TEXAS TECH UNIVERSITY

CS4379: Parallel and Concurrent Programming
CS5379: Parallel Processing

Lecture 23

Dr. Yong Chen

Associate Professor

Computer Science Department

Texas Tech University

Lecture Video

- Please view the lecture video either from Teams or from the below link:

  https://texastechuniversity.sharepoint.com/sites/CS4379-CS5379/Shared%20Documents/General/Lecture23.mp4

Course Info

- **Lecture Time**: TR, 12:30-1:50
- **Lecture Location**: ECE 217
- **Sessions**: CS4379-001, CS4379-002, CS5379-001, CS5379-D01
- **Instructor:** Yong Chen, Ph.D., Associate Professor
- **Email:** yong.chen@ttu.edu
- **Phone:** 806-834-0284
- **Office**: Engineering Center 315
- **Office Hours**: 2-4 p.m. on Wed., or by appointment
- **TA:** Mr. Ghazanfar Ali, Ghazanfar.Ali@ttu.edu
- **TA Office hours:** Tue. and Fri., 2-3 p.m., or by appointment
- **TA Office:** EC 201 A
- **More info:**
  - http://www.myweb.ttu.edu/yonchen
  - http://discl.cs.ttu.edu; http://cac.ttu.edu/; http://nsfcac.org

Outline

- Questions?

- Collective Communications in MPI
- More on message passing: communication modes and avoiding deadlocks
- Datatypes in MPI

Collective Communications in MPI

- Collective operations are called by all processes in a communicator
- No message tags used
- In many applications, point-to-point can be replaced by collective communication, improving both simplicity and efficiency
  - Let the internal implementation optimize the communication for you

- Three broad classes:
  - Synchronization: barrier
  - Data movement routines: broadcast, gather, scatter
  - Global computation routines: reduction, scan

Barrier Routine

- Used to synchronize execution of a group of processes:

  int MPI_Barrier(MPI_Comm comm);

- A barrier is a simple way to separate two phases of computation to ensure that messages in two phases do no interact

Data Movement Routines

- Broadcast routine implements a one-to-all broadcast where a single named process (root) sends the same data to all other processes

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

buffer *starting address of buffer (choice)*
count *number of entries in buffer (integer)*
datatype *data type of buffer (handle)*
root *rank of broadcast root (integer)*
comm *communicator (handle)*

