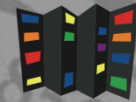


Introduction to MPI

Martin Čuma

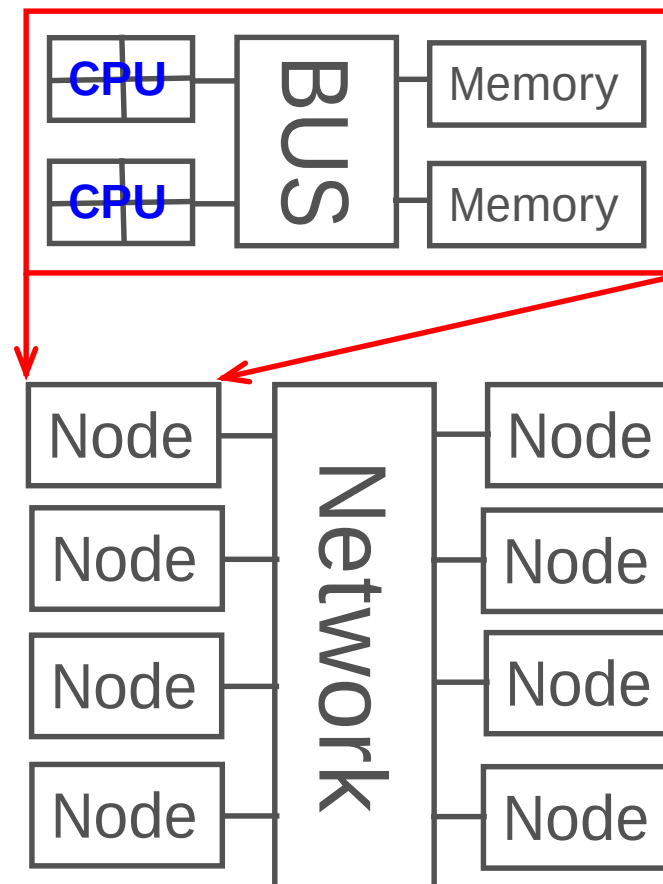
*Center for High Performance
Computing University of Utah
m.cuma@utah.edu*

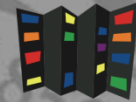


- Quick introduction (in case you slept/missed last time).
- MPI concepts, initialization.
- Point-to-point communication.
- Collective communication.
- Grouping data for communication.
- Quick glance at advanced topics.
- Survey

<https://www.surveymonkey.com/r/8GGBT78>

- Process has access only to its local memory
- Data between processes must be communicated
- More complex programming
- Cheap commodity hardware
- CHPC: Linux clusters





- Standardized message-passing library
 - uniform API
 - guaranteed behavior
 - source code portability
- Complex set of operations
 - various point-to-point communication
 - collective communication
 - process groups
 - processor topologies
 - one sided communication (RMA)
 - parallel I/O

```
program hello
integer i, n, ierr, my_rank, nodes
include "mpif.h"

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
call MPI_Comm_rank(MPI_COMM_WORLD,my_rank,ierr)
if (my_rank .eq. 0) then
  do i=1,nproc-1
    call MPI_Recv(n,1,MPI_INTEGER,i,0,MPI_COMM_WORLD,
&      status,ierr)
    print*, 'Hello from process',n
  enddo
else
  call MPI_Send(my_rank,1,MPI_INTEGER,0,0,MPI_COMM_WORLD,ierr)
endif
call MPI_Finalize(ierr)
return
```

```
ember1:~>%module load mpich
ember1:~>%mpif77 ex1.f -o ex1
em001:~>%srun -n 12 -N 1 -A chpc -p ember -t 1:00:00 -
pty=/bin/tcsh -l
em001:~>%mpirun -np 4 ex1
```

```
Hello from process      1
Hello from process      2
Hello from process      3
```

- must be included in subroutines and functions that use MPI calls
- provide required declarations and definitions
- Fortran – `mpif.h`
 - declarations of MPI-defined datatypes
 - error codes
- C – `mpi.h`
 - also function prototypes

- Initializing MPI:
 - `MPI_Init(ierr)`
 - `int MPI_Init(int *argc, char **argv)`
- Terminating MPI
 - `MPI_Finalize(ierr)`
 - `int MPI_Finalize()`
- Determine no. of processes
 - `MPI_Comm_Size(comm, size, ierr)`
 - `int MPI_Comm_Size(MPI_comm comm, int* size)`
- Determine rank of the process
 - `MPI_Comm_Rank(comm, rank, ierr)`
 - `int MPI_Comm_Rank(MPI_comm comm, int* rank)`

