs.html
Example1_3 – Parallel Integration

MPI functions used for this example:

• MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Finalize
• MPI_Recv, MPI_Isend, MPI_Wait
• MPI_ANY_SOURCE, MPI_ANY_TAG

PROGRAM Example1_3
  implicit none
  integer n, p, i, j, proc, ierr, master, myid, tag, request
  real h, a, b, integral, pi, ai, my_int, integral_sum
  include "mpif.h"  ! This brings in pre-defined MPI constants, ...
  integer status(MPI_STATUS_SIZE)
  data master/0/

**Example1_3 (continued)**

```c
**Starts MPI processes ...**

```c
    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
```

```c
    pi = acos(-1.0)    !  = 3.14159...
    a = 0.0           ! lower limit of integration
    b = pi/2.         ! upper limit of integration
    n = 500           ! number of increment within each process
    dest = master     ! define process that computes the final result
    tag = 123         ! set the tag to identify this particular job
    h = (b-a)/n/p     ! length of increment

    ai = a + myid*n*h;  ! starting location of partition “myid”
    my_int = integral(ai,h,n)  ! Integral of process myid
    write(*,*)'myid=',myid,', my_int=',my_int
```
Example1_3 (continued)

if (myid .eq. master) then
   integral_sum = my_int
   do k=1,p-1
      call MPI_Recv(my_int, 1, MPI_REAL,
                       MPI_ANY_SOURCE, MPI_ANY_TAG,
                       MPI_COMM_WORLD, status, ierr)
      integral_sum = integral_sum + my_int
   enddo
else
   call MPI_Isend(my_int, 1, MPI_REAL, dest, tag,
                       MPI_COMM_WORLD, req, ierr)
   C**more computation here . . .
   call MPI_Wait(req, status, ierr)
endif
C**results from all procs have been collected and summed ...
if (myid .eq. 0) write(*,*)'The Integral =',integral_sum
call MPI_Finalize(ierr)
stop
end
1. Write a C or FORTRAN program to print the statement "Hello, I am process X of Y processes" where X is the current process while Y is the number of processes for job.

2. Write a C or FORTRAN program to do the following:
   1. On process 0, send a message "Hello, I am process 0" to other processes.
   2. On all other processes, print the process's ID, the message it receives and where the message came from.

Makefile and programs are in /net/katana/scratch/kadin
Example 1_4 Parallel Integration

MPI functions and constants used for this example:

- `MPI_Init`, `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Finalize`
- `MPI_Bcast`, `MPI_Reduce`, `MPI_SUM`

```fortran
PROGRAM Example1_4
  implicit none
  integer n, p, i, j, ierr, master
  real h, integral_sum, a, b, integral, pi, ai
  include "mpif.h"  ! This brings in pre-defined MPI constants, ...
  integer myid, source, dest, tag, status(MPI_STATUS_SIZE)
  real my_int

  data master/0/
```

June 2010
**Example 1_4 (continued)**

```fortran

**C** Starts MPI processes ...

```fortran
            call MPI_Init(ierr)
            call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
            call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)

pi = acos(-1.0)   ! = 3.14159...

a = 0.0         ! lower limit of integration
b = pi/2.     ! upper limit of integration
h = (b-a)/n/p    ! length of increment
dest = 0     ! define the process that computes the final result
tag = 123    ! set the tag to identify this particular job
if(myid .eq. master) then
    print *, 'The requested number of processors = ', p
    print *, 'enter number of increments within each process'
    read(*,*) n
endif
```
Example1_4 (continued)

```fortran
! Broadcast "n" to all processes
    call MPI_Bcast(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
    ai = a + myid*h*n
    my_int = integral(ai,h,n)
    write(*,'("Process ",i2," has the partial sum of ",f10.6")')
    & myid, my_int

    call MPI_Reduce(my_int, integral_sum, 1, MPI_REAL, MPI_SUM,
    & dest, MPI_COMM_WORLD, ierr) ! Compute integral sum

    if(myid.eq.master) then
        print *,"The Integral Sum = ", integral_sum
    endif

    call MPI_Finalize(ierr) ! let MPI finish up ...
    stop
end
```
Example1_5 Parallel Integration