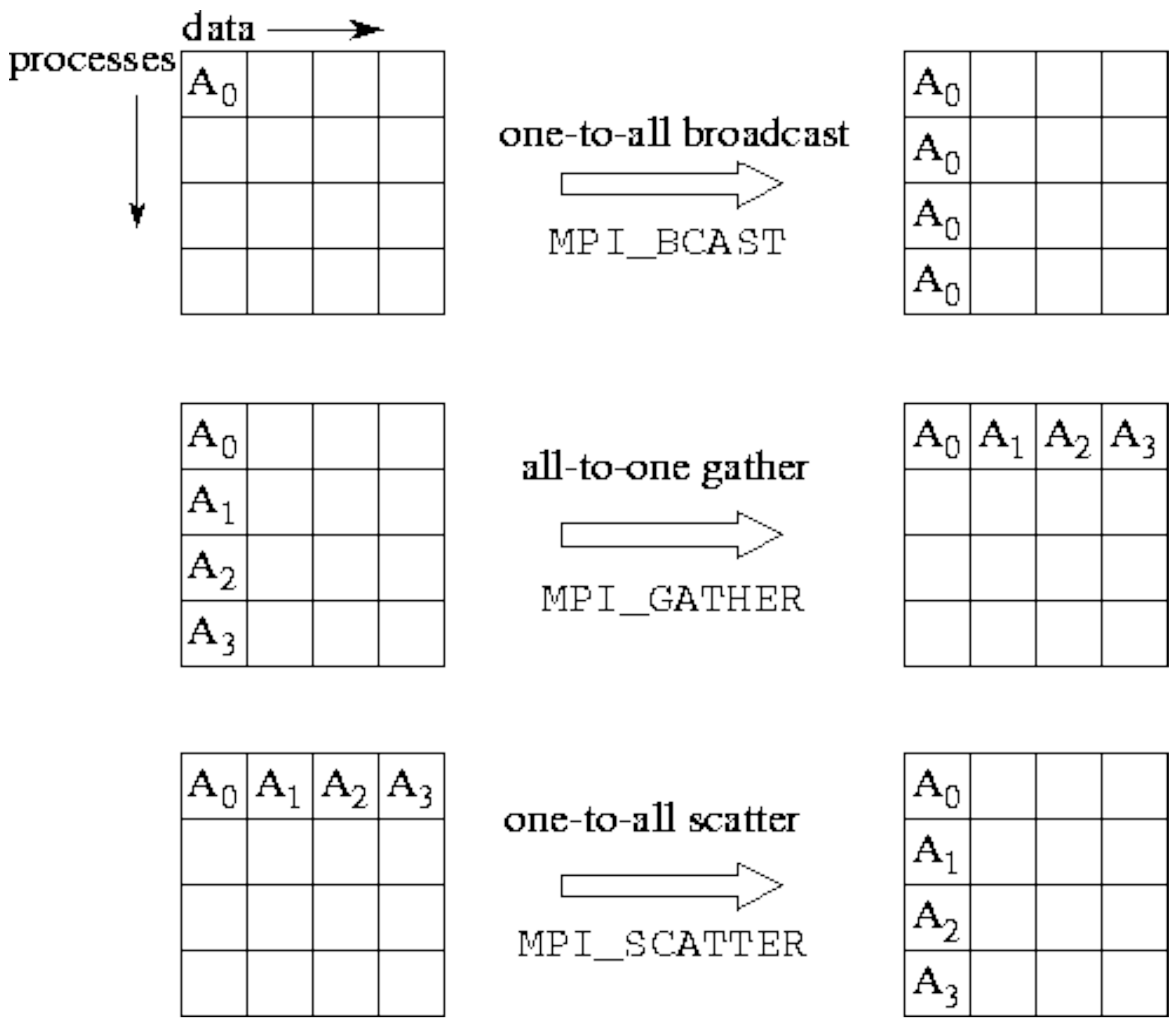
Data Movement Routines

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

int MPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)

sendbuf *starting address of send buffer (choice)*
sendcount *number of elements in send buffer (integer)*
sendtype *data type of send buffer elements (handle)*
recvcount *number of elements for any single receive (integer, significant only at root)*
recvtype *data type of recv buffer elements (significant only at root) (handle)*
root *rank of receiving/sending process (integer)*
comm *communicator (handle)*
recvbuf *address of receive buffer (choice, significant only at root) (OUT)*

