
Parallel Programming

Distributed-Memory Programming with MPI (Part II)

Professor Yi-Ping You (游逸平)
Department of Computer Science
<http://www.cs.nctu.edu.tw/~ypyou/>



Acknowledgement

- This set of slides is mainly developed by Prof. Thomas Sterling and Dr. Steve Brandt from Louisiana State University



Review of Basic MPI Calls

- In review, the 6 main MPI calls:
 - MPI_Init
 - MPI_Finalize
 - MPI_Comm_size
 - MPI_Comm_rank
 - MPI_Send
 - MPI_Recv
- Include MPI Header file
 - #include “mpi.h”
- Basic MPI Datatypes
 - MPI_INT, MPI_FLOAT,

Collective Calls

- A communication pattern that encompasses all processes within a communicator is known as **collective communication**
- MPI has several collective communication calls, the most frequently used are:
 - Synchronization
 - Barrier
 - Communication
 - Broadcast
 - Gather & Scatter
 - All Gather
 - Reduction
 - Reduce
 - AllReduce

Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives
- Derived Datatypes: Introduction
- Derived Datatypes: Contiguous
- Derived Datatypes: Vector
- Derived Datatypes: Indexed
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- MPI Profiling calls
- Additional Topics
- Summary

Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives
- Derived Datatypes: Introduction
- Derived Datatypes: Contiguous
- Derived Datatypes: Vector
- Derived Datatypes: Indexed
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- MPI Profiling calls
- Additional Topics
- Summary

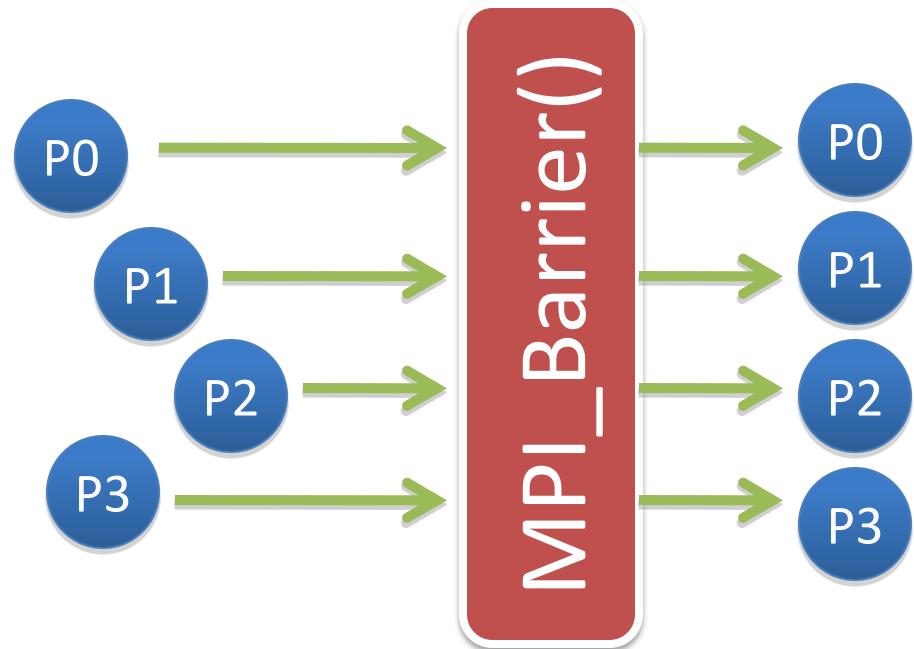
MPI Collective Calls : Barrier

Function: **MPI_Barrier()**

```
int MPI_Barrier (  
    MPI_Comm comm )
```

Description:

Creates barrier synchronization in a communicator group *comm*. Each process, when reaching the MPI_Barrier call, blocks until all the processes in the group reach the same MPI_Barrier call.



Example: MPI_Barrier()

```
#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[]){
    int      rank, size, len;
    char     name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc, &argv);
MPI_Barrier(MPI_COMM_WORLD);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(name, &len);
MPI_Barrier(MPI_COMM_WORLD);

    printf ("Hello world! Process %d of %d on %s\n", rank, size, name);
    MPI_Finalize();
    return 0;
}
```

```
[cdekkate@celeritas collective]$ mpirun -np 8 barrier
Hello world! Process 0 of 8 on celeritas.cct.lsu.edu
Writing logfile....
Finished writing logfile.
Hello world! Process 4 of 8 on compute-0-3.local
Hello world! Process 1 of 8 on compute-0-0.local
Hello world! Process 3 of 8 on compute-0-2.local
Hello world! Process 6 of 8 on compute-0-5.local
Hello world! Process 7 of 8 on compute-0-6.local
Hello world! Process 5 of 8 on compute-0-4.local
Hello world! Process 2 of 8 on compute-0-1.local
[cdekkate@celeritas collective]$
```

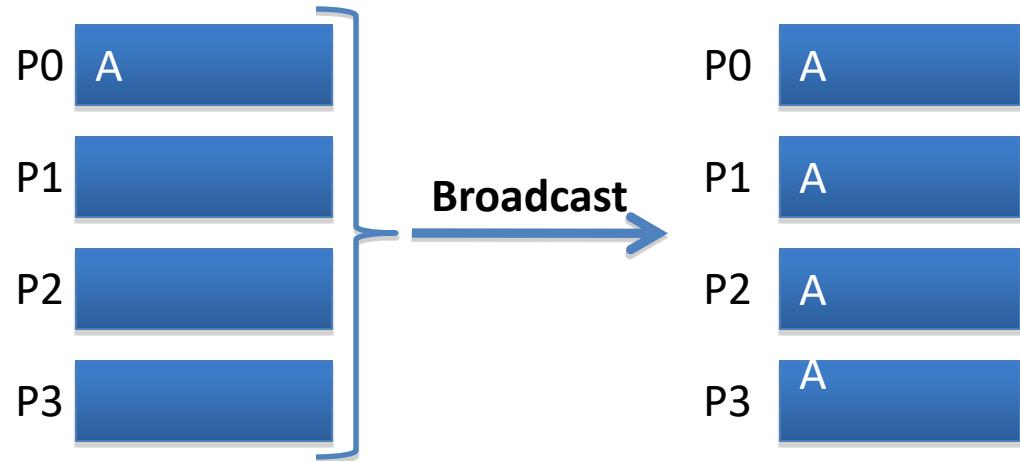
Topics

- MPI Collective Calls: Synchronization Primitives
- **MPI Collective Calls: Communication Primitives**
- MPI Collective Calls: Reduction Primitives
- Derived Datatypes: Introduction
- Derived Datatypes: Contiguous
- Derived Datatypes: Vector
- Derived Datatypes: Indexed
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- MPI Profiling calls
- Additional Topics
- Summary

MPI Collective Calls : Broadcast

Function: **MPI_Bcast()**

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



Description:

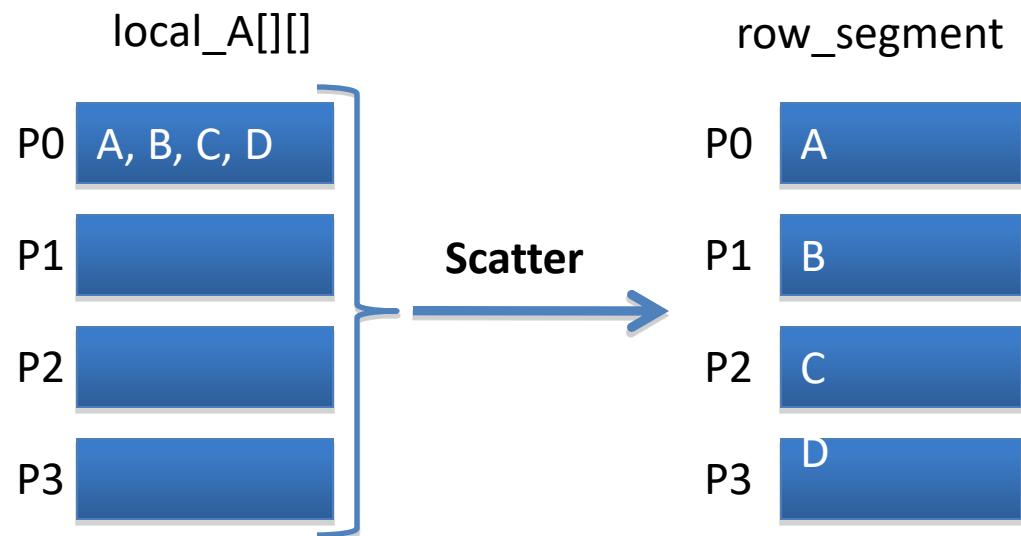
A collective communication call where a single process sends the same data contained in the *message* to every process in the communicator. By default a tree like algorithm is used to broadcast the message to a block of processors, a linear algorithm is then used to broadcast the message from the first process in a block to all other processes. All the processes invoke the *MPI_Bcast* call with the same arguments for *root* and *comm*.

```
float      endpoint[2];
...
MPI_Bcast(endpoint, 2, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

MPI Collective Calls : Scatter

Function: **MPI_Scatter()**

