Using MPI

1. Scatter and Gather
   - parallel summation algorithm
   - collective communication: `MPI_Scatter` and `MPI_Gather`

2. Send and Recv
   - squaring numbers in an array
   - point-to-point communication: `MPI_Send` and `MPI_Recv`

3. Reducing the Communication Cost
   - measuring wall time with `MPI_Wtime`
   - sequential versus fan out

4. MPI for Python
   - point-to-point communication and numpy arrays

MCS 572 Lecture 5
Introduction to Supercomputing
Jan Verschelde, 31 August 2016
Using MPI

1. **Scatter and Gather**
   - parallel summation algorithm
   - collective communication: `MPI_Scatter` and `MPI_Gather`

2. **Send and Recv**
   - squaring numbers in an array
   - point-to-point communication: `MPI_Send` and `MPI_Recv`

3. **Reducing the Communication Cost**
   - measuring wall time with `MPI_Wtime`
   - sequential versus fan out

4. **MPI for Python**
   - point-to-point communication and numpy arrays
Consider the addition of 100 numbers on a distributed memory 4-processor computer.

For simplicity of coding: \[ S = \sum_{i=1}^{100} i. \]

Parallel algorithm to sum 100 numbers:

1. Distribute 100 numbers evenly among the 4 processors.
2. Every processor sums 25 numbers.
3. Collect the 4 sums to the manager node.
4. Add the 4 sums and print the result.
scattering data

Scattering an array of 100 numbers over 4 processors:

![Diagram showing data scattering to processors]

- **Data**: Array of 100 numbers.
- **Processors**: $p_0$, $p_1$, $p_2$, $p_3$.

The diagram illustrates how the data is scattered across the processors.
gathering results

Gathering the partial sums at the 4 processors to the root:
Using MPI

1. Scatter and Gather
   - parallel summation algorithm
   - collective communication: `MPI_Scatter` and `MPI_Gather`

2. Send and Recv
   - squaring numbers in an array
   - point-to-point communication: `MPI_Send` and `MPI_Recv`

3. Reducing the Communication Cost
   - measuring wall time with `MPI_Wtime`
   - sequential versus fan out

4. MPI for Python
   - point-to-point communication and numpy arrays
Scatter data with **MPI_Scatter**

To scatter data from one member to all members of a group:

\[
\text{MPI\_SCATTER}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \\
\quad \text{recvbuf}, \text{recvcount}, \text{recvtype}, \text{root}, \text{comm})
\]

where the arguments are

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>address of send buffer</td>
</tr>
<tr>
<td>sendcount</td>
<td>number of elements sent to each process</td>
</tr>
<tr>
<td>sendtype</td>
<td>data type of send buffer elements</td>
</tr>
<tr>
<td>recvbuf</td>
<td>address of receive buffer</td>
</tr>
<tr>
<td>recvcount</td>
<td>number of elements in receive buffer</td>
</tr>
<tr>
<td>recvtype</td>
<td>data type of receive buffer elements</td>
</tr>
<tr>
<td>root</td>
<td>rank of sending process</td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
</tr>
</tbody>
</table>
Gathering data with **MPI_Gather**

To gather data from all members to one member in a group:

\[
\text{MPI\_GATHER}(\text{sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm})
\]

where the arguments are

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>starting address of send buffer</td>
</tr>
<tr>
<td>sendcount</td>
<td>number of elements in send buffer</td>
</tr>
<tr>
<td>sendtype</td>
<td>data buffer of send buffer elements</td>
</tr>
<tr>
<td>recvbuf</td>
<td>address of receive buffer</td>
</tr>
<tr>
<td>recvcount</td>
<td>number of elements for any single receive</td>
</tr>
<tr>
<td>recvtype</td>
<td>data type of receive buffer elements</td>
</tr>
<tr>
<td>root</td>
<td>rank of receiving process</td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
</tr>
</tbody>
</table>
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define v 1 /* verbose flag, output if 1,
    no output if 0 */

int main ( int argc, char *argv[] )
{
    int myid, j,*data,tosum[25],sums[4];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    }
if(myid==0) /* manager allocates and initializes the data */
{
    data = (int*)calloc(100,sizeof(int));
    for (j=0; j<100; j++) data[j] = j+1;
    if(v>0)
    {
        printf("The data to sum : ");
        for (j=0; j<100; j++) printf(" %d",data[j]);
    }
}