Illustration of MPI Communication Functions



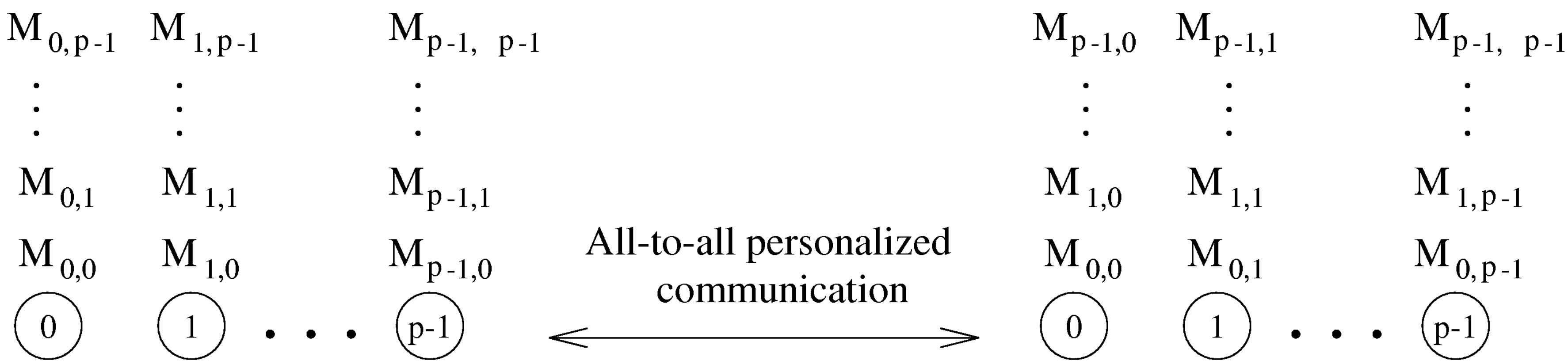
Collective Communication Operations

- The all-to-all personalized communication operation is performed by:

```
int MPI_Alltoall(void *sendbuf, int sendcount,
  MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
  MPI_Comm comm)
```

- Each process sends to process *i* sendcount contiguous elements of type senddatatype starting from the *i* * sendcount location of its sendbuf
- The data that are received are stored in the recvbuf array
- Each process receives from process *i* recvcount elements of type recvdatatype and stores them in its recvbuf array starting at location *i* * recvcount

All-to-all Personalized Communication



Reduction Operations

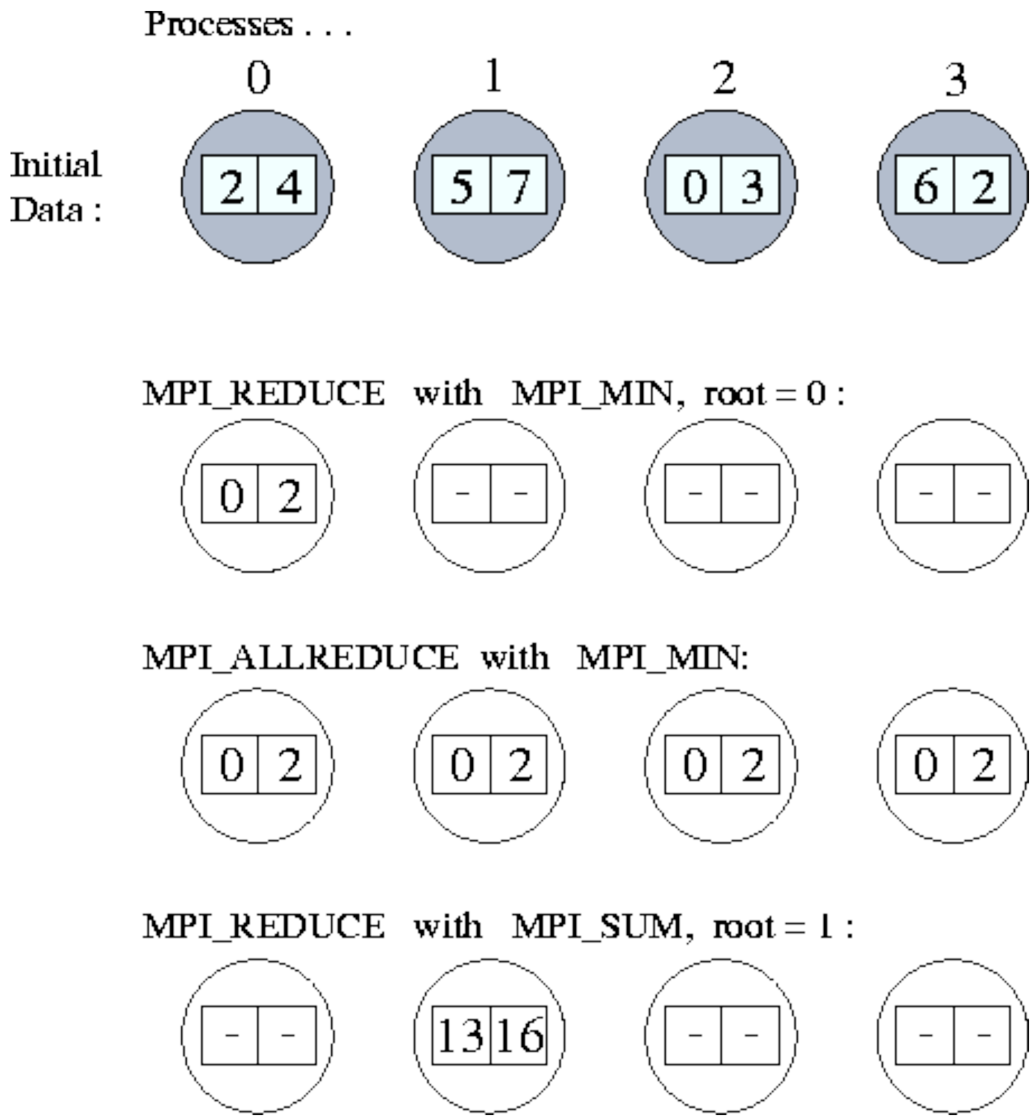
- Reduction operations combine the values provided in the input buffer of each process using a specified operation OP, and return combined value into output buffer of single root process or output buffer of all processes