New MPI functions and constants used for this example:

• `MPI_Init`, `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Finalize`
• `MPI_Pack`, `MPI_Unpack`
• `MPI_FLOAT_INT`, `MPI_MINLOC`, `MPI_MAXLOC`, `MPI_PACKED`

```c
#include <mpi.h>
#include <math.h>
#include <stdio.h>
float fct(float x) { return cos(x); } /* Prototype */
float integral(float ai, float h, int n);
int main(int argc, char* argv[]) {
```

int n, p;
float h, integral_sum, a, b, pi, ai;
int myid, dest, m, index, minid, maxid, Nbytes=1000, master=0;
char line[10], scratch[Nbytes];
struct {
    float val;
    int loc; } local_sum, min_sum, max_sum;

MPI_Init(&argc,&argv); /* starts MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myid); /* process id */
MPI_Comm_size(MPI_COMM_WORLD, &p); /* num of procs*/
pi = acos(-1.0); /* = 3.14159... */
dest = 0; /* define the process to compute final result */
comm = MPI_COMM_WORLD;
if(myid == master) {
    printf("The requested number of processors = %d\n", p);
    printf("enter number of increments within each process\n");
    (void) fgets(line, sizeof(line), stdin);
    (void) sscanf(line, "%d", &n);
    printf("enter a & m\n");
    printf(" a = lower limit of integration\n");
    printf(" b = upper limit of integration\n");
    printf("   = m * pi/2\n");
    (void) fgets(line, sizeof(line), stdin);
    (void) sscanf(line, "%d %d", &a, &m);
    b = m * pi / 2.;
}
If (myid == master) {
    /* to be efficient, pack all things into a buffer for broadcast */
    index = 0;
    MPI_Pack(&n, 1, MPI_INT, scratch, Nbytes, &index, comm);
    MPI_Pack(&a, 1, MPI_FLOAT, scratch, Nbytes, &index, comm);
    MPI_Pack(&b, 1, MPI_FLOAT, scratch, Nbytes, &index, comm);
    MPI_Bcast(scratch, Nbytes, MPI_PACKED, master, comm);
} else {
    MPI_Bcast(scratch, Nbytes, MPI_PACKED, master, comm);
    /* things received have been packed, unpack into expected locations */
    index = 0;
    MPI_Unpack(scratch, Nbytes, &index, &n, 1, MPI_INT, comm);
    MPI_Unpack(scratch, Nbytes, &index, &a, 1, MPI_FLOAT, comm);
    MPI_Unpack(scratch, Nbytes, &index, &b, 1, MPI_FLOAT, comm);
}
Example1_5 (cont’d)

h = (b-a)/n/p;  /* length of increment */
ai = a + myid*h*n;
local_sum.val = integral(ai,h,n);
local_sum.loc = myid;

printf("Process %d has the partial sum of %f\n", myid, local_sum.val);

/* data reduction with MPI_SUM */
   MPI_Reduce(&local_sum.val, &integral_sum, 1, MPI_FLOAT,
               MPI_SUM, dest, comm);

/* data reduction with MPI_MINLOC */
   MPI_Reduce(&local_sum, &min_sum, 1, MPI_FLOAT_INT,
               MPI_MINLOC, dest, comm);

/* data reduction with MPI_MAXLOC */
   MPI_Reduce(&local_sum, &max_sum, 1, MPI_FLOAT_INT,
               MPI_MAXLOC, dest, comm);
Example1_5 (cont’d)

if(myid == master) {
    printf("The Integral = %f\n", integral_sum);
    maxid = max_sum.loc;
    printf("Proc %d has largest integrated value of %f\n", maxid, max_sum.val);
    minid = min_sum.loc;
    printf("Proc %d has smallest integrated value of %f\n", minid, min_sum.val);
}