- Sending data

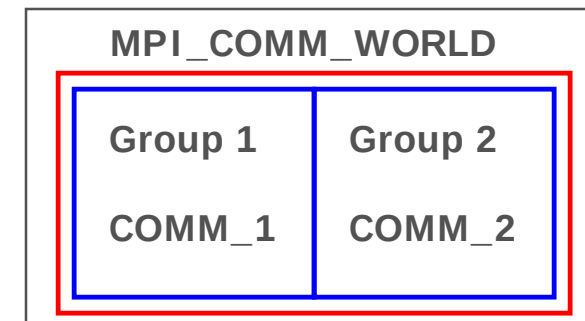
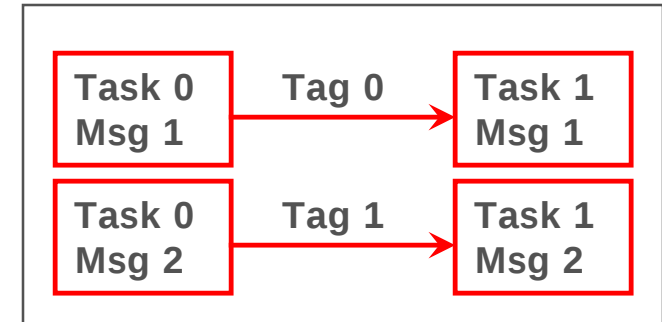
- `MPI_Send(buf, count, datatype, dest, tag, comm, ierr)`
- `int MPI_Send(void *buf, int count, MPI_Datatype, int dest, int tag, MPI_comm comm)`
call `MPI_Send(my_rank, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, ierr)`

- Receiving data

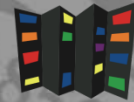
- `MPI_Recv(buf, count, datatype, source, tag, comm, status, ierr)`
- `int MPI_Recv(void *buf, int count, MPI_Datatype, int source, int tag, MPI_comm comm, MPI_Status status)`
call `MPI_Recv(n, 1, MPI_INTEGER, i, 0, MPI_COMM_WORLD, status, ierr)`



- Data (buffer, count)
- Sender / Recipient
- Message envelope
 - data type – see next two slides
 - tag – integer to differentiate messages
 - communicator – group of processes that take place in the communication
default group communicator – MPI_COMM_WORLD



Predefined data structures



MPI Datatype	Fortran Datatype
MPI_BYTE	
MPI_CHARACTER	CHARACTER
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_REAL	REAL
MPI_INTEGER	INTEGER
MPI_LOGICAL	LOGICAL
MPI_PACKED	

Predefined data structures

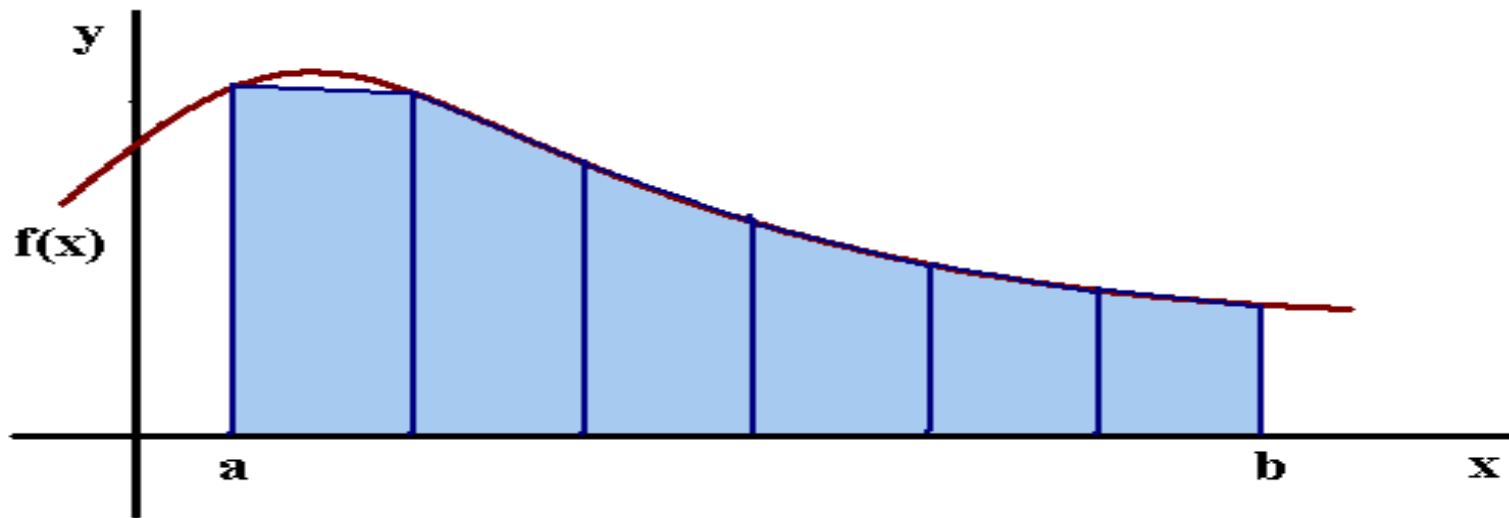
MPI Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
...	...
MPI_PACKED	