int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

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

sendbuf *address of send buffer (choice)*
count *number of elements in send buffer (integer)*
datatype *data type of elements of send buffer (handle)*
op *reduce operation (handle)*
root *rank of root process (integer)*
comm *communicator (handle)*
recvbuf *address of receive buffer (choice, significant only at root) (OUT)*

Illustration of Reduction Operations

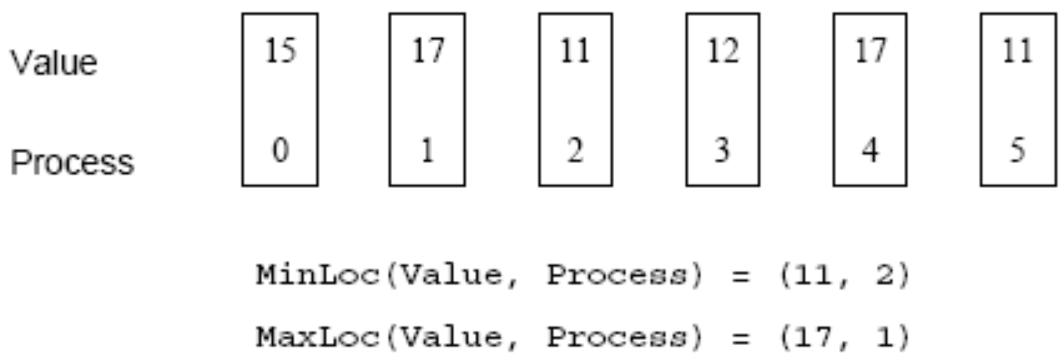


MPI Built-in Collective Computation Operations

`MPI_Max`	Maximum
`MPI_Min`	Minimum
`MPI_Prod`	Product
`MPI_Sum`	Sum
`MPI_Land`	Logical and
`MPI_Lor`	Logical or
`MPI_Lxor`	Logical exclusive or
`MPI_Band`	Binary and
`MPI_Bor`	Binary or
`MPI_Bxor`	Binary exclusive or
`MPI_Maxloc`	Maximum and location
`MPI_Minloc`	Minimum and location

Collective Communication Operations

- The operation `MPI_MAXLOC` combines pairs of values (v_i, l_i) and returns the pair (v, l) such that v is the maximum among all v_i 's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i 's).
- `MPI_MINLOC` does the same, except for minimum value of v_i.



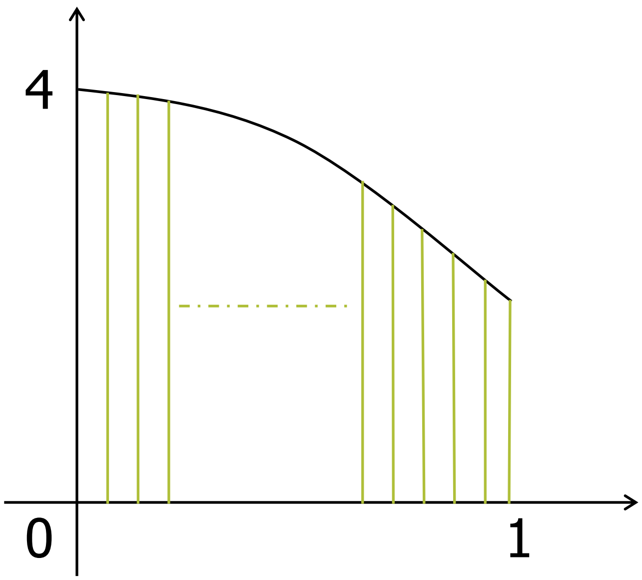
An example use of the `MPI_MINLOC` and `MPI_MAXLOC` operators.

