MPI Tutorial

Shao-Ching Huang

High Performance Computing Group
UCLA Institute for Digital Research and Education

Center for Vision, Cognition, Learning and Art, UCLA
July 15–22, 2013

A few words before we start

- You have got the class handout
- You can download the slides (see handout)
- You can download examples (see handout)
- Learn by going through simple examples

Think Parallel

Step 1: Find concurrency

- Decomposition
  - Task decomposition
    - example – compute one element of a vector
  - Data decomposition
    - example – split an image among processors
- Dependency Analysis
  - Group tasks
  - Order tasks
  - Data sharing

Step 2: Algorithm design

- Tasks
  - linear - task parallelizm
  - recursive - devide and conquer
- Data decomposition
  - geometric partition
    - domain decomposition
    - data exchange
  - tree data structure
    - “trees” and “forest”
- Data flow
  - pipeline
  - event-driven

Step 3: Program Structures

Commonly used patterns:

- SPMD (single program multiple data)
- Master-worker
- Loop parallelism
- Fork/join

Step 4: Implemtation

- Programming languages
  - Fortran (90)
  - C, C++
  - Python
  - others...
- Processing elements (PE) management
- Synchronization
- Communication

MPI Basics

What is MPI

- API for distributed memory parallel programming
- History
  - MPI 1.0 – 1994
  - MPI 2.0 – 1996
  - MPI 3.0 – 2012
- `http://www.mpi-forum.org`
- Fortran, C, C++, Python, ...
- Available on all serious parallel computers
- including your laptop

MPI program example (C)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank, nproc;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  printf("MPI process %d of %d\n",rank,nproc);
  MPI_Finalize();
}
```

How to run an MPI program

1. Install a MPI library (an "implementation"), e.g.
   - `http://www.mpich.org`
   - `http://www.open-mpi.org`
2. Use MPI compiler wrappers

```
$ mpicc hello.c
```

3. Run

```
$ mpiexec -n 4 ./a.out
MPI process 0 of 4
MPI process 1 of 4
MPI process 2 of 4
MPI process 3 of 4
```

MPI Execution Model

SPMD (single program multiple data)

- One executable
- Each process runs the same copy of executable
- Each copy of executable may do different things on different data (controlled by the program's logic)
- More examples later

Note: It is also possible to do MPMD with MPI...

Install MPI on your computer

1. Download

```
$ wget http://www.mpich.org/static/
downloads/3.0.4/mpich-3.0.4.tar.gz
```

2. Configure

```
$ configure --prefix=/path/to/mpi ...
```

3. Build & Install:

```
$ make
$ make install
```

4. Set up environment variable (also `.bashrc`)

```
$ export PATH=/path/to/mpi/bin:$PATH
```

MPI compiler wrappers ("MPI compilers")

Compiler wrappers to link MPI libraries

- mpicc (C)
- mpicxx (C++)
- mpif77 (F77)
- mpif90 (F90)

```
$ mpicxx -show
g++ -I/home/schuang/sw/mpich/include \
     -L/home/schuang/sw/mpich/lib -lmpichcxx\
     -Wl,-rpath -Wl,/home/schuang/sw/mpich/lib\
     -lmpich -lopa -lmpl -lrt -lpthread
```

Standard vs Implementations

- MPI Standard and MPI implementations (the software you download) are two different things
- Each implementation (such as MPICH, OpenMPI, ...) tries to comply to the standard
- Each implemtation have their own additional features
- The latest MPI standard is 3.0. Some implementions only support MPI 2.x.
- Read the fine manuals!

Terminology

In order to discuss MPI programming, we need to introduce a few MPI-specific terms

- Communicator
- Rank

and basic routines

- Environment query
- Send and Receive

Communicator

- Define the collection of MPI processes
- Many MPI routines require specifying a communicator
- Default communicator: `MPI_COMM_WORLD` (containing all processes)
- Using the default is enough for many cases

Rank

- Within a communicator of N processes, each process is numbered 0,1,. . . ,N-1.
- Which rank is this process:

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

- How many processes are in a communicator:

```
int nproc;
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

Run-time environment query

- Processor (host) name

```
int len;
char name[MPI_MAX_PROCESSOR_NAME];
MPI_Get_processor_name(name, &len);
```

- MPI API version:

```
int version, subversion;
MPI_Get_version(&version, &subversion);
```

- Elapsed wall-clock time in seconds

```
double t1,t2;
t1 = MPI_Wtime();
...
t2 = MPI_Wtime(); // elapsed time = t2-t1
```