- Initiates operation and returns
 - overlap communication with computation
 - receive requires 2 function calls – initiate the communication, and finish it
 - prepend function name with I and use request handle at the end of message

```
call MPI_Irecv(n,1,MPI_INTEGER,i,0,MPI_COMM_WORLD,status,req,ierr)
```
 - usually completed at the point when the communicated data are to be used
 - consume system resources, which must be released (MPI_Wait, MPI_Test)

```
call MPI_Wait(req,status, ierr)
```

$$\int_a^b f(x) \approx \sum_{i=1}^n \frac{1}{2} h [f(x_{i-1}) + f(x_i)] = \frac{1}{2} h [f(x_0) + f(x_n)] + \sum_{i=1}^{n-1} h [f(x_i)]$$



1. Initialize MPI
 2. Get interval and no. of trapezoids
 3. Broadcast input to all processes
 4. Each process calculates its interval
 5. Collect the results from all the processes
- New concepts:
 - collective communication – involves more processes
 - explicit work distribution
 - derived data types – more efficient data transfer

```
#include <stdio.h>
#include "mpi.h"
int main (int argc, char* argv[]){
int p, my_rank, n , i , local_n;
float a, b, h, x, integ, local_a, local_b, total;
MPI_Datatype mesg_ptr;
float f(float x);
void Build_der_data_t(float *a,float *b,int *n,MPI_Datatype
    *mesg_ptr);
```

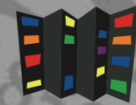
1. MPI_Init(&argc,&argv);
2. MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
2. MPI_Comm_size(MPI_COMM_WORLD,&p);
- if (my_rank == 0) {
 printf("Input integ. interval, no. of trap:\n");
 scanf("%f %f %d",&a,&b,&n);}
3. Build_der_data_t(&a,&b,&n, &mesg_ptr);
- MPI_Bcast(&a,1,mesg_ptr,0,MPI_COMM_WORLD);

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

    integ = (f(local_a)+f(local_b))/2.;
    x = local_a;
    for (i=1;i<local_n;i++){
        x = x+h;
        integ = integ+ f(x);}
    integ = integ*h;
    printf("Trapezoids n = %d, local integral from ",local_n);
    printf("%f to %f is %f\n",local_a,local_b,integ);
    total = 0.;

5.  MPI_Reduce(&integ,&total,1,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD);
    if (my_rank == 0)
        printf("Total integral = %f\n",total);
    MPI_Finalize();
    return 0;}
```