MPI_Finalize(); /* let MPI finish up ... */
C++ example

#include <mpi.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    cout << "Hello world! I am " << rank << " of " << size << endl;
    MPI::Finalize();
    return 0; }

Twister % mpCC -DHAVE_MPI_CXX -o hello hello.C
Twister % hello -procs 4
\[
S = \frac{T_1}{T_N} \cdot \left( f + \frac{1 - f}{N} \right) T_1 \cdot \frac{1}{f} \quad \text{as} \quad N \to \infty
\]

Amdahl’s Law above states that a code with its parallelizable component comprising 90% of total computation time can at best achieve a 10X speedup with lots of workers. A code that is 50% parallelizable speeds up two-fold with lots of workers.

The parallel efficiency is \( E = S / N \)

Program that scales linearly (\( S = N \)) has parallel efficiency 1. A task-parallel program is usually more efficient than a data-parallel program. Data-parallel codes can sometimes achieve super-linear behavior due to efficient cache usage per worker.
Speedup Ratio & Parallel Efficiency

Graph showing speedup ratio and parallel efficiency for DGETRF.
Collective Communications

Pass data among a group of processors.
### Collective Functions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td><strong>MPI_Bcast</strong></td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td><strong>MPI_Gather</strong></td>
<td>a,b,c,d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td><strong>MPI_Allgather</strong></td>
<td>a,b,c,d</td>
<td>a,b,c,d</td>
<td>a,b,c,d</td>
<td>a,b,c,d</td>
</tr>
<tr>
<td>a,b,c,d</td>
<td></td>
<td></td>
<td></td>
<td><strong>MPI_Scatter</strong></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>a,b,c,d</td>
<td>e,f,g,h</td>
<td>i,j,k,l</td>
<td>m,n,o,p</td>
<td><strong>MPI_Alltoall</strong></td>
<td>a,e,i,m</td>
<td>b,f,j,n</td>
<td>c,g,k,o</td>
<td>d,h,l,p</td>
</tr>
</tbody>
</table>

- **This example uses 4 processes**
- **Rank 1 is, arbitrarily, designated data gather/scatter process**
- **a, b, c, d are scalars or arrays of any data type**
- **Data are gathered/scattered according to rank order**
program collectives_example
  implicit none
  integer p, ierr, i, myid, root
  include "mpif.h" ! This brings in pre-defined MPI constants, ... 
  character*1 x(0:3), y(0:3), alphabets(0:15)
  data alphabets/'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p'/
                  'm','n','o','p'/
  data root/1/ ! process 1 is the data sender/receiver
  ! Starts MPI processes ...
  call MPI_Init(ierr) ! starts MPI
  call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr) ! current pid
  call MPI_Comm_size(MPI_COMM_WORLD, p, ierr) ! # of procs

if (myid .eq. 0) then
    write(*,*)
    write(*,*)'* This program demonstrates the use of collective',
&                     ' MPI functions'
    write(*,*)'* Four processors are to be used for the demo'
    write(*,*)'* Process 1 (of 0,1,2,3) is the designated root'
    write(*,*)
    write(*,*)
    write(*,*)' Function Proc Sendbuf Recvbuf'
    write(*,*)' -------- ---- ------- --------'
endif
Gather Operation

\[ c^{\ast\ast} \text{Performs a gather operation} \]

\[ x(0) = \text{alphabets}(\text{myid}) \]

\[ \text{do } i=0,p-1 \]
\[ \quad y(i) = ' ' \]
\[ \text{enddo} \]

\[ \text{call MPI\_Gather}(x,1,\text{MPI\_CHARACTER}, ! \text{Send-buf, count, type,} \]
\[ & \quad y,1,\text{MPI\_CHARACTER}, ! \text{Recv-buf, count?, type?}, \]
\[ & \quad \text{root,} ! \text{Data destination} \]
\[ & \quad \text{MPI\_COMM\_WORLD, ierr) ! Comm, flag} \]