```
int MPI_Scatter (
    void          *sendbuf,
    int            send_count,
    MPI_Datatype  send_type,
    void          *recvbuf,
    int            recv_count,
    MPI_Datatype  recv_type,
    int            root,
    MPI_Comm      comm)
```



Description: http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Scatter.html

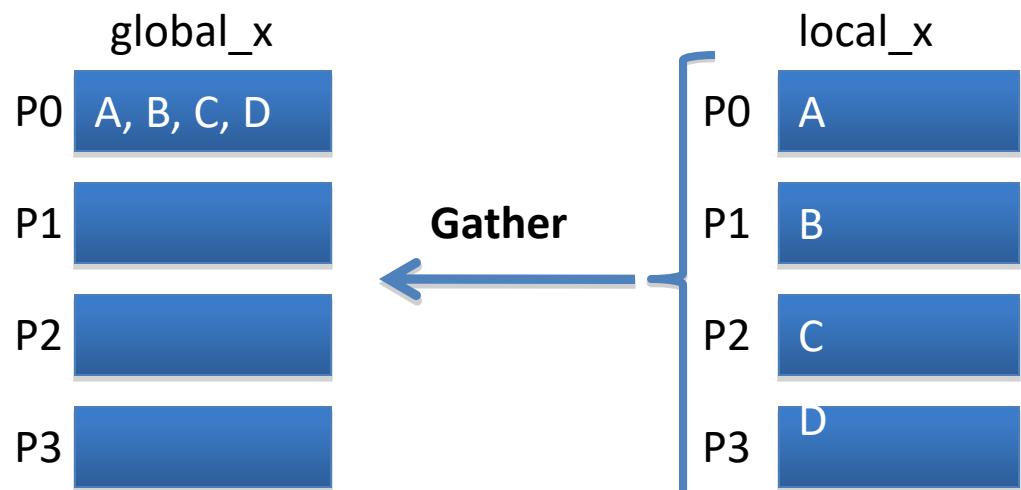
MPI_Scatter splits the data referenced by the *sendbuf* on the process with rank *root* into *p* segments each of which consists of *send_count* elements of type *send_type*. The first segment is sent to process0 and the second segment to process1. The send arguments are significant on the process with rank *root*.

```
...
MPI_Scatter(&(local_A[0][0]), n/p, MPI_FLOAT, row_segment, n/p, MPI_FLOAT, 0,MPI_COMM_WORLD);
...
```

MPI Collective Calls : Gather

Function: **MPI_Gather()**

```
int MPI_Gather (
    void             *sendbuf,
    int              send_count,
    MPI_Datatype    sendtype,
    void             *recvbuf,
    int              recvcount,
    MPI_Datatype    recvtype,
    int              root,
    MPI_Comm         comm )
```



Description: http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Gather.html

MPI_Gather collects the data referenced by *sendbuf* from each process in the communicator *comm*, and stores the data in process rank order on the process with rank *root* in the location referenced by *recvbuf*. The *recv* parameters are only significant.

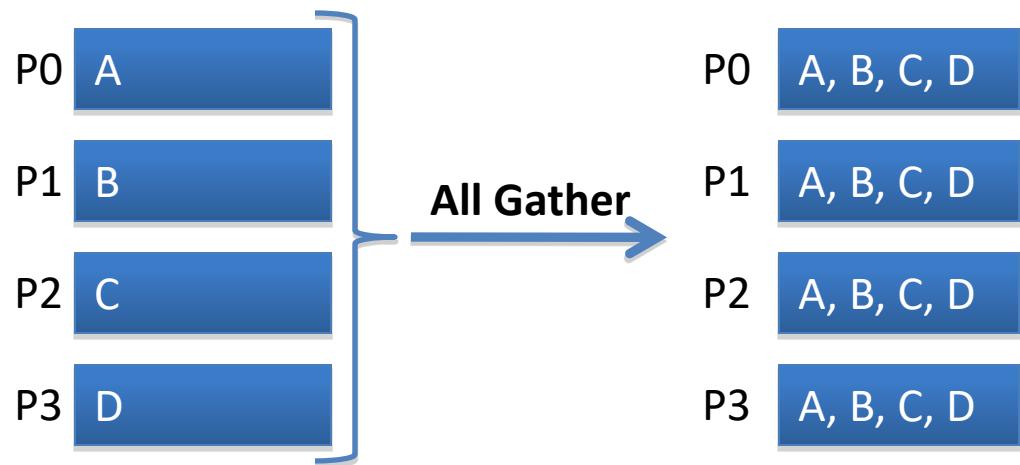
```
...
MPI_Gather(local_x, n/p, MPI_FLOAT, global_x, n/p, MPI_FLOAT, 0, MPI_COMM_WORLD);
...
```

MPI Collective Calls : All Gather

Function: **MPI_Allgather()**

```
int MPI_Allgather (
```

```
    void *sendbuf,  
    int send_count,  
    MPI_Datatype sendtype,  
    void *recvbuf,  
    int recvcount,  
    MPI_Datatype recvtype,  
    MPI_Comm comm )
```



http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Allgather.html

Description:

MPI_Allgather gathers the content from the send buffer (*sendbuf*) on each process. The effect of this call is similar to executing MPI_Gather() *p* times with a different process acting as the root.

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

CAN BE REPLACED WITH :

```
MPI_Allgather(local_x, local_n, MPI_FLOAT, global_x, local_n, MPI_FLOAT, MPI_COMM_WORLD);
```

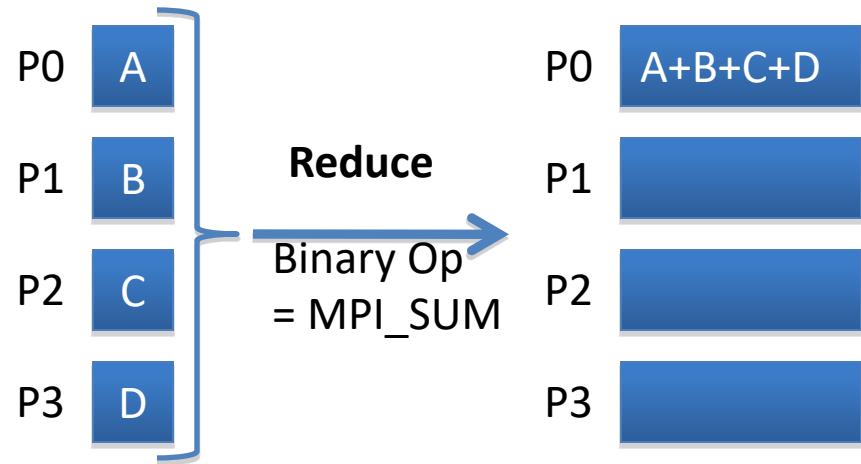
Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- **MPI Collective Calls: Reduction Primitives**
- Derived Datatypes: Introduction
- Derived Datatypes: Contiguous
- Derived Datatypes: Vector
- Derived Datatypes: Indexed
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- MPI Profiling calls
- Additional Topics
- Summary

MPI Collective Calls : Reduce

Function: **MPI_Reduce()**

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



http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Reduce.html

Description:

A collective communication call where all the processes in a communicator contribute data that is combined using binary operations (MPI_Op) such as addition, max, min, logical, and, etc. MPI_Reduce combines the operands stored in the memory referenced by *operand* using the operation *operator* and stores the result in **result*. MPI_Reduce is called by all the processes in the communicator *comm* and for each of the processes *count*, *datatype* *operator* and *root* remain the same.

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

MPI Binary Operations

- MPI binary operators are used in the MPI_Reduce function call as one of the parameters. MPI_Reduce performs a global reduction operation (dictated by the MPI binary operator parameter) on the supplied operands.
- Some of the common MPI Binary Operators used are :

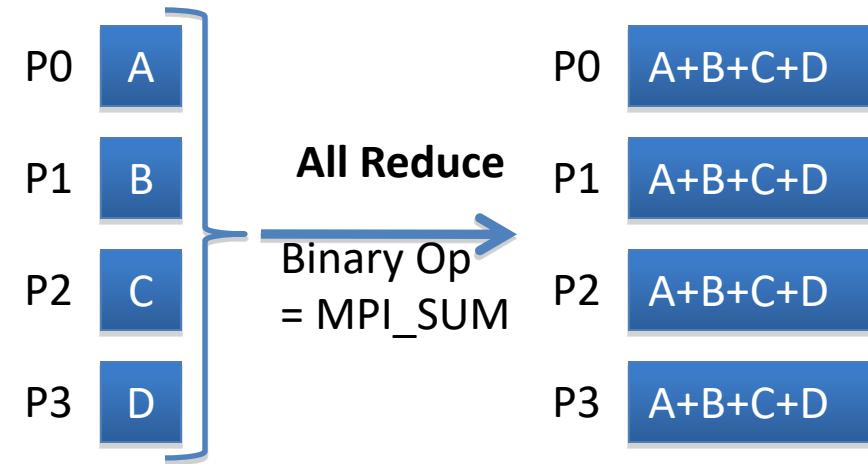
Operation Name Meaning	
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_MAXLOC	Maximum and location of max.
MPI_MINLOC	Maximum and location of min.
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical And
MPI_BAND	Bitwise And
MPI_LOR	Logical Or
MPI_BOR	Bitwise Or
MPI_LXOR	Logical XOR
MPI_BXOR	Bitwise XOR

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

MPI Collective Calls : All Reduce

Function: `MPI_Allreduce()`

```
int MPI_Allreduce (
    void          *sendbuf,
    void          *recvbuf,
    int           count,
    MPI_Datatype datatype,
    MPI_Op        op,
    MPI_Comm      comm )
```



Description:

`MPI_Allreduce` is used exactly like `MPI_Reduce`, except that the result of the reduction is returned on all processes, as a result there is no *root* parameter.