```
em001:~>% mpicc trapp.c -o trapp
em001:~>% mpirun -np 4 ./trapp
Input integ. interval, no. of trap:
0 10 100
Trapezoids n = 25, local integral from 0.000000 to
  2.500000 is 5.212501
Total integral = 333.350098
Trapezoids n = 25, local integral from 2.500000 to
  5.000000 is 36.462475
Trapezoids n = 25, local integral from 5.000000 to
  7.500000 is 98.962471
Trapezoids n = 25, local integral from 7.500000 to
  10.000000 is 192.712646
```



- Broadcast – from one node to the rest
 - `MPI_Bcast(buf, count, datatype, root, comm, ierr)`
 - `int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_comm comm)`

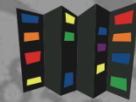
On root, `buf` is data to be broadcast, on other nodes it's data to be received

- Reduction – collect data from all nodes
 - `MPI_Reduce(sndbuf, rcvbuf, count, datatype, op, root, comm, ierr)`
 - `int MPI_Reduce(void *sndbuf, void *rcvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_comm comm)`

`MPI_Reduce(&integ, &total, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);`

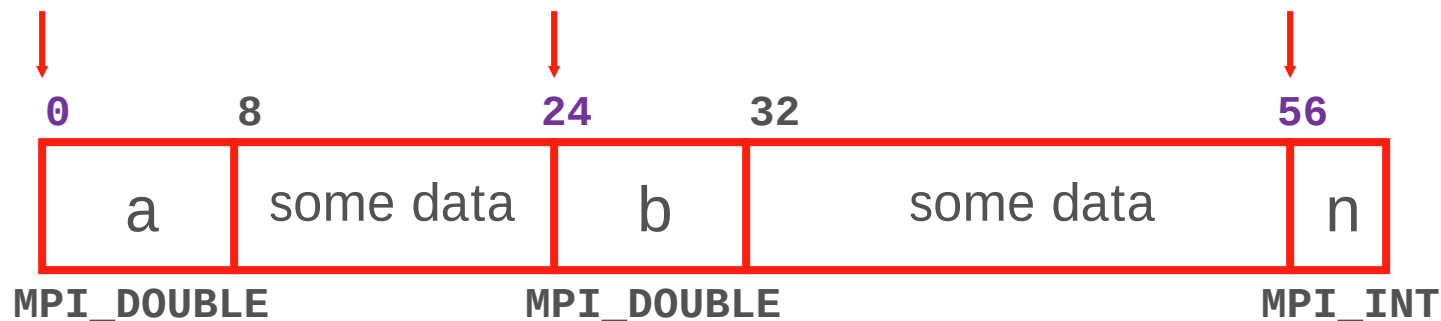
Supported operations, e.g. `MPI_MAX`, `MPI_MIN`, `MPI_SUM`,...

Result stored in `rcvbuf` only on processor with rank `root`.



- Communication operations that involve more than one process
 - *broadcast* from one process to all the others in the group
 - *reduction* collect data from all the processes in certain manner (sum, max,...)
 - *barrier synchronization* for all processes of the group
 - *gather* from all group processes to one process
 - *scatter* distribute data from one process to all the others
 - *all-to-all gather/scatter/reduce* across the group
- NOTE: There is no implicit barrier before collective communication operations, but there is a barrier after

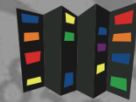
- Used to group data for communication
- Built from basic MPI data types
- Must specify:
 - number of data variables in the derived type and their length (1,1,1)
 - type list of these variables (MPI_DOUBLE, MPI_DOUBLE, MPI_INT)
 - displacement of each data variable in bytes from the beginning of the message (0,24,56)



```
void Build_der_data_t(float *a, float *b,  
                      int *n, MPI_Datatype *mesg_ptr){  
    int blk_len[3] = {1, 1, 1};  
    MPI_Aint displ[3], start_addr, addr;  
    MPI_Datatype type1[3] = {MPI_FLOAT, MPI_FLOAT, MPI_INT};  
  
    displ[0] = 0;  
    MPI_Get_address(a, &start_addr);  
    MPI_Get_address(b, &addr);  
    displ[1] = addr - start_addr;  
    MPI_Get_address(n, &addr);  
    displ[2] = addr - start_addr;  
  
    MPI_Type_create_struct(3, blk_len, displ, type1, mesg_ptr);  
    MPI_Type_commit(mesg_ptr);  
}
```

- Address displacement
 - `MPI_Get_address(location, address)`
 - `int MPI_Get_address(void *location, MPI_Aint *address)`
- Derived data type create
 - `MPI_Type_create_struct(count, bl_len, displ, typelist, new_mpi_t)`
 - `int MPI_Type_create_struct(int count, int bl_len[], MPI_Aint displ[], MPI_Datatype typelist[], MPI_Datatype *new_mpi_t)`
`MPI_Type_create_struct(3, blk_len, displ, type1, msg_ptr);`
- Derived data type commit/free
 - `MPI_Type_commit(new_mpi_t)`
 - `int MPI_Type_commit(MPI_Datatype *new_mpi_t)`
 - `MPI_Type_free(new_mpi_t)`
 - `int MPI_Type_free(MPI_Datatype *new_mpi_t)`

- Simpler d.d.t. constructors
 - `MPI_Type_contiguous`
= contiguous entries in an array
 - `MPI_Type_vector`
= equally spaced entries in an array
 - `MPI_Type_indexed`
= arbitrary entries in an array