Error handling

MPI routine returns an error code

```
int ierr;
ierr = MPI_Init(&argc, &argv);
if (rc != MPI_SUCCESS) {
    printf("program error. terminating...\n");
    MPI_Abort(MPI_COMM_WORLD, ierr);
}
```

MPI_Abort(): shut down all MPI processes

Sending data from A to B

- Sender (data source) calls MPI_Send
- Receiver (destination) calls MPI_Recv

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  MPI_Send(&msg,1,MPI_CHAR,dst,tag,
           MPI_COMM_WORLD);
} else if (rank == 1) {
  MPI_Recv(&msg,1,MPI_CHAR,src,tag,
           MPI_COMM_WORLD,&stat);
}
```

Sending data from A to B

`MPI_Send(buf,count,datatype,dest,tag,comm)`

- buf: starting address, pass by reference (pointer)
- count: number of data elements
- datatype: MPI data type
- dest: process rank of receiving process
- tag: non-positive integer assigned by programmer to uniquely identify message
- comm: communicator, e.g. `MPI_COMM_WORLD`

Sending data from A to B

`MPI_Recv(buf,count,type,dest,tag,comm,status)`

- buf: starting address, pass by reference (pointer)
- count: number of data elements
- type: MPI data type
- dest: process rank of receiving process
- tag: non-positive integer assigned by programmer to uniquely identify message
- comm: communicator, e.g. `MPI_COMM_WORLD`
- status: `MPI_Status` type, storing message info

MPI data types vs C types

There is a type for that:

- `MPI_INT`: int
- `MPI_FLOAT`: float
- `MPI_DOUBLE`: double
- `MPI_CHAR`: char
- `MPI_DOUBLE_COMPLEX`: double _Complex
- `MPI_INT8T`: int8_t
- `MPI_INT16T`: int16_t
- ...
- See MPI standard for a complete list.

Copy data from one process to all others

```
MPI_Bcast(buffer,count,datatype,root,comm)
```

- Broadcast from root (rank) to all other processes in the same communicator (comm).
- Use example: rank root (e.g. 0) reads input parameters, then broadcast to all other processes needing the parameters

Distribute data to other processes

```
MPI_Scatter(sendbuf,sendcnt,sendtype,
    recvbuf,recvcnt,recvtype,root,comm)
```

Distribute 5 doubles from rank=0 to all other processes in the communicator:

```
double x[100];  /* rank=0 process */
double y[5];    /* all processes */
MPI_Scatter(x,5,MPI_DOUBLE,y,5,MPI_DOUBLE,
            0, /* use rank=0 as root */
            MPI_COMM_WORLD);
```

Collect data from other processes

```
MPI_Gather(sendbuf,sendcnt,sendtype,
    recvbuf,recvcount,recvtype,root,comm)
```

Collect chunks of data onto root rank from all other processes in the communicator

```
double x[100];  /* rank=0 */
double y[5];    /* all other processes */
MPI_Gather(y,5,MPI_DOUBLE,x,5,MPI_DOUBLE,
           0,  /* collected onto rank=0 */
           MPI_COMM_WORLD);
```

Reduction operation

```
MPI_Reduce(sendbuf,recvbuf,count,
           type,op,root,comm)
```

- sendbuf: data source
- recvbuf: result
- count: number of data elements
- type: MPI data type
- op: operator
- root: process rank of result
- comm: communicator

MPI_Reduce examples

Find the maximum value of x among all processes and put the result in y on processor 0:

```
MPI_Reduce(&x,&y,1,MPI_DOUBLE,MPI_MAX,0,
           MPI_COMM_WORLD);
```

Sum over x among all processes and put the result in y on process 0:

```
MPI_Reduce(&x,&y,1,MPI_INT,MPI_SUM,0,
           MPI_COMM_WORLD);
```

- Other reduction operators: `MPI_PROD`, `MPI_MIN`, ...

Online resources

Web page for MPI routines
http:
//www.mcs.anl.gov/research/projects/mpi/www/www3

MPI Standards
http://www.mpi-forum.org/docs/docs.html

Examples

Vector inner product – sequential

Pseudo code

```
input: x[N], y[N]
output: s (inner product)

double sum = 0.0;
for (i=0; i<N; i++)
  sum += x[i]*y[i];
s = sum
```

See code browser

Vector inner product – MPI

Suppose we have np processors. Each process only stores (N/np) elements of the vector, and compute a local inner product.
Consider the data structure:

```
struct parVec {
    int    N;         /* global vector length */
    int    n;         /* local vector length */
    int    *offset;   /* starting index */
    double *data;     /* vector content */
};
```

See code browser.

Matrix-vector product – sequential

Pseudo code:

```
input: A(i,j), x(j), i=[1:M], j=[1:N]
output: y(i), i=[1:M]

for (i=0; i<M; i++) {
  sum = 0.0;
  for (j=0; j<N; j++) {
    sum += A(i,j)*x(j);
  }
  y(i) = sum
}
```

See code browser.

Matrix-vector product – MPI (Method 1)

- Partition **rows** of A (M-by-N); each process holds a M-by-(N/np) piece.
- Partition x; each process holds a (N/np) piece.
- Partition y; each process holds a (M/np) piece.

See the code browser.

Matrix-vector product – MPI (Method 2)

- Partition **columns** of A (M-by-N); each process holds a (M/np)-by-N piece.
- Partition x; each process holds a (N/np) piece.
- Partition y; each process holds a (M/np) piece.

See the code browser.

Parallel data structure

To store a "parallel" matrix, consider:

```
struct parMat {
    int    M,N;         /* global size */
    int    n;           /* local width */
    int    *col_offset; /* column offset */
    int    *row_offset; /* row offset */
    double *data;       /* matrix content */
};
...
n = N/np
col_offset[i] = i*(N/np)
row_offset[i] = i*(M/np)
```

See code browser.

Summary

- MPI is the most portable way to write distributed-memory programs for scientific computing
- Every process runs the same copy of executable
- Use rank to make different processes do different things
- Same variable may hold different values on different processes
- Communication done by MPI calls
- Define "parallel" data structures to keep program organized

Topics

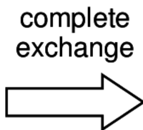
1. FFT
   - data transpose across processes
2. Stencil computations
   - data exchange across neighbor processes
   - ghost points

Parallel Fast Fourier Transform

Data transpose

- A general gather-scatter
- For process j
  - send i-th part to process i (scatter)
  - receive j-th part from other processes (gather)
- Schematic

A_0	A_1	A_2	A_3	A_4	A_5
B_0	B_1	B_2	B_3	B_4	B_5
C_0	C_1	C_2	C_3	C_4	C_5
D_0	D_1	D_2	D_3	D_4	D_5
E_0	E_1	E_2	E_3	E_4	E_5
F_0	F_1	F_2	F_3	F_4	F_5



A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_2
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	C_5	D_5	E_5	F_5

2D FFT

- 1D FFT (definition)

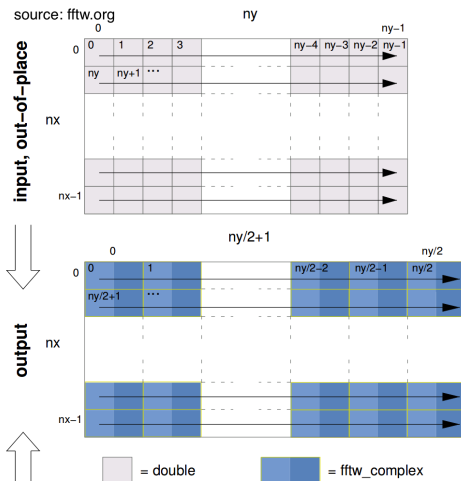
$$Y_k = \sum_{j=0}^{n-1} X_j e^{-2\pi jk\sqrt{-1}/n}$$

- `N`D FFT by applying 1D FFTs recursively
- Many applications
  - solving PDEs, image processing, ...
- FFTW library
  - `www.fftw.org`

2D sequential FFT

Real-to-complex FFT

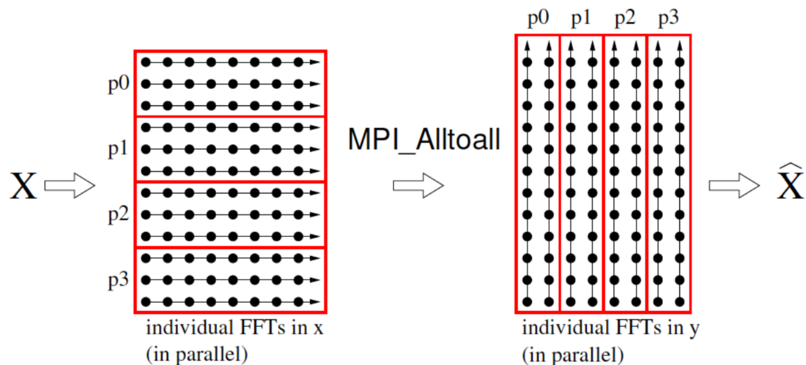
- input: M-by-N doubles (+ padding if in-place)
- output: (M/2+1)-by-N double complexes



2D parallel FFT

- Data partition
- Perform FFT dimension by dimension
- Input: M-by-(N/np) doubles
  - distributed across np processes
- FFT in M direction is the same as sequential version
  - speed up due to parallel execution
- Data transpose
  - transpose $*$-by-(N/np) into $*$-by-N layout
  - Use `MPI_Alltoallv` because of uneven distribution
- FFT in N direction

2D parallel FFT



- Before FFT(x): x-size is M (doubles)
- After FFT(x): x-size is (M/2+1) (complex doubles)
- Cannot partition evenly among processes
- Have to use `MPI_alltoallv`

MPI_Alltoallv