```
...
MPI_Allreduce(&integral, &integral, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
...
```

Parallel Trapezoidal Rule

Send, Recv

```
#include <stdio.h>
#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' 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);
```

Parallel Trapezoidal Rule

Send, Recv

```
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);
}
MPI_Finalize();
} /* main */
```

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco Ch 4

Parallel Trapezoidal Rule

Send, Recv

```
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 */

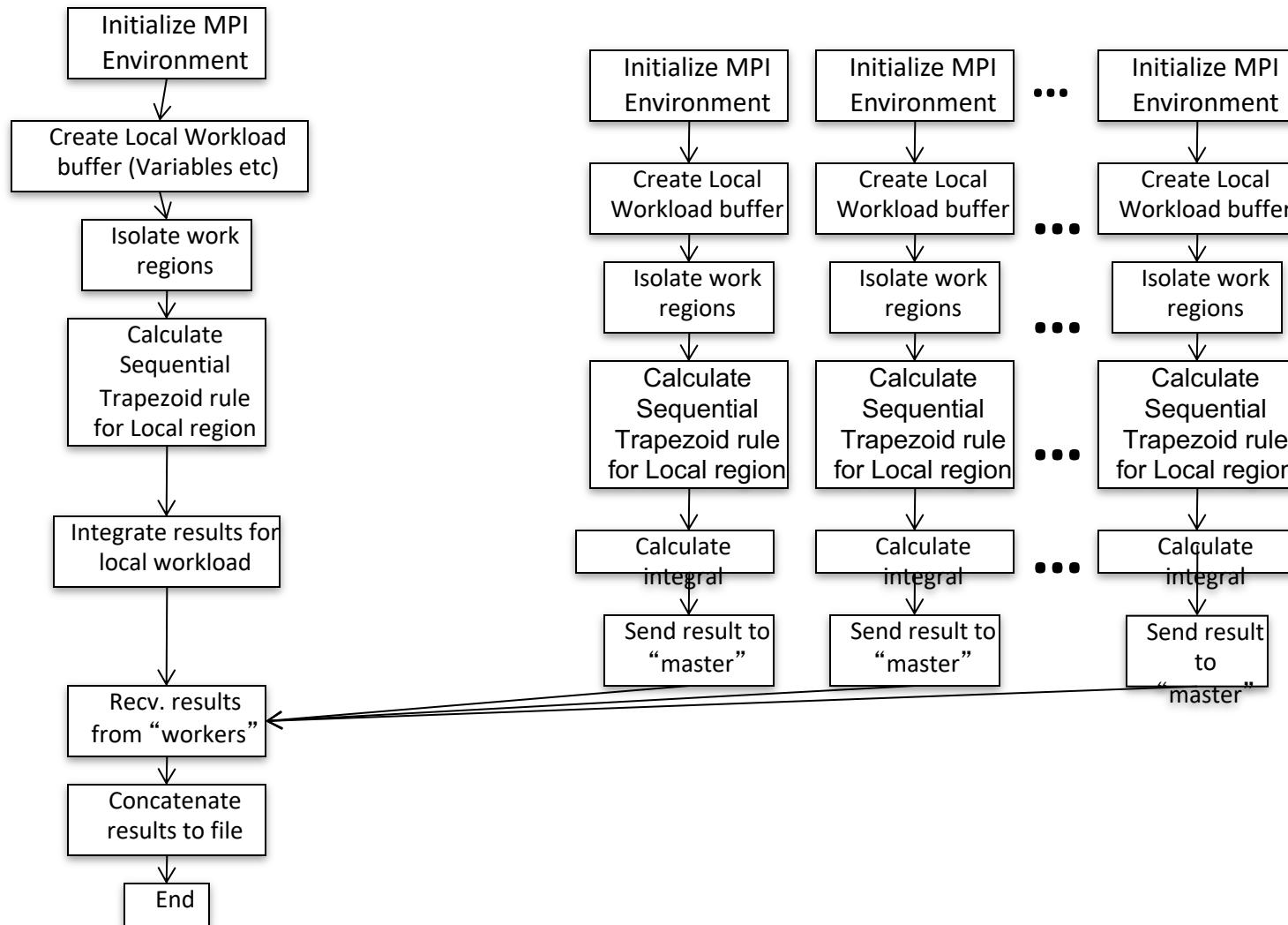
    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
} /* Trap */
float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
} /* f */
```

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco Ch 4

Flowchart for Parallel Trapezoidal Rule

MASTER

WORKERS



Trapezoidal Rule : with MPI_Bcast, MPI_Reduce

```
#include <stdio.h>
#include <stdlib.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   endpoint[2]; /* Left and right      */
    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 */

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

if (argc != 3) {
    if (my_rank==0)
        printf("Usage: mpirun -np <numprocs> trapezoid
<left> <right>\n");
    MPI_Finalize();
    exit(0);
}

if (my_rank==0) {
    endpoint[0] = atof(argv[1]); /* left endpoint */
    endpoint[1] = atof(argv[2]); /* right endpoint */
}
```

**MPI_Bcast(endpoint, 2, MPI_FLOAT,
0, MPI_COMM_WORLD);**

Trapezoidal Rule : with MPI_Bcast, MPI_Reduce

```
h = (endpoint[1]-endpoint[0])/n; /* h is the same for all processes
*/
local_n = n/p; /* so is the number of trapezoids */
if (my_rank == 0) printf("a=%f, b=%f, Local number of
trapezoids=%d\n", endpoint[0], endpoint[1], local_n );

local_a = endpoint[0] + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);

MPI_Reduce(&integral, &total, 1,
MPI_FLOAT, MPI_SUM, 0,
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",
        endpoint[0], endpoint[1], total);
}

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

Trapezoidal Rule : with MPI_Bcast, MPI_Reduce

```
#!/bin/bash
#PBS -N name
#PBS -l walltime=120:00:00,nodes=2:ppn=4
cd /home/lwu00/Demos/l9/trapBcast
pwd
date
PROCS=`wc -l < $PBS_NODEFILE`
mpdboot --file=$PBS_NODEFILE
/usr/lib64/mpich2/bin/mpiexec -n $PROCS ./trapBcast 2 25 >>out.txt
mpdallexit
date
```

```
[lsu00@master trapBcast]$ qsub try.pbs
518.master.arete.cct.lsu.edu
[lsu00@master trapBcast]$ cat out.txt
a=2.000000, b=25.000000, Local number of trapezoids=128
With n = 1024 trapezoids, our estimate
of the integral from 2.000000 to 25.000000 = 5205.667969
```

Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives
- **Derived Datatypes: Introduction**
- Derived Datatypes: Contiguous
- Derived Datatypes: Vector
- Derived Datatypes: Indexed
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- MPI Profiling calls
- Additional Topics
- Summary

Consolidating Data

- In virtually all distributed-memory systems, communication can be much more expensive than local computation
- We would expect the following pair of **for** loops to be much slower than the single send/receive pair

```
double x[1000];
. . .
if (my_rank == 0)
    for (i = 0; i < 1000; i++)
        MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
else /* my_rank == 1 */
    for (i = 0; i < 1000; i++)
        MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);

if (my_rank == 0)
    MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);
else /* my_rank == 1 */
    MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

Consolidating Data (Cont'd)

- MPI provides three basic approaches to consolidating data that might otherwise require multiple messages:
 - the count argument to the various communication functions,
 - derived datatypes, and
 - MPI_Pack/Unpack

Constructing Datatypes

- Creating data structures in C:

```
typedef struct {  
    . . .  
} STRUCT_NAME;
```

- For example : In the numerical integration by trapezoidal rule we could create a data structure for storing the attributes of the problem as follows:

```
typedef struct {  
    float a,  
    float b,  
    int n;  
} DATA_INTEGRAL;  
.  
. . .  
DATA_INTEGRAL intg_data;
```

- What would happen when you use:

```
MPI_Bcast( &intg_data, 1, DATA_INTEGRAL, 0, MPI_COMM_WORLD);
```

ERROR!!!
Intg_data is of the type
DATA_INTEGRAL
NOT an MPI_Datatype

Constructing MPI Datatypes

- MPI allows users to define derived MPI datatypes, using basic datatypes that build during execution time
- These derived data types can be used in the MPI communication calls, instead of the basic predefined datatypes.
- A sending process can pack noncontiguous data into contiguous buffer and send the buffered data to a receiving process that can unpack the contiguous buffer and store the data to noncontiguous location.
- A derived datatype is an opaque object that specifies :
 - A sequence of primitive datatypes
 - A sequence of integer (byte) displacements
- MPI has several functions for constructing derived datatypes :
 - Contiguous
 - Vector
 - Indexed
 - Struct

MPI : Basic Data Types (Review)

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

You can also define your own (derived datatypes), such as an array of ints of size 100, or more complex examples, such as a struct or an array of structs

Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives
- Derived Datatypes: Introduction
- **Derived Datatypes: Contiguous**
- Derived Datatypes: Vector
- Derived Datatypes: Indexed
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- MPI Profiling calls
- Additional Topics
- Summary

Derived Datatypes : Contiguous

Function: **MPI_Type_contiguous()**