MPI_Scatter(data,25,MPI_INT,tosum,25,MPI_INT,0, MPI_COMM_WORLD);
if(v>0) /* after the scatter, every node has 25 numbers to sum */
{
    printf("Node %d has numbers to sum :",myid);
    for(j=0; j<25; j++) printf(" %d", tosum[j]);
    printf("\n");
}

sums[myid] = 0;
for(j=0; j<25; j++) sums[myid] += tosum[j];
if(v>0) printf("Node %d computes the sum %d\n", myid, sums[myid]);

MPI_Gather(&sums[myid],1,MPI_INT,sums,1,MPI_INT,0,
            MPI_COMM_WORLD);
if (myid==0) /* after the gather, 
sums contains the four sums */
{
    printf("The four sums : ");
    printf("%d", sums[0]);
    for(j=1; j<4; j++) printf(" + %d", sums[j]);
    for(j=1; j<4; j++) sums[0] += sums[j];
    printf(" = %d, which should be 5050.\n", 
           sums[0]);
}
MPI_Finalize();
return 0;
Using MPI

1. Scatter and Gather
   - parallel summation algorithm
   - collective communication: `MPI_Scatter` and `MPI_Gather`

2. Send and Recv
   - squaring numbers in an array
   - point-to-point communication: `MPI_Send` and `MPI_Recv`

3. Reducing the Communication Cost
   - measuring wall time with `MPI_Wtime`
   - sequential versus fan out

4. MPI for Python
   - point-to-point communication and numpy arrays
squaring numbers

Example of an input and output sequence:
  Input : 2, 4, 8, 16, ...
  Output : 4, 16, 64, 256, ...

Instead of squaring, we could apply a difficult function $y = f(x)$ to an array of values for $x$.

$ mpirun -np 4 /tmp/parallel_square$

The data to square : 2 4 8 16
Node 1 will square 4
Node 2 will square 8
Node 3 will square 16
The squared numbers : 4 16 64 256
$
a parallel squaring algorithm

To square $p$ numbers:

1. The manager sends $p - 1$ numbers $x_1, x_2, \ldots, x_{p-1}$ to workers.
   Every worker receives: the $i$-th worker receives $x_i$ in $f$.
   The manager copies $x_0$ to $f$: $f = x_0$.

2. Every node (manager and all workers) squares $f$.

3. Every worker sends $f$ to the manager.
   The manager receives $x_i$ from $i$-th worker, $i = 1, 2, \ldots, p - 1$.
   The manager copies $f$ to $x_0$: $x_0 = f$, and prints.
Using MPI

1. Scatter and Gather
   - parallel summation algorithm
   - collective communication: `MPI_Scatter` and `MPI_Gather`

2. Send and Recv
   - squaring numbers in an array
   - point-to-point communication: `MPI_Send` and `MPI_Recv`

3. Reducing the Communication Cost
   - measuring wall time with `MPI_Wtime`
   - sequential versus fan out

4. MPI for Python
   - point-to-point communication and numpy arrays
sending data with **MPI_Send**

The blocking send operation has the syntax:

\[
\text{MPI\_SEND} \left( \text{buf, count, datatype, dest, tag, comm} \right)
\]

where the arguments are

- **buf**: initial address of the send buffer
- **count**: number of elements in send buffer
- **datatype**: data type of each send buffer element
- **dest**: rank of destination
- **tag**: message tag
- **comm**: communication
receiving data with *MPI_Recv*

The syntax of the blocking receive operation:

```c
MPI_RECV(buf, count, datatype, source, tag, comm, status)
```

where the arguments are

- **buf**: initial address of the receive buffer
- **count**: number of elements in receive buffer
- **datatype**: data type of each receive buffer element
- **source**: rank of source
- **tag**: message tag
- **comm**: communication
- **status**: status object
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define v 1 /* verbose flag, output if 1, no output if 0 */
#define tag 100 /* tag for sending a number */

int main ( int argc, char *argv[] )
{
    int p,myid,i,f,*x;
    MPI_Status status;