```
void Exch_data(float *a, float *b, int *n, int my_rank){  
    char buffer[100];  
    int position = 0;  
  
    if (my_rank == 0){  
        MPI_Pack(a, 1, MPI_FLOAT, buffer, 100, &position, MPI_COMM_WORLD);  
        MPI_Pack(b, 1, MPI_FLOAT, buffer, 100, &position, MPI_COMM_WORLD);  
        MPI_Pack(n, 1, MPI_INT, buffer, 100, &position, MPI_COMM_WORLD);  
        MPI_Bcast(buffer, 100, MPI_PACKED, 0, MPI_COMM_WORLD);  
    }  
    else{  
        MPI_Bcast(buffer, 100, MPI_PACKED, 0, MPI_COMM_WORLD);  
        MPI_Unpack(buffer, 100, &position, a, 1, MPI_FLOAT, MPI_COMM_WORLD);  
        MPI_Unpack(buffer, 100, &position, b, 1, MPI_FLOAT, MPI_COMM_WORLD);  
        MPI_Unpack(buffer, 100, &position, n, 1, MPI_INT, MPI_COMM_WORLD);  
    }  
}
```

- Explicit storing of noncontiguous data for communication
- Pack – before send
 - `MPI_Pack(pack_data, in_cnt, datatype, buf, buf_size, position, comm, ierr)`
 - `int MPI_Pack(void *pack_data, int in_cnt, MPI_Datatype datatype, void *buf, int buf_size, int *position, MPI_comm comm)`
`MPI_Pack(a, 1, MPI_FLOAT, buffer, 100, &position, MPI_COMM_WORLD);`
- Unpack – after receive
 - `MPI_Unpack(buf, size, position, unpack_data, cnt, datatype, comm, ierr)`
 - `int MPI_Unpack(void *buf, int size, int *position, void *unpack_data, int cnt, MPI_Datatype datatype, MPI_comm comm)`
 - position gets updated after every call to MPI_Pack/Unpack
`MPI_Unpack(buffer, 100, &position, a, 1, MPI_FLOAT, MPI_COMM_WORLD);`

- count and datatype
 - sending contiguous array or a scalar
- MPI_Pack/Unpack
 - sending heterogeneous data only once
 - variable length messages (sparse matrices)
- Derived data types
 - everything else, including:
 - repeated send of large heterogeneous data
 - sending of large strided arrays

- Advanced point-to-point communication
- Specialized collective communication
- Process groups, communicators
- Virtual processor topologies
- Error handling
- MPI I/O
- Dynamic processes
- One sided communication

Distributed matrix-vector multiply, $\text{vecout}[M] = A[M][M] * \text{vecin}[M]$

```
double _Complex out1[FNxy], out2[FNxy], *commbuf, *compbuf;
```

2 communication buffers +
comm and comp pointers

```
for (iis=0; iis <= FNlocs; iis++) {
```

```
    if (iis%2==1) commbuf=out2; else commbuf=out1;
```

swap comm. buffer

```
    xind = iis*FNxy;
```

```
    MPI_Iallgatherv(&(vecin[xind]), FNxy, MPI_DOUBLE_COMPLEX, commbuf, counts,
stride, MPI_DOUBLE_COMPLEX, MPI_COMM_WORLD, &allg_handle);
```

```
    if (iis%2==0) compbuf=out2; else compbuf=out1;
```

swap comp. buffer

```
    if (iis>0) {
```

```
        for (iir=0; iir<FNloc; iir++) {
```

```
            iy = iir*FNxy;
```

vector offset

```
            for (ip=0; ip<numprocs; ip++) {
```

```
                ia = iy + ipoffA[ip] * Fnxy*FNloc;
```

```
                for (ix=0; ix<FNxy; ix++)
```

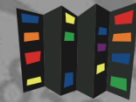
matrix offset

```
                    vecout[iy+ix] += A[(ia+ix)] * compbuf[ipoffV[ip]+ix];
```

```
            }
```

```
    MPI_Wait(&allg_handle, &allg_status);
```

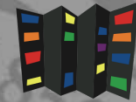
```
}}
```



- Basics
- Point-to-point communication
- Collective communication
- Grouping data for communication
- XSEDE monthly workshop – MPI programming next Tue, Wed

registration closes tomorrow 7am

http://www.chpc.utah.edu/short_courses/intro_mpi



<http://www-unix.mcs.anl.gov/mpi/>

Pacheco - Parallel Programming with MPI

Gropp, Lusk, Skjellum – Using MPI 1, 2

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

- No clear text passwords use ssh and scp
- You may not share your account under any circumstances
- Don't leave your terminal unattended while logged into your account
- Do not introduce classified or sensitive work onto CHPC systems
- Use a good password and protect it

- Do not try to break passwords, tamper with files etc.
- Do not distribute or copy privileged data or software
- Report suspicious to CHPC (security@chpc.utah.edu)
- Please see <http://www.chpc.utah.edu/docs/policies/security.html> for more details

- Debugging
 - Profiling
 - Mathematical Libraries at the CHPC
 - MPI-IO
 - Introduction to OpenMP
 - Hybrid MPI/OpenMP programming
- Survey

<https://www.surveymonkey.com/r/8GGBT78>