```
int MPI_Type_contiguous(  
    int count,  
    MPI_Datatype old_type,  
    MPI_Datatype *new_type)
```

Description:

This is the simplest constructor in the MPI derived datatypes.
Contiguous datatype constructors create a new datatype by
making *count* copies of existing data type (*old_type*)

```
MPI_Datatype rowtype;  
...  
MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);  
MPI_Type_commit(&rowtype);  
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Type_contiguous.html

Example : Derived Datatypes - Contiguous

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[];
{
int numtasks, rank, source=0, dest, tag=1, i;
MPI_Request req;
float a[SIZE][SIZE] =
{1.0, 2.0, 3.0, 4.0,
 5.0, 6.0, 7.0, 8.0,
 9.0, 10.0, 11.0, 12.0,
 13.0, 14.0, 15.0, 16.0};
float b[SIZE];
MPI_Status stat;
MPI_Datatype rowtype;
```

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Type_contiguous(SIZE, MPI_FLOAT,
&rowtype);
MPI_Type_commit(&rowtype);
if (numtasks == SIZE) {
    if (rank == 0) {
        for (i=0; i<numtasks; i++){
            dest = i;
            MPI_Isend(&a[i][0], 1, rowtype, dest,
tag, MPI_COMM_WORLD, &req);
        }
    }
    MPI_Recv(b, SIZE, MPI_FLOAT, source, tag,
MPI_COMM_WORLD, &stat);
    printf("rank= %d b= %3.1f %3.1f %3.1f %3.1f\n",
           rank,b[0],b[1],b[2],b[3]);
}
else
    printf("Must specify %d processors. Terminating.\n",SIZE);
MPI_Type_free(&rowtype);
MPI_Finalize();
}
```

<https://computing.llnl.gov/tutorials/mpi/>

Example : Derived Datatypes - Contiguous

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc, argv) {
    int numtasks, rank, source=0, dest, tag=1, i;
    MPI_Request req;
    float a[SIZE][SIZE] =
        {1.0, 2.0, 3.0, 4.0,
         5.0, 6.0, 7.0, 8.0,
         9.0, 10.0, 11.0, 12.0,
         13.0, 14.0, 15.0, 16.0};
    float b[SIZE];
    MPI_Status stat;
    MPI_Datatype rowtype;
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

Declares a 4x4 array of
datatype float

```
PI_Init(&argc, &argv);
PI_Comm_rank(MPI_COMM_WORLD, &rank);
PI_Comm_size(MPI_COMM_WORLD, &numtasks);
PI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
PI_Type_commit(&rowtype);
if (numtasks == SIZE) {
    if (rank == 0) {
        for (i=0; i<numtasks; i++) {
            dest = i;
            MPI_Isend(a[i], 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD, &req);
        }
    }
    MPI_Recv(b, 1, MPI_FLOAT, MPI_COMM_WORLD, tag, MPI_COMM_WORLD, &stat);
    printf("rank= %d, b= %3.1f %3.1f %3.1f %3.1f\n", rank, b[0], b[1], b[2], b[3]);
}
else
    printf("Must specify %d processors. Terminating.\n", SIZE);
MPI_Type_free(&rowtype);
MPI_Finalize();
}
```

Homogenous datastructure
of size 4 (Type : rowtype)

<https://computing.llnl.gov/tutorials/mpi/>

Example : Derived Datatypes - Contiguous

MPI_Type_contiguous

```
count = 4;  
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
[lsu00@master derived]$ qsub contiguous.pbs  
531.master.arete.cct.lsu.edu  
[lsu00@master derived]$ cat out.txt  
rank= 0  b= 1.0 2.0 3.0 4.0  
rank= 1  b= 5.0 6.0 7.0 8.0  
rank= 3  b= 13.0 14.0 15.0 16.0  
rank= 2  b= 9.0 10.0 11.0 12.0
```

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```

9.0	10.0	11.0	12.0
-----	------	------	------

1 element of
rowtype

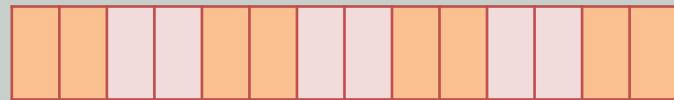
Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives
- Derived Datatypes: Introduction
- Derived Datatypes: Contiguous
- **Derived Datatypes: Vector**
- Derived Datatypes: Indexed
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- MPI Profiling calls
- Additional Topics
- Summary

Derived Datatypes : Vector

Function: **MPI_Type_vector()**

```
int MPI_Type_vector(  
    int count,  
    int blocklen,  
    int stride,  
    MPI_Datatype old_type,  
    MPI_Datatype *newtype )
```



count = 4
blocklen = 2
stride = 4

Description:

Returns a new datatype that represents equally spaced blocks. The spacing between the start of each block is given in units of extent (*oldtype*). The *count* represents the number of blocks, *blocklen* details the number of elements in each block, *stride* represents the number of elements between start of each block of the *old_type*. The new datatype is stored in *new_type*

```
...  
MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);  
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Type_vector.html

Example : Derived Datatypes - Vector

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[]; {
int numtasks, rank, source=0, dest, tag=1, i;
MPI_Request req;
float a[SIZE][SIZE] =
{1.0, 2.0, 3.0, 4.0,
 5.0, 6.0, 7.0, 8.0,
 9.0, 10.0, 11.0, 12.0,
 13.0, 14.0, 15.0, 16.0};
float b[SIZE];

MPI_Status stat;
MPI_Datatype columntype;
```

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
MPI_Type_commit(&columntype);

if (numtasks == SIZE) {
    if (rank == 0) {
        for (i=0; i<numtasks; i++)
            MPI_Isend(&a[0][i], 1, columntype, i, tag,
MPI_COMM_WORLD, &req);
    }
}

MPI_Recv(b, SIZE, MPI_FLOAT, source, tag,
MPI_COMM_WORLD, &stat);
printf("rank= %d b= %3.1f %3.1f %3.1f %3.1f\n",
       rank,b[0],b[1],b[2],b[3]);
}
else
printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Type_free(&columntype);
MPI_Finalize();
}
```

<https://computing.llnl.gov/tutorials/mpi/>

Example - Point-to-Point Datatypes - Vector

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[]; {
int numtasks, rank,
MPI_Request req;
float a[SIZE][SIZE] =
{1.0, 2.0, 3.0, 4.0,
 5.0, 6.0, 7.0, 8.0,
 9.0, 10.0, 11.0, 12.0,
 13.0, 14.0, 15.0, 16.0};
float b[SIZE];
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

Declares a 4x4 array of datatype float

```
argc,&argv);
rank(MPI_COMM_WORLD, &rank);
size(MPI_COMM_WORLD, &numtasks);

e_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
commit(&columntype);
```

```
source=0, dest, tag=1, i;
if (rank == 0) {
    for (i=0; i<numtasks; i++)
        MPI_Isend(&a[0][i],
```

```
MPI_COMM_WORLD,
```

```
}
```

**Homogenous
datastructure of size 4
(Type : columntype)**

```
else
printf("Must specify %d processors. Terminating.\n",SIZE);
```

```
MPI_Type_free(&columntype);
MPI_Finalize();
}
```

Example : Derived Datatypes - Vector

MPI_Type_vector

```
count = 4; blocklength = 1; stride = 4;  
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT,  
    &columntype);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
[lsu00@master derived]$ qsub vector.pbs  
562.master.arete.cct.lsu.edu  
[lsu00@master derived]$ cat out.txt  
rank= 0 b= 1.0 5.0 9.0 13.0  
rank= 2 b= 3.0 7.0 11.0 15.0  
rank= 1 b= 2.0 6.0 10.0 14.0  
rank= 3 b= 4.0 8.0 12.0 16.0
```

```
MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);
```

2.0	6.0	10.0	14.0
-----	-----	------	------

1 element of
columntype

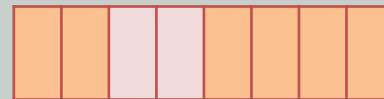
Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives
- Derived Datatypes: Introduction
- Derived Datatypes: Contiguous
- Derived Datatypes: Vector
- **Derived Datatypes: Indexed**
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- MPI Profiling calls
- Additional Topics
- Summary

Derived Datatypes : Indexed

Function: **MPI_Type_indexed()**

```
int MPI_Type_indexed(  
    int count,  
    int *array_of_blocklengths,  
    int *array_of_displacements,  
    MPI_Datatype oldtype,  
    MPI_datatype *newtype);
```



count = 2
array_of_blocklengths[2] = {2, 4}
array_of_displacements[2] = {0, 4}

Description:

Returns a new datatype that represents count blocks. Each block is defined by an entry in *array_of_blocklengths* and *array_of_displacements*. Displacements are expressed in units of extent(*oldtype*). The *count* is the number of blocks and the number of entries in *array_of_displacements* (displacement of each block in units of the *oldtype*) and *array_of_blocklengths* (number of instances of *oldtype* in each block).