Example: Calculating Pi

- One way to calculate Pi:

$$\pi = \int_0^1 \frac{4}{1+x^2}dx$$

- Calculating Pi via numerical integration
  - Divide and assign to processes
  - Each process calculates partial sum
  - Add all the partial sums together to get Pi



Example: PI in C (1/2)

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.141592653589793238462643;
   double mypi, pi, width, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done) {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
     }

     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     if (n == 0) break;
```

message/data broadcasted

root process

Example: PI in C (2/2)

```
    width = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = width * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = width * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is %.16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
  return 0;
}
```

send buffer

receive buffer

operation

root process

More on MPI Collective Routines

- Many Routines: **Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, ReduceScatter, Scan, Scatter, Scatterv**
- **All** versions deliver results to all participating processes.
- V versions allow the chunks to have different sizes.
- **Allreduce**, **Reduce**, **ReduceScatter**, and **Scan** take both built-in and user-defined combiner/reduction functions.
  - Create your own collective computations with:
    MPI_Op_create(user_fcn, commutes, &op);
    MPI_Op_free(&op);

Outline

- Questions?

- Collective Communications in MPI
- More on message passing: communication modes and avoiding deadlocks
- Datatypes in MPI

Communication Modes

- MPI has multiple *communication modes* for sending messages:
  - Synchronous mode (MPI_Ssend): the send waits for a matching receive has begun. (Unsafe programs deadlock.)
  - Buffered mode (MPI_Bsend): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.)
  - Ready mode (MPI_Rsend): user guarantees that a matching receive has been posted.
    - Allows access to fast protocols
    - Undefined behavior if matching receive not posted
- Non-blocking versions (MPI_Issend, etc.)
- MPI_Recv/MPI_IRecv receives messages sent in any mode

Flavors of Communication

- For a send operation there are:
  - 4 communication modes: standard,ready,synchronous,buffered
  - 2 blocking modes: blocking, nonblocking
  - 4*2 =8 types of sends
- For a receive operation there are:
  - 1 communication mode: standard
  - 2 blocking modes: blocking, nonblocking
  - 1*2 = 2 types of receive

Naming Conventions

- Send routines:

Comm Mode	Blocking	Non-blocking
Standard	MPI_Send	MPI_Isend
Ready	MPI_Rsend	MPI_Irsend
Synchronous	MPI_Ssend	MPI_Issend
Buffered	MPI_Bsend	MPI_Ibsend

- Receive routines:

Comm Mode	Blocking	Non-blocking
Standard	MPI_Recv	MPI_Irecv

- Any type of receive routine can be used to receive messages from any type of send routine

Send/Receive Operations

- In many applications, processes send to one process while receiving from another
- Deadlock may arise if care is not taken
- MPI provides routines for such send/receive operations