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

first $p$ even numbers

if (myid == 0) /* the manager allocates and initializes $x$ */
{
    x = (int*)calloc(p, sizeof(int));
    x[0] = 2;
    for (i=1; i<p; i++) x[i] = 2*x[i-1];
    if (v>0)
    {
        printf("The data to square : ");
        for (i=0; i<p; i++)
            printf(" %d", x[i]); printf("\n");
    }
}
every `MPI_Send` is matched by a `MPI_Recv`.

```c
if (myid == 0) /* the manager copies x[0] to f */
{
    /* and sends the i-th element to the i-th processor */
    f = x[0];
    for (i=1; i<p; i++)
        MPI_Send(&x[i],1,MPI_INT,i,tag,
                 MPI_COMM_WORLD);
}
else /* every worker receives its f from root */
{
    MPI_Recv(&f,1,MPI_INT,0,tag,MPI_COMM_WORLD,
             &status);
    if (v>0)
        printf("Node %d will square %d\n",myid,f);
}
```
squaring and sending results to root

\[ f = f; \quad /* \text{every node does the squaring */} \]

if(myrank == 0) /* the manager receives \( f \) in \( x[i] \) from processor \( i \) */
  for(i=1; i<p; i++)
    MPI_Recv(&x[i],1,MPI_INT,i,tag,
             MPI_COMM_WORLD,&status);
else /* every worker sends \( f \) to the manager */
  MPI_Send(&f,1,MPI_INT,0,tag,MPI_COMM_WORLD);
if (myid == 0) /* the manager prints results */
{
    x[0] = f;
    printf("The squared numbers : ");
    for (i=0; i<p; i++) printf(" %d", x[i]);
    printf("\n");
}

MPI_Finalize();
return 0;
Using MPI

1. Scatter and Gather
   - parallel summation algorithm
   - collective communication: `MPI_Scatter` and `MPI_Gather`

2. Send and Recv
   - squaring numbers in an array
   - point-to-point communication: `MPI_Send` and `MPI_Recv`

3. Reducing the Communication Cost
   - measuring wall time with `MPI_Wtime`
   - sequential versus fan out

4. MPI for Python
   - point-to-point communication and numpy arrays
measuring wall time with MPI_Wtime

To measure the communication cost, we run our parallel program without any computations.

MPI_Wtime() returns a double containing the elapsed time in seconds since some arbitrary time in the past.

Example usage:

```c
double startwtime, endwtime, totalwtime;

startwtime = MPI_Wtime();
/* code to be timed */
endwtime = MPI_Wtime();

totalwtime = endwtime - startwtime;
```
Using MPI

1. Scatter and Gather
   - parallel summation algorithm
   - collective communication: `MPI_Scatter` and `MPI_Gather`

2. Send and Recv
   - squaring numbers in an array
   - point-to-point communication: `MPI_Send` and `MPI_Recv`

3. Reducing the Communication Cost
   - measuring wall time with `MPI_Wtime`
   - sequential versus fan out

4. MPI for Python
   - point-to-point communication and numpy arrays
Consider the broadcast of one item over 8 processors:

a sequential broadcast

a fan out broadcast
algorithm for fan out broadcast

Algorithm: at step $k$, $2^{k-1}$ processors have data, and execute:

for $j$ from 0 to $2^{k-1}$ do
  processor $j$ sends to processor $j + 2^{k-1}$;
  processor $j + 2^{k-1}$ receives from processor $j$.

The cost to broadcast of one item
  - is $O(p)$ for a sequential broadcast,
  - is $O(\log_2(p))$ for a fan out broadcast.

The cost to scatter $n$ items
  - is $O(p \times n/p)$ for a sequential broadcast,
  - is $O(\log_2(p) \times n/p)$ for a fan out broadcast.
Using MPI

1. Scatter and Gather
   - parallel summation algorithm
   - collective communication: `MPI_Scatter` and `MPI_Gather`

2. Send and Recv
   - squaring numbers in an array
   - point-to-point communication: `MPI_Send` and `MPI_Recv`

3. Reducing the Communication Cost
   - measuring wall time with `MPI_Wtime`
   - sequential versus fan out

4. MPI for Python
   - point-to-point communication and numpy arrays
send and receive

Process 0 sends \texttt{DATA} to process 1:

\begin{verbatim}
MPI.COMM_WORLD.send(DATA, dest=1, tag=2)
\end{verbatim}

Every \texttt{send} must have a matching \texttt{recv}.