```
...  
MPI_Type_indexed(2, blocklengths, displacements, MPI_FLOAT,  
&indextype);  
...
```

https://computing.llnl.gov/tutorials/mpi/man/MPI_Type_indexed.txt

Example : Derived Datatypes - Indexed

```
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 6

int main(argc,argv)
int argc;
char *argv[]; {
int numtasks, rank, source=0, dest, tag=1, i;
MPI_Request req;
int blocklengths[2], displacements[2];
float a[16] =
{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
float b[NELEMENTS];

MPI_Status stat;
MPI_Datatype indextype;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
```

```
blocklengths[0] = 4;
blocklengths[1] = 2;
displacements[0] = 5;
displacements[1] = 12;
```

```
MPI_Type_indexed(2, blocklengths,
displacements, MPI_FLOAT, &indextype);
```

```
MPI_Type_commit(&indextype);
```

```
if (rank == 0) {
    for (i=0; i<numtasks; i++)
        MPI_Isend(a, 1, indextype, i, tag,
MPI_COMM_WORLD, &req);
}
```

```
MPI_Recv(b, NELEMENTS, MPI_FLOAT, source,
tag, MPI_COMM_WORLD, &stat);
```

```
printf("rank= %d b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
    rank,b[0],b[1],b[2],b[3],b[4],b[5]);
```

```
MPI_Type_free(&indextype);
MPI_Finalize();
}
```

Example : Derived Datatypes - Indexed

```
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 6
```

```
blocklengths[0] = 4;
blocklengths[1] = 2;
displacements[0] = 5;
displacements[1] = 12;
```

```
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0
```

Creates a new datatype *indextype*

```
    MPI_Type_commit(&indextype);
```

Declares a[16] array of type float

```
float a[16] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
```

for (i=0; i<NELEMENTS; i++) a[i] = i + displacements[i];

```
    MPI_Isend(a, 1, indextype, i, tag,
              MPI_COMM_WORLD, &req);
    MPI_Recv(b, NELEMENTS, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
    MPI_Type_free(&indextype);
    MPI_Finalize();
}
```

for (i=0; i<NELEMENTS; i++) b[i] = rank * blocklengths[i] + displacements[i];

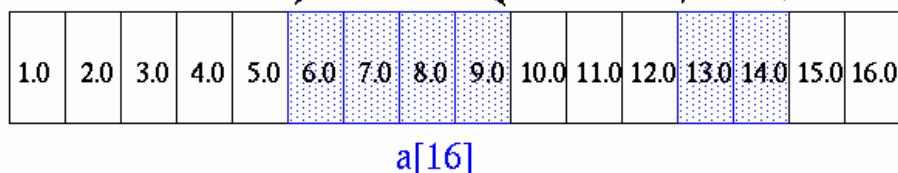
```
    printf("rank= %d b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
           rank,b[0],b[1],b[2],b[3],b[4],b[5]);
```

Example : Derived Datatypes - Indexed

MPI_Type_indexed

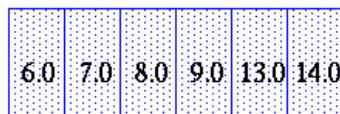
```
count = 2;    blocklengths[0] = 4;  
displacements[0] = 5;
```

```
blocklengths[1] = 2;  
displacements[1] = 12;
```



```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
```

```
MPI_Send(&a, 1, indextype, dest, tag, comm);
```



1 element of
indextype

<https://computing.llnl.gov/tutorials/mpi/>

Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives
- Derived Datatypes: Introduction
- Derived Datatypes: Contiguous
- Derived Datatypes: Vector
- Derived Datatypes: Indexed
- **Derived Datatypes: Struct**
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- MPI Profiling calls
- Additional Topics
- Summary

Derived Datatypes : struct

Function: **MPI_Type_struct()**

```
int MPI_Type_struct(  
    int count,  
    int *array_of_blocklengths,  
    MPI_Aint *array_of_displacements,  
    MPI_Datatype *array_of_types,  
    MPI_datatype *newtype);
```

Deprecated since MPI-2.0
Removed since MPI-3.0
Replacement: MPI_Type_create_struct

Description:

Returns a new datatype that represents *count* blocks. Each is defined by an entry in *array_of_blocklengths*, *array_of_displacements* and *array_of_types*. Displacements are expressed in bytes. *count* is an integer that specifies the number of blocks (number of entries in arrays). The *array_of_blocklengths* is the number of elements in each blocks & *array_of_displacements* specifies the byte displacement of each block. The *array_of_types* parameter comprising each block is made of concatenation of type *array_of_types*.

```
...  
MPI_Type_struct(2, blockcounts, offsets, oldtypes, &particletype);  
...
```

https://computing.llnl.gov/tutorials/mpi/man/MPI_Type_struct.txt

Example : Derived Datatype - struct

```
#include "mpi.h"
#include <stdio.h>
#define NELEM 25
int main(argc,argv)
int argc;
char *argv[];
int numtasks, rank, source=0, dest, tag=1, i;
typedef struct {
    float x, y, z;
    float velocity;
    int n, type;
} Particle;
Particle p[NELEM], particles[NELEM];
MPI_Datatype particletype, oldtypes[2];
int blockcounts[2];
MPI_Aint offsets[2];

/* MPI_Aint type used to be consistent with syntax of */
/* MPI_Type_get_extent routine */
MPI_Aint lb, float_extent;

MPI_Status stat;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

MPI_Type_get_extent(MPI_FLOAT, &lb, &float_extent);
```

```
blockcounts[0] = 4; blockcounts[1] = 2;
offsets[0] = 0; offsets[1] = 4 * float_extent;
oldtypes[0] = MPI_FLOAT; oldtypes[1] = MPI_INT;
MPI_Type_create_struct(2, blockcounts, offsets, oldtypes,
&particletype);
MPI_Type_commit(&particletype);

if (rank == 0) {
    for (i=0; i<NELEM; i++) {
        particles[i].x = i * 1.0;
        particles[i].y = i * -1.0;
        particles[i].z = i * 1.0;
        particles[i].velocity = 0.25;
        particles[i].n = i;
        particles[i].type = i % 2;
    }
    for (i=0; i<numtasks; i++)
        MPI_Send(particles, NELEM, particletype, i, tag, MPI_COMM_WORLD);
}
MPI_Recv(p, NELEM, particletype, source, tag, MPI_COMM_WORLD, &stat);
printf("rank= %d  %3.2f %3.2f %3.2f %3.2f %d %d\n", rank,p[3].x,
      p[3].y,p[3].z,p[3].velocity,p[3].n,p[3].type);

MPI_Type_free(&particletype);
MPI_Finalize();
}
```

Example : Derived Datatype - struct

```
#include "mpi.h"
#include <stdio.h>
#define NELEM 25
int main(argc,argv)
int argc;
char *argv[];
int numtasks, rank, source=0, dest, tag=1, i;
typedef struct {
    float x, y, z;
    float velocity;
    int n, type;
} Particle;
Particle p[NELEM], particles[NELEM];
MPI_Datatype particletype, oldtypes[2];
int blockcounts[2];
MPI_Aint offsets[2];

/* MPI_Aint type used to be consistent with syntax of */
/* MPI_Type_get_extent routine */
MPI_Aint lb, float_extent;

MPI_Status stat;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

MPI_Type_get_extent(MPI_FLOAT, &lb, &float_extent);
```

Declaring the structure of
the heterogeneous
datatype
Float, Float, Float, Float,
Int, Int

```
blockcounts[0] = 4; blockcounts[1] = 2;
offsets[0] = 0; offsets[1] = 4 * float_extent;
oldtypes[0] = MPI_FLOAT; oldtypes[1] = MPI_INT;
MPI_Type_create_struct(2, blockcounts, offsets, oldtypes,
&particletype);
MPI_Type_commit(&particletype);

if (rank == 0) {
    for (i=0; i<NELEM; i++) {
        particles[i].x = i * 1.0;
        particles[i].y = i * -1.0;
        particles[i].z = i * 1.0;
        particles[i].velocity = 0.0;
        particles[i].n = i;
        particles[i].type = i % 2;
    }
    for (i=0; i<numtasks; i++)
        MPI_Send(particles, NELEM, MPI_FLOAT, i, tag, MPI_COMM_WORLD);
    MPI_Recv(p, NELEM, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &stat);
    printf("rank= %d  %3.2f %3.2f %3.2f %3.2f\n",
           p[3].x,p[3].y,p[3].z,p[3].velocity,p[3].n);
}

MPI_Type_free(&particletype);
MPI_Finalize();
```

Construct the
heterogeneous
datatype as an MPI
datatype using Struct

Populate the
heterogenous MPI
datatype with
heterogeneous data

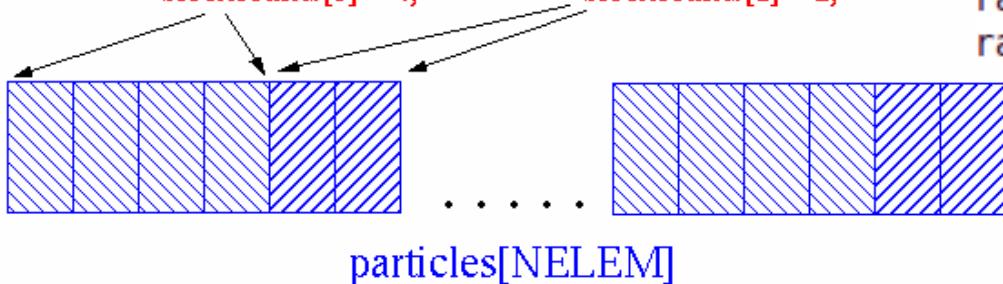
Example : Derived Datatype - struct

MPI_Type_struct