```
MPI_Alltoallv(sendbuf, sendcnts, sdispls,
              sendtype, recvbuf, recvcnts,
              rdispls, recvtype, comm)
```

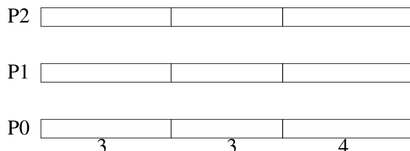
What to send

- `sendbuf`: starting address of data source
- `sendcnts`: number of elements (integer array)
- `sdispls`: send displacement (integer array)

Where to store received data

- `recvbuf`: starting address to receive
- `recvcnts`: number of received elements (integer array)
- `rdispls`: receive displacement (integer array)

MPI_Alltoallv example



Proces 0:

```
sendcnts = {3,3,4}; sdispls = {0,3,6}
recvcnts = {3,3,3}; rdispls = {0,3,6}
```

Proces 1:

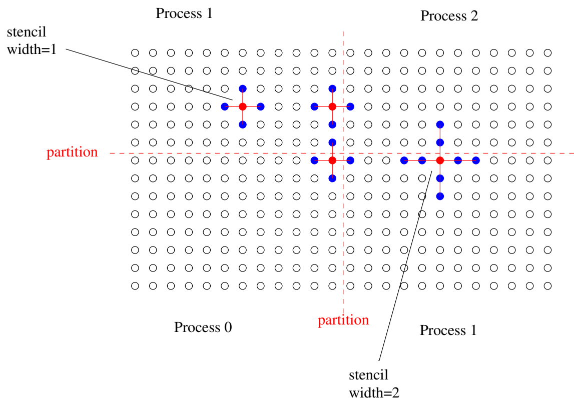
```
sendcnts = {3,3,4}; sdispls = {0,3,6}
recvcnts = {3,3,3}; rdispls = {0,3,6}
```

Proces 2:

```
sendcnts = {3,3,4}; sdispls = {0,3,6}
recvcnts = {4,4,4}; rdispls = {0,4,8}
```

Ghost Points

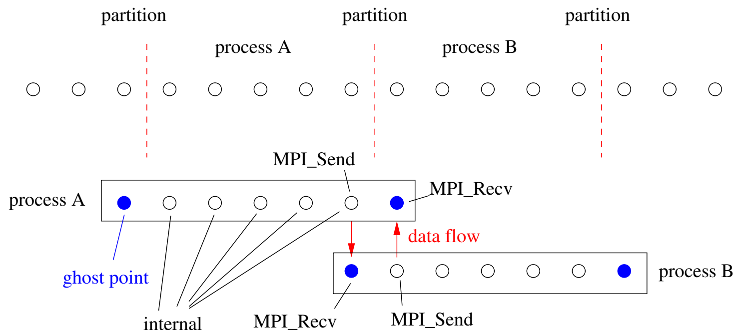
Neighbor points



- Operation on a point involves close neighbor points
- Applications: finite difference methods, image processing...

Ghost points

- Extra storage to hold copies of values from adjancent processes
- 1D example



Ghost point update

- Send to left, receive from right

```
myleft = rank-1
myright = rank+1
MPI_Send(...,dest = myleft,...)
MPI_Recv(...,src = myright,...)
```

- Send to right, receive from left

```
myleft = rank-1
myright = rank+1
MPI_Send(...,dest = myright,...)
MPI_Recv(...,src = myleft,...)
```

- Handle end points

```
if (myleft==0) myleft = MPI_PROC_NULL
if (myright==np-1) myright = MPI_PROC_NULL
```

2D ghost point update

- data order in memory
- design data structures to keep data exchange code clean