```
int MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest,
                  int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype,
                  int source, int recvtag, MPI_Comm comm, MPI_Status *status )
```

Deadlocks

- What happens with this code?

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

- MPI_Send and MPI_Recv are blocking comm:
  - MPI_Send does not complete until sending buffer is empty
  - MPI_Recv does not complete until receiving buffer is full

- Simple, but maybe “unsafe” because completion depends on the message size and availability of system buffers

Solutions to the Unsafe Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	`Sendrecv(0)`

More Solutions to the Unsafe Problem

- Supply own space as buffer for send

Process 0	Process 1
`Bsend(1)` **`Recv(1)`**	**`Bsend(0)`** **`Recv(0)`**

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)` **`Irecv(1)`** **`Waitall`**	**`Isend(0)`** **`Irecv(0)`** **`Waitall`**

Outline

- Questions?

- Collective Communications in MPI
- More on message passing: communication modes and avoiding deadlocks
- Datatypes in MPI

Datatypes in MPI

- MPI datatypes have two purposes:
  - Heterogeneity
  - Noncontiguous data

- Basic vs. derived datatype:
  - Basic datatype
  - Derived datatype
    - Contiguous
    - Vector
    - Indexed
    - Hindexed
    - Structure

Basic Datatype in C

MPI Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	singed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSINGED_SHORT	unsigned short

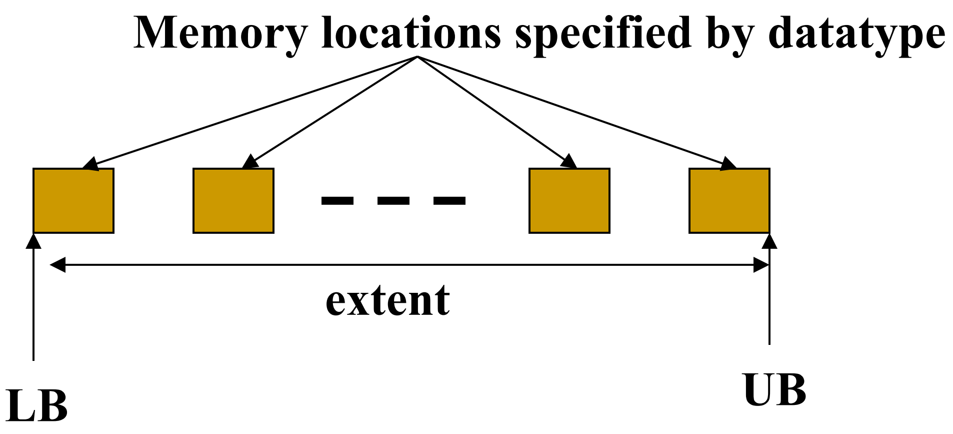
Additional datatypes defined in MPI 2.2 corresponding to C99 language types int32_t, int64_t, etc.

Typemaps in MPI

- In MPI, a datatype is represented as a typemap

$$typemap = (type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})$$

- Extent of a datatype



- An artificial extent can be set by using MPI_UB and MPI_LB

Typemaps in MPI (cont.)

- Example:
  - (int,0),(char,4) is a typemap
  - The extent of this typemap is 5

CONTIGUOUS Datatype