```
typedef struct { float x,y,z,velocity; int n,type; } Particle;  
Particle particles[NELEM];
```

```
MPI_Type_extent(MPI_FLOAT,&extent);
```

```
count = 2; oldtypes[0] = MPI_FLOAT; oldtypes[1] = MPI_INT  
offsets[0] = 0; offsets[1] = 4 * extent;  
blockcounts[0] = 4; blockcounts[1] = 2;
```



```
MPI_Type_struct(count,blockcounts,offsets,oldtypes,&particletype);
```

```
MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

```
[lsu00@master derived]$ qsub struct.pbs  
564.master.arete.cct.lsu.edu  
[lsu00@master derived]$ cat out.txt  
rank= 0 3.00 -3.00 3.00 0.25 3 1  
rank= 1 3.00 -3.00 3.00 0.25 3 1  
rank= 2 3.00 -3.00 3.00 0.25 3 1  
rank= 3 3.00 -3.00 3.00 0.25 3 1
```

Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives
- Derived Datatypes: Introduction
- Derived Datatypes: Contiguous
- Derived Datatypes: Vector
- Derived Datatypes: Indexed
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- MPI Profiling calls
- Additional Topics
- Summary

MPI: Pack and Unpack

- We explicitly pack non-contiguous data into a contiguous buffer for transmission, then unpack it at the other end.
- When sending/receiving packed messages, must use `MPI_PACKED` datatype in send/receive calls.

MPI: Packing

```
int MPI_Pack(void *inbuf, int incount,  
MPI_Datatype datatype, void *outbuf, int  
outsize, int *position, MPI_Comm comm)
```

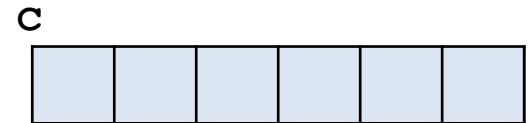
- will pack the information specified by `inbuf` and `incount` into the buffer space provided by `outbuf` and `outsize`
- the current packing call will pack this data starting at `offset` position in the `outbuf`
- `position` is incremented by the size of the packed message

MPI: Unpacking

```
int MPI_Unpack(void *inbuf, int insize, int  
*position, void *outbuf, int outcount,  
MPI_Datatype datatype, MPI_Comm comm)
```

- unpacks message to outbuf.
- updates the position argument so it can be used in a subsequent call to MPI_Unpack.

Example



buffer



↑ position

```
int i, position=0;  
char c[100]; buffer[110];  
//pack  
MPI_Pack(&i,1,MPI_INT,buffer,110,&position,MPI_COMM_WORLD);  
MPI_Pack(c,100,MPI_CHAR,buffer,110,&position,MPI_COMM_WORLD);  
//send  
MPI_Send(buffer,position,MPI_PACKED,1,0,MPI_COMM_WORLD);  
...  
  
//correspongding receive  
// int position = 0;  
MPI_Recv(buffer,110,MPI_PACKED,0,0,MPI_COMM_WORLD,&status);  
//and unpack  
MPI_Unpack(buffer,110,&position,&i,1,MPI_INT,MPI_COMM_WORLD);  
MPI_Unpack(buffer,110,&position,c,100,MPI_CHAR,MPI_COMM_WORLD);  
...
```

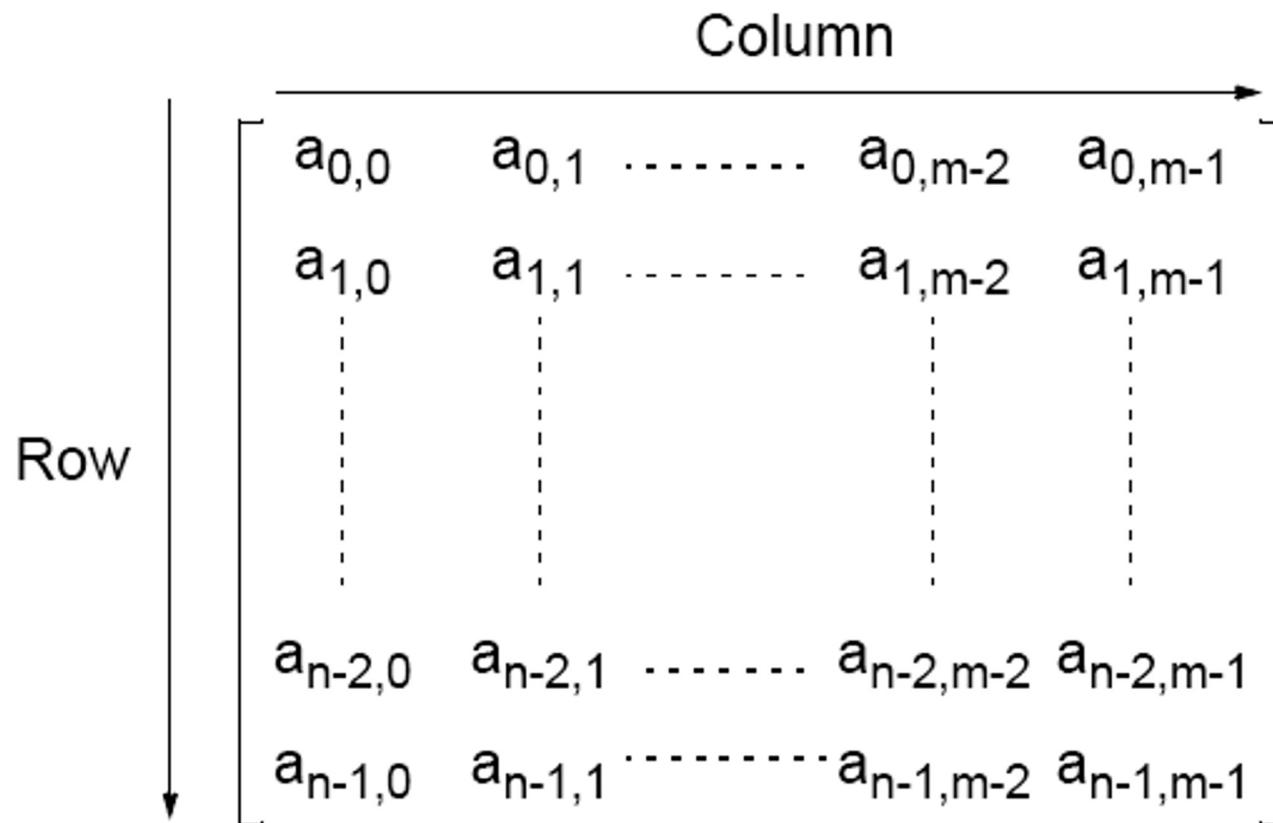
MPI: Derived Datatypes vs. Pack/Unpack

- Pack/Unpack is quicker/easier to program.
- Derived datatypes are more flexible in allowing complex derived datatypes to be defined.
- Pack/Unpack has higher overhead.
- Derived datatypes are better if the datatype is regularly reused.

Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives
- Derived Datatypes: Introduction
- Derived Datatypes: Contiguous
- Derived Datatypes: Vector
- Derived Datatypes: Indexed
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- **Matrix-Vector multiplication : A Case Study**
- MPI Profiling calls
- Additional Topics
- Summary

Matrix Vector Multiplication



Matrix Vector Multiplication

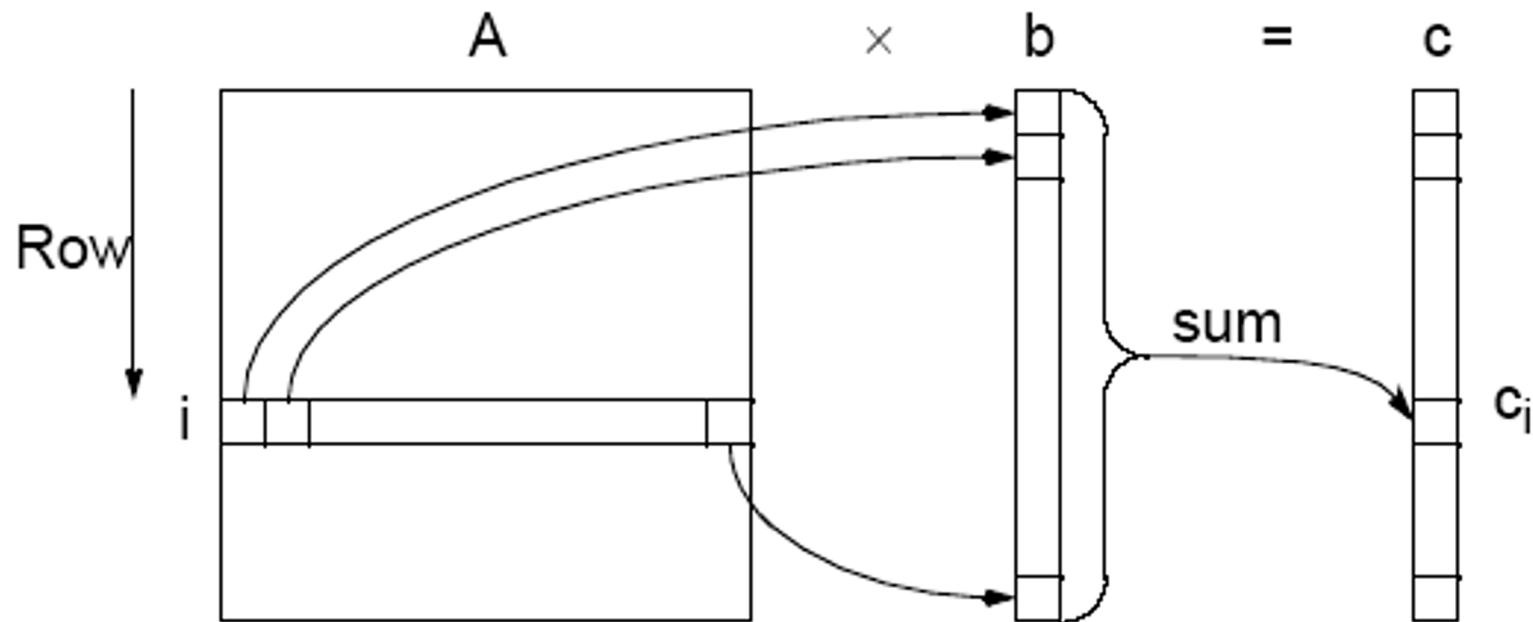
Multiplication of a matrix, **A** and a vector **B**, produces the vector **C** whose elements, c_i ($0 \leq i < n$), are computed as follows:

$$C_i = \sum_{k=0}^{k=m} A_{ik} * B_k$$

where **A** is an $n \times m$ matrix and **B** is a vector of size m and **C** is a vector of size n .

Matrix-Vector Multiplication

$$\mathbf{c} = \mathbf{A} \times \mathbf{b}$$



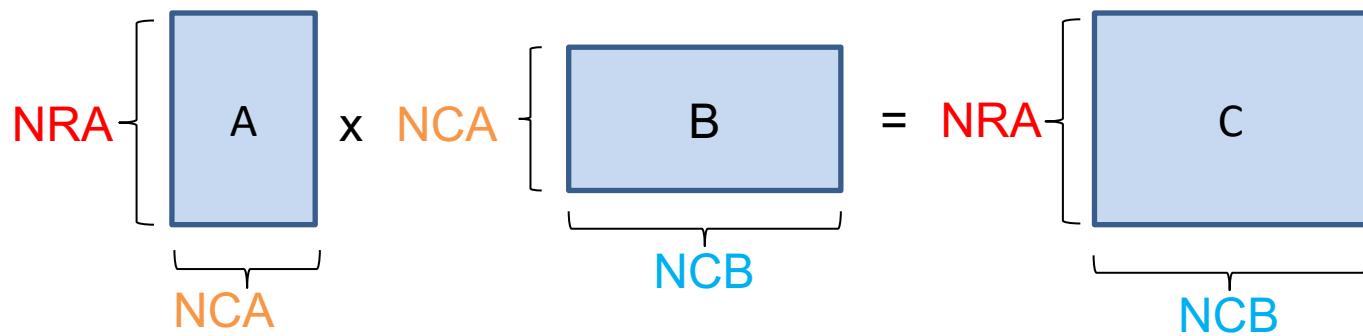
Example: Matrix-Vector Multiplication

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define NRA 4      /* number of rows in matrix A */
#define NCA 4      /* number of columns in matrix A */
#define NCB 1      /* number of columns in matrix B */
#define MASTER 0    /* taskid of first task */
#define FROM_MASTER 1 /* setting a message type */
#define FROM_WORKER 2 /* setting a message type */
```

```
int main (int argc, char *argv[])
{
int      numtasks,      /* number of tasks in partition */
        taskid,        /* a task identifier */
        numworkers,     /* number of worker tasks */
        source,         /* task id of message source */
        dest,          /* task id of message destination */
        mtype,          /* message type */
        rows,           /* rows of matrix A sent to each worker */
averow, extra, offset, /* used to determine rows sent to each worker */
i, j, k, rc;          /* misc */
```

Define the dimensions of
the Matrix $a([4][4])$ and
Vector $b([4][1])$



Example: Matrix-Vector Multiplication

```
double    a[NRA][NCA],    /* Matrix A to be multiplied */  
          b[NCA][NCB],    /* Vector B to be multiplied */  
          c[NRA][NCB];    /* result Vector C */
```

```
MPI_Status status;
```

```
MPI_Init(&argc,&argv);  
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);  
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);  
if (numtasks < 2 ) {  
    printf("Need at least two MPI tasks. Quitting...\n");  
    MPI_Abort(MPI_COMM_WORLD, rc);  
    exit(1);  
}  
numworkers = numtasks-1;  
***** master task *****  
if (taskid == MASTER)  
{  
    printf("mpi_mm has started with %d tasks.\n",numtasks);  
    printf("Initializing arrays...\n");  
    for (i=0; i<NRA; i++)  
        for (j=0; j<NCA; j++)  
            a[i][j]= i+j;  
    for (i=0; i<NCA; i++)  
        for (j=0; j<NCB; j++)  
            b[i][j]= (i+1)*(j+1);
```

Declare the matrix , vector
to be multiplied and the
resultant vector

MASTER Initializes the Matrix A :

0.00	1.00	2.00	3.00
1.00	2.00	3.00	4.00
2.00	3.00	4.00	5.00
3.00	4.00	5.00	6.00

MASTER Initializes B :

1.00
2.00
3.00
4.00

Example: Matrix-Vector Multiplication

```
for (i=0; i<NRA; i++)  
{  
    printf("\n");  
    for (j=0; j<NCA; j++)  
        printf("%6.2f ", a[i][j]);  
}  
for (i=0; i<NRA; i++)  
{  
    printf("\n");  
    for (j=0; j<NCB; j++)  
        printf("%6.2f ", b[i][j]);  
}  
/* Send matrix data to the worker tasks */  
averow = NRA/numworkers;  
extra = NRA%numworkers;  
offset = 0;  
mtype = FROM_MASTER;  
for (dest=1; dest<=numworkers; dest++)  
{  
    rows = (dest <= extra) ? averow+1 : averow;  
    printf("Sending %d rows to task %d offset=%d\n",rows,dest,offset);  
    MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);  
    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);  
    MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,  
            MPI_COMM_WORLD);  
    MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);  
    offset = offset + rows;  
}
```

Load Balancing : Dividing
the Matrix A based on the
number of processors

MASTER sends Matrix A to workers :

PROC[1] :: 0.00	1.00	2.00	3.00
PROC[1] :: 1.00	2.00	3.00	4.00
PROC[2] :: 2.00	3.00	4.00	5.00
PROC[3] :: 3.00	4.00	5.00	6.00

MASTER Sends Vector B to Workers:

PROC[1] :: 1.00	2.00	3.00	4.00
PROC[2] :: 1.00	2.00	3.00	4.00
PROC[3] :: 1.00	2.00	3.00	4.00

Example: Matrix-Vector Multiplication

```
/* Receive results from worker tasks */
mtype = FROM_WORKER;
for (i=1; i<=numworkers; i++)
{
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype,
             MPI_COMM_WORLD, &status);
    printf("Received results from task %d\n",source);
}

/* Print results */
printf("*****\n");
printf("Result Matrix:\n");
for (i=0; i<NRA; i++)
{
    printf("\n");
    for (j=0; j<NCB; j++)
        printf("%6.2f ", c[i][j]);
}
printf("\n*****\n");
printf ("Done.\n");
}
```

The Master process gathers the results and populates the result matrix in the correct order (easily done in this case because matrix index i is used to indicate position in result array)

Example: Matrix-Vector Multiplication

```
***** worker task *****:  
if (taskid > MASTER)  
{  
    mtype = FROM_MASTER;  
    MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);  
    MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);  
    MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);  
    MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);  
  
    for (k=0; k<NCB; k++)  
        for (i=0; i<rows; i++)  
        {  
            c[i][k] = 0.0;  
            for (j=0; j<NCA; j++)  
                c[i][k] = c[i][k] + a[i][j] * b[j][k];  
        }  
    mtype = FROM_WORKER;  
    MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);  
    MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);  
    MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);  
}  
MPI_Finalize();  
}
```

Worker Processes receive workload

Proc[1]	A:	0.00	1.00	2.00	3.00
Proc[1]	A:	1.00	2.00	3.00	4.00
Proc[1]	B :	1.00	2.00	3.00	4.00

Calculate Result

Proc[1]	C[0]:	0.00 + 2.00 + 6.00 + 12.00
Proc[1]	C[1]:	1.00 + 4.00 + 9.00 + 16.00

Worker sends result to MASTER

Proc[1]	C[0]:	20.00
Proc[1]	C[1]:	30.00

Example: Matrix-Vector Multiplication (Results)

```
[lsu00@master mm]$ qsub mm.pbs
568.master.arete.cct.lsu.edu
[lsu00@master mm]$ cat out.txt
mpi_mm has started with 4 tasks.
Initializing arrays...
*****
0.00      1.00      2.00      3.00
1.00      2.00      3.00      4.00
2.00      3.00      4.00      5.00
3.00      4.00      5.00      6.00
*****
1.00
2.00
3.00
4.00
*****
Sending 2 rows to task 1 offset=0
Sending 1 rows to task 2 offset=2
Sending 1 rows to task 3 offset=3
Received results from task 1
Received results from task 2
Received results from task 3
*****
Result Matrix:
20.00
30.00
40.00
50.00
*****
Done.
```

Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives
- Derived Datatypes: Introduction
- Derived Datatypes: Contiguous
- Derived Datatypes: Vector
- Derived Datatypes: Indexed
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- **MPI Profiling calls**
- Additional Topics
- Summary

MPI Profiling : MPI_Wtime

Function: **MPI_Wtime()**

double MPI_Wtime()

Description:

Returns time in seconds elapsed on the calling processor. Resolution of time scale is determined by the MPI environment variable **MPI_WTICK**. When the MPI environment variable **MPI_WTIME_IS_GLOBAL** is defined and set to true, the value of **MPI_Wtime** is synchronized across all processes in **MPI_COMM_WORLD**