For the script to continue, process 1 must do

\begin{verbatim}
DATA = MPI.COMM_WORLD.recv(source=0, tag=2)
\end{verbatim}

\texttt{mpi4py} uses \texttt{pickle} on Python objects.

The user can declare the MPI types explicitly.
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()

if(RANK == 0):
    DATA = {'a': 7, 'b': 3.14}
    COMM.send(DATA, dest=1, tag=11)
    print RANK, 'sends', DATA, 'to 1'
elif(RANK == 1):
    DATA = COMM.recv(source=0, tag=11)
    print RANK, 'received', DATA, 'from 0'

$ mpiexec -n 2 python mpi4py_point2point.py
0 sends {'a': 7, 'b': 3.14} to 1
1 received {'a': 7, 'b': 3.14} from 0
a parallel sum

To sum an array of numbers, we distribute the numbers among the processes who compute the sum of a slice. The sums of the slices are sent to process 0 who computes the total sum.

$ mpiexec -n 10 python mpi4py_parallel_sum.py

0 has data [0 1 2 3 4 5 6 7 8 9] sum = 45
2 has data [20 21 22 23 24 25 26 27 28 29] sum = 245
3 has data [30 31 32 33 34 35 36 37 38 39] sum = 345
4 has data [40 41 42 43 44 45 46 47 48 49] sum = 445
5 has data [50 51 52 53 54 55 56 57 58 59] sum = 545
1 has data [10 11 12 13 14 15 16 17 18 19] sum = 145
8 has data [80 81 82 83 84 85 86 87 88 89] sum = 845
9 has data [90 91 92 93 94 95 96 97 98 99] sum = 945
7 has data [70 71 72 73 74 75 76 77 78 79] sum = 745
6 has data [60 61 62 63 64 65 66 67 68 69] sum = 645

total sum = 4950
$
distributing slices

```
from mpi4py import MPI
import numpy as np

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()
SIZE = COMM.Get_size()
N = 10

if(RANK == 0):
    DATA = np.arange(N*SIZE, dtype='i')
    for i in range(1, SIZE):
        SLICE = DATA[i*N:(i+1)*N]
        COMM.Send([SLICE, MPI.INT], dest=i)
    MYDATA = DATA[0:N]
else:
    MYDATA = np.empty(N, dtype='i')
    COMM.Recv([MYDATA, MPI.INT], source=0)
```
collecting the sums of the slices

\[ S = \text{sum}(\text{MYDATA}) \]

\[ \text{print RANK}, \ 'has data', \ \text{MYDATA}, \ 'sum =', \ S \]

\[ \text{SUMS} = \text{np.zeros}(\text{SIZE}, \ \text{dtype}='i') \]

\[ \text{if}(\text{RANK} > 0) : \]
\[ \quad \text{COMM.send}(S, \ \text{dest}=0) \]
\[ \text{else} : \]
\[ \quad \text{SUMS}[0] = S \]
\[ \quad \text{for} \ i \ \text{in} \ \text{range}(1, \ \text{SIZE}) : \]
\[ \quad \quad \text{SUMS}[i] = \text{COMM.recv}(\text{source}=i) \]
\[ \quad \text{print 'total sum =', sum(SUMS)} \]

Observe the following:

- `COMM.send` and `COMM.recv` have no type declarations.
- `COMM.Send` and `COMM.Recv` have type declarations.
Summary + Exercises


Exercises:

1. Adjust the parallel summation to work for $p$ processors where the dimension $n$ of the array is a multiple of $p$.

2. Use C or Python to rewrite the program to sum 100 numbers using `MPI_Send` and `MPI_Recv` instead of `MPI_Scatter` and `MPI_Gather`.

3. Use C or Python to rewrite the program to square $p$ numbers using `MPI_Scatter` and `MPI_Gather`.

4. Show that a hypercube network topology has enough direct connections between processors for a fan out broadcast.