MPI_Type_contiguous(count, oldtype, &newtype)
MPI_Type_commit(&newtype)

- Assume an original datatype oldtype has typemap (double,0), (char,8), then
  MPI_Type_contiguous(3,oldtype,&newtype);

- To actually send such a data use the sequence of calls:
  MPI_Type_contiguous(count,datatype,&newtype);
  MPI_Type_commit(&newtype);
  MPI_Send(buffer,1,newtype,dest,tag,comm);
  MPI_Type_free(&newtype);

VECTOR Datatype

MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)
MPI_Type_commit(&newtype)

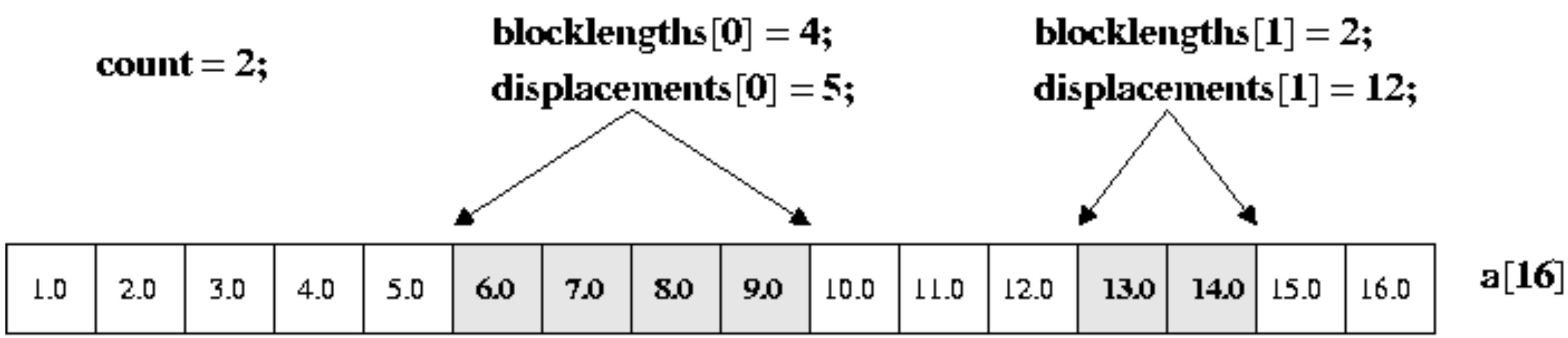
29	30	31	32	33	34	35
22	23	24	25	26	27	28
15	16	17	18	19	20	21
8	9	10	11	12	13	14
1	2	3	4	5	6	7

MPI_Type_Vector(5,1,7,MPI_DOUBLE,newtype);
MPI_Type_commit(&newtype);
MPI_Send(buffer,1,newtype,dest,tag,comm);
MPI_Type_free(&newtype);

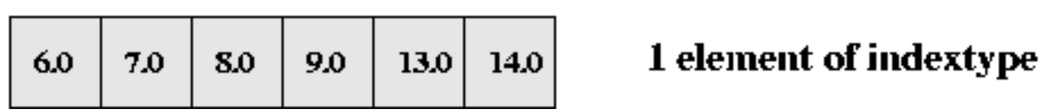
INDEXED Datatype

MPI_Type_indexed(count, &array_of_blocklengths, &array_of_displacements, oldtype, &newtype)
MPI_Type_commit(&newtype)

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



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



Hindexed:
MPI_Type_hindexed()
is the same except that offsets array is specified in bytes

Structure Datatype

MPI_Type_struct(count, &array_of_blocklengths, &array_of_displacements, oldtypes, &newtype)

MPI_Type_commit(&newtype)

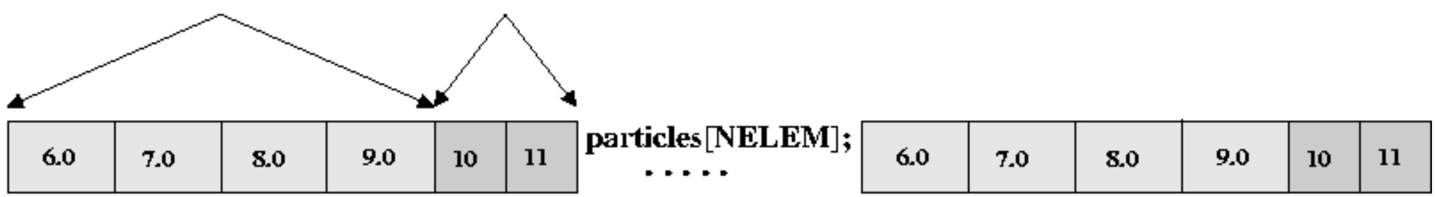
MPI_Type_struct(count, blocklengths, offsets, oldtypes, &particletype);

typedef struct { float f1, f2, f3, f4; int n1, n2;} Particle;

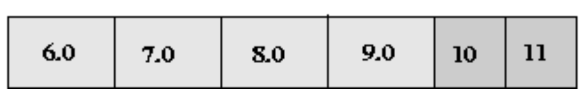
Particle particles[NELEM];

count = 2;

oldtypes[0] = MPI_FLOAT; oldtypes[1] = MPI_INT;
offsets[0] = 0; offsets[1] = 4 * sizeof(float);
blockcounts[0] = 4; blockcounts[1] = 2;



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



1 element of particletype

Readings

- Reference book ITPC – Chapter 6, 6.3-6.6

Questions?

Questions/Suggestions/Comments are always welcome!

Write me: yong.chen@ttu.edu
Call me: 806-834-0284
See me: ENGCTR 315

If you write me an email for this class, please start the email subject with [CS4379] or [CS5379].