```
double time0;  
...  
time0 = MPI_Wtime();  
...  
printf("Hello From Worker # %d %lf \n", rank, (MPI_Wtime() - time0));
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Wtime.html

Timing Example: MPI_Wtime

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char **argv)
{
    int size, rank;
    double time0, time1;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    time0 = MPI_Wtime();

    if(rank==0)
    {
        printf(" Hello From Proc0 Time = %lf \n", (MPI_Wtime() - time0));
    }
    else
    {
        printf("Hello From Worker # %d %lf \n", rank, (MPI_Wtime() - time0));
    }
    MPI_Finalize();
}
```

Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives
- Derived Datatypes: Introduction
- Derived Datatypes: Contiguous
- Derived Datatypes: Vector
- Derived Datatypes: Indexed
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- MPI Profiling calls
- Additional Topics
- Summary

Potential Pitfall

- We need to be sure that every receive has a matching send
 - Suppose process q calls
 - `MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag, send_comm);`
 - And process r calls
 - `MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv tag, recv_comm, &status);`
 - The message sent by q with the above call to `MPI_Send` can be received by r with the call to `MPI_Recv` if:
 - `recv_comm = send_comm,`
 - `recv_tag = send_tag,`
 - `dest = r, and`
 - `src = q`

Wildcard Arguments

- `MPI_ANY_SOURCE`
- `MPI_ANY_TAG`
- Only a receiver can use a wildcard argument. Senders must specify a process rank and a nonnegative tag
- Example:
 - If process 0 executes the following code, it can receive the results in the order in which the processes finish:

```
for (i = 1; i < comm_sz; i++) {  
    MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,  
             result_tag, comm, MPI_STATUS_IGNORE);  
    Process_result(result);  
}
```

Input/Ouput

- Although the MPI standard doesn't specify which processes have access to which I/O devices, virtually all MPI implementations allow *all* the processes in `MPI_COMM_WORLD` full access to `stdout` and `stderr`
 - so most MPI implementations allow all processes to execute `printf` and `fprintf(stderr, ...)`
- Unlike output, most MPI implementations only allow process 0 in `MPI_COMM_WORLD` access to `stdin`
 - This makes sense: If multiple processes have access to `stdin`, which process should get which parts of the input data?

Collective vs. Point-to-point Communications

- All the processes in the communicator must call the same collective function
 - For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process is erroneous, and, in all likelihood, the program will hang or crash.
- The arguments passed by each process to an MPI collective communication must be “compatible.”
 - For example, if one process passes in 0 as the dest process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.
- Collective communications don’t use tags

Collective vs. Point-to-point Communications (Cont'd)

- The names of the memory locations are irrelevant to the matching, of the calls to MPI_Reduce
 - The order of the calls will determine the matching
 - For example, suppose that each process calls MPI_Reduce with operator MPI_SUM, and destination process 0
 - The value stored in b will be $1 + 2 + 1 = 4$
 - The value stored in d will be $2 + 1 + 2 = 5$

Table 3.3 Multiple Calls to MPI_Reduce

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

MPI topics we did Not Cover

- Topologies: map a communicator onto, say, a 3D Cartesian processor grid
 - Implementation can provide ideal logical to physical mapping
- Rich set of I/O functions: individual, collective, blocking and non-blocking
 - Collective I/O can lead to many small requests being merged for more efficient I/O
- One-sided communication: puts and gets with various synchronization schemes
 - Implementations not well-optimized and rarely used
 - Redesign of interface is underway
- Task creation and destruction: change number of tasks during a run
 - Few implementations available

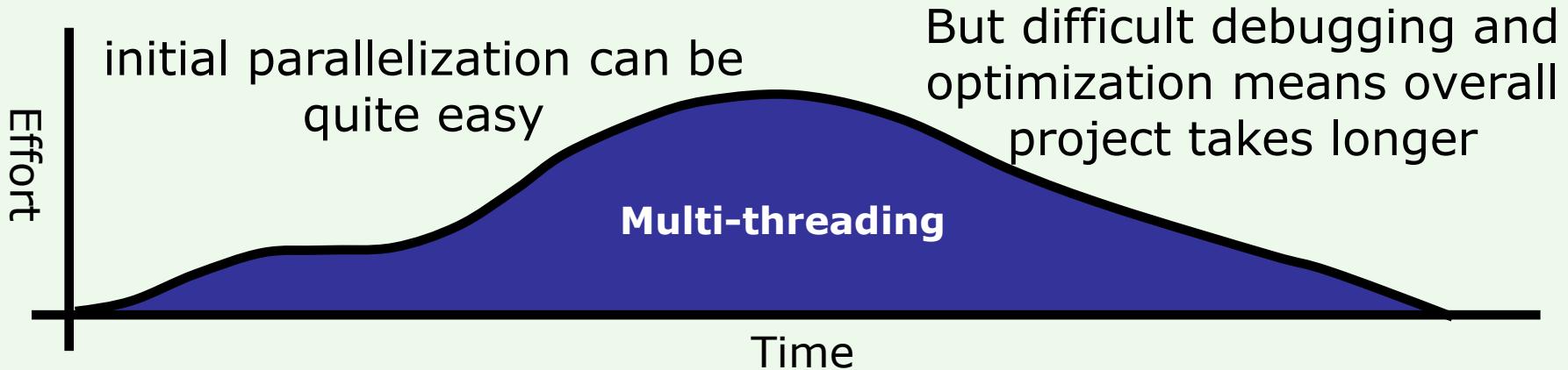
Topics

- MPI Collective Calls: Synchronization Primitives
- MPI Collective Calls: Communication Primitives
- MPI Collective Calls: Reduction Primitives
- Derived Datatypes: Introduction
- Derived Datatypes: Contiguous
- Derived Datatypes: Vector
- Derived Datatypes: Indexed
- Derived Datatypes: Struct
- MPI_Pack/MPI_Unpack
- Matrix-Vector multiplication : A Case Study
- MPI Profiling calls
- Additional Topics
- Summary

MPI isn't as hard as many believe

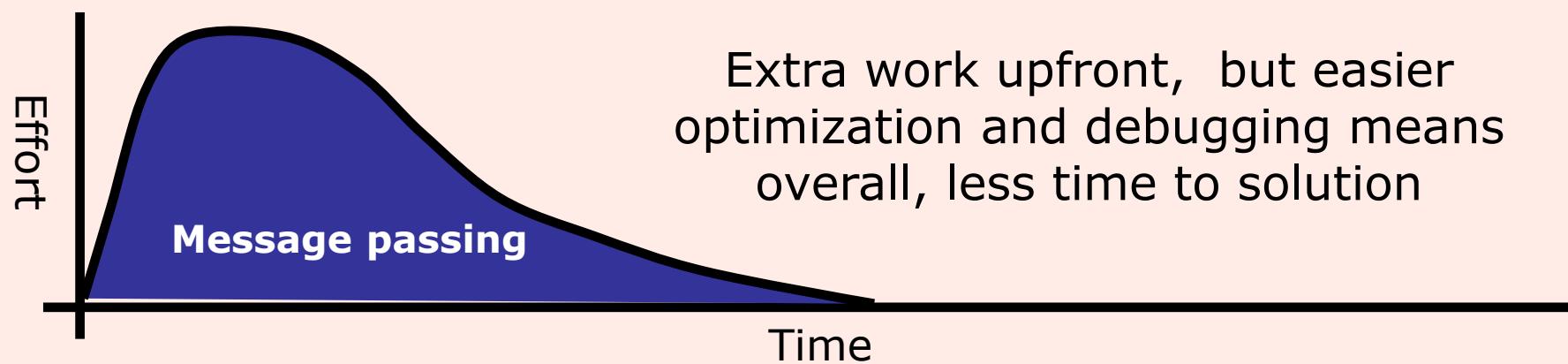
- There are over 330 functions in the MPI spec, but most programs only use a small subset:
 - Startup
 - `MPI_Init`, `MPI_Finalize`
 - Information on the processes
 - `MPI_Comm_rank`, `MPI_Comm_size`,
 - Point-to-point communication
 - `MPI_Irecv`, `MPI_Isend`, `MPI_Wait`, `MPI_Send`, `MPI_Recv`
 - Collective communication
 - `MPI_Reduce`, `MPI_Allreduce`, `MPI_Bcast`, `MPI_Allgather`

Multithreading vs Message Passing



Proving that a shared address space program using semaphores is race free is an NP-complete problem

P. N. Klein, H. Lu, and R. H. B. Netzer, Detecting Race Conditions in Parallel Programs that Use Semaphores, Algorithmica, vol. 35 pp. 321-345, 2003



Source: Tim Mattson, "Recent developments in parallel programming: the good, the bad, and the ugly", Euro-Par 2013