\[ \text{write(*,}''(''\text{MPI\_Gather:}'',t20,i2,(3x,a1),t40,4(3x,a1))'')\text{myid, }x(0),y \]

\[ \text{Recv-buf according to rank order} \]

\[ \text{alphabets}(0) = 'a' \]
\[ \text{alphabets}(1) = 'b' \]
\[ \ldots \]
\[ \text{alphabets}(14) = 'o' \]
\[ \text{alphabets}(15) = 'p' \]
c**Performs an all-gather operation
x(0) = alphabets(myid)
do i=0,p-1
    y(i) = ' '
enddo
call MPI_Allgather(x,1,MPI_CHARACTER, ! send buf,count,type
&        y,1,MPI_CHARACTER,       ! recv buf,count,type
&        MPI_COMM_WORLD,ierr) ! comm,flag
write(*,"('MPI_Allgather:',t20,i2,(3x,a1),t40,4(3x,a1))")myid,x(0),y
c**Perform a scatter operation
   if (myid .eq. root) then
      do i=0, p-1
         x(i) = alphabets(i)
         y(i) = ' ' 
      enddo
   else
      do i=0, p-1
         x(i) = ' ' 
         y(i) = ' ' 
      enddo
   endif
   call MPI_scatter(x,1,MPI_CHARACTER, ! Send-buf,count,type
                   & y,1,MPI_CHARACTER,    ! Recv-buf,count,type
                   & root,                 ! data origin
                   & MPI_COMM_WORLD,ierr) ! comm,flag
write(*,"('MPI_scatter:',t20,i2,4(3x,a1),t40,4(3x,a1))")myid,x,y
Alltoall Operation

c**Perform an all-to-all operation
     do i=0,p-1
         x(i) = alphabets(i+myid*p)
         y(i) = ' '
     enddo
     call MPI_Alltoall(x,1,MPI_CHARACTER,    ! send buf,count,type
                      &                      y,1,MPI_CHARACTER,    ! recv buf,count,type
                      &                      MPI_COMM_WORLD,ierr)   ! comm,flag
     write(*,"('MPI_Alltoall:',t20,i2,4(3x,a1),t40,4(3x,a1))")myid,x,y

c** Performs a broadcast operation
  do i=0, p-1
    x(i) = ' '
    y(i) = ' '
  enddo
  if(myid .eq. root) then
    x(0) = 'b'
    y(0) = 'b'
  endif
  call MPI_Bcast(y,1,MPI_CHARACTER, root,MPI_COMM_WORLD,ierr) ! buf,count,type
  write(*,"('MPI_Bcast:',t20,i2,4(3x,a1),t40,4(3x,a1))")myid,x,y
  call MPI_Finalize(ierr) ! let MPI finish up ...
end
#include <mpi.h>
#include <math.h>
#include <stdio.h>

float fct(float x)
{
    return cos(x);
}

/* Prototype */
float integral(float a, int i, float h, int n);

int main(int argc, char* argv[])
{
    int n, p, myid, i;
    float h, integral_sum, a, b, pi, my_int;
    float buf[50], tmp;

Example 1.6 (cont’d)

```c
MPI_Init(&argc,&argv);                 /* starts MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* current proc id */
MPI_Comm_size(MPI_COMM_WORLD, &p);        /* num of procs */

pi = acos(-1.0);  /* = 3.14159... */
a = 0.;                /* lower limit of integration */
b = pi*1./2.;       /* upper limit of integration */
n = 500;            /* number of increment within each process */
h = (b-a)/n/p;    /* length of increment */

my_int = integral(a,myid,h,n);

printf("Process %d has the partial sum of %f\n", myid,my_int);

MPI_Gather(&my_int, 1, MPI_FLOAT, buf, 1, MPI_FLOAT, 0,
MPI_COMM_WORLD);
```
Example 1.6 (cont’d)

MPI_Scatter(buf, 1, MPI_FLOAT, &tmp, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
printf("Result sent back from buf = %f\n", tmp);

if(myid == 0) {
    integral_sum = 0.0;
    for (i=0; i<p; i++) {
        integral_sum += buf[i];
    }
    printf("The Integral =%f\n", integral_sum);
}

MPI_Finalize(); /* let MPI finish up ... */
This example demonstrates dynamic memory allocation and parallel timer.

Program dma_example
implicit none
include "mpif.h"
integer, parameter :: real_kind = selected_real_kind(8,30)
real(real_kind), dimension(55) :: sdata
real(real_kind), dimension(:), allocatable :: rdata
real(real_kind) :: start_time, end_time
integer :: p, i, count, myid, n, status(MPI_STATUS_SIZE), ierr

!* Starts MPI processes ...*
call MPI_Init(ierr)  !* starts MPI*
call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)  ! myid
 call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)  ! Num. proc
start_time = MPI_Wtime() ! start timer, measured in seconds
if (myid == 0) then
  sdata(1:50) = (/ (i, i=1,50) /)
call MPI_Send(sdata, 50, MPI_DOUBLE_PRECISION, 1, 123, &
          MPI_COMM_WORLD, ierr)
else
  call MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
call MPI_Get_count(status, MPI_DOUBLE_PRECISION, count, ierr)
allocation(rdata(count))
call MPI_Recv(rdata, count, MPI_DOUBLE_PRECISION, 0, &
          MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
write(*, '(5f10.2)') rdata(1:count:10)
endif
end_time = MPI_Wtime() ! stop timer
if (myid .eq. 1) then
    WRITE(*,"(‘ Total cpu time =’,f10.5,’ x ‘,i3)"") end_time -
    start_time,p
endif

call MPI_Finalize(ierr)  !* let MPI finish up ...

call MPI_Finalize(ierr)  !* let MPI finish up ...
end program dma_example
#include <mpi.h>
#include <math.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
    double sdata[55], *rdata, start_time, end_time;
    int p, i, count, myid, n;
    MPI_Status status;

    /* Starts MPI processes ... */
    MPI_Init(&argc, &argv);  /* starts MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &myid); /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD, &p); /* get number of processes */
    ...
start_time = MPI_Wtime(); /* starts timer */
if (myid == 0) {
    for(i=0;i<50;++i) { sdata[i]=(double)i; }

    MPI_Send(sdata,50,MPI_DOUBLE,1,123,MPI_COMM_WORLD);
} else {
    MPI_Probe(0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
    MPI_Get_count(&status,MPI_DOUBLE,&count);
    MPI_Type_size(MPI_DOUBLE,&n); /* sizeof */
    rdata= (double*) calloc(count,n);
    MPI_Recv(rdata,count,MPI_DOUBLE,0,MPI_ANY_TAG,
              MPI_COMM_WORLD, &status);
    for(i=0;i<count;i+=10) {
        printf("rdata element %d is %f\n",i,rdata[i]);
    }
}
end_time = MPI_Wtime(); /* ends timer */
if (myid == 1) {
    printf("Total time is %f x %d\n", end_time-start_time, p);
}

MPI_Finalize(); /* let MPI finish up ... */
Useful SCV Info

Please help us do better in the future by participating in a quick survey:
http://scv.bu.edu/survey/sumtut_survey.html

• **SCV home page** http://scv.bu.edu/

• Resource Applications
  – [http://scv.bu.edu/accounts/](http://scv.bu.edu/accounts/)

• Help
  – FAQs
  – Web-based tutorials (MPI, OpenMP, MATLAB, Graphics tools)
  – HPC consultations by appointment
    - Kadin Tseng (kadin@bu.edu)
    - Doug Sondak (sondak@bu.edu)
  – help@twister.bu.edu, help@cootie.bu.edu