Distributed Memory Programming with MPI

Moreno Marzolla
Dip. di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna

moreno.marzolla@unibo.it
Copyright © 2013, 2014, Moreno Marzolla, Università di Bologna, Italy (http://www.moreno.marzolla.name/teaching/AA2014/)

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 License (CC-BY-SA). To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.
Credits

- Peter Pacheco, Dept. of Computer Science, University of San Francisco
  http://www.cs.usfca.edu/~peter/
- Mary Hall, School of Computing, University of Utah
  https://www.cs.utah.edu/~mhall/
- Blaise Barney, Message Passing Interface (MPI)
  https://computing.llnl.gov/tutorials/mpi/ (highly recommended!!)
Introduction
Message Passing and MPI

- **Message passing** is the principal alternative to shared memory parallel programming
  - Message passing represents the predominant programming model for supercomputers and clusters

- **What MPI is**
  - A library used within conventional sequential languages (Fortran, C, C++)
  - Based on Single Program, Multiple Data (SPMD)
  - Isolation of separate address spaces
    - no data races, but communication errors possible
    - exposes execution model and forces programmer to think about locality, both good for performance
    - Complexity and code growth!
Message Passing and MPI

- All communication and synchronization operations require subroutine calls
  - No shared variables
- Subroutines for
  - Communication
    - Pairwise or point-to-point
    - Collectives involving multiple processors
  - Synchronization
    - Barrier
    - No locks because there are no shared variables to protect
  - Queries
    - How many processes? Which one am I? Any messages waiting?
Using MPI under Debian/Ubuntu

- For Ubuntu 12.04/14.04 install the openmpi-bin and libopenmpi-dev packages
- Use mpicc to compile, mpirun to execute
  - see next slides
- To execute your program on remote hosts, make sure you can ssh into them without entering a password
  - Generate a public/private key pair on your local machine, if you have not already done so, with ssh-keygen -t dsa; do not enter a passphrase
  - Append your file .ssh/id_dsa.pub to the remote file .ssh/authorized_keys
  - The exact file names above may vary depending on your system and ssh version; check your documentation
Finding Out About the Environment

• Two important questions that arise early in a parallel program are:
  – How many processes are participating in this computation?
  – Which one am I?

• MPI provides functions to answer these questions:
  – MPI_Comm_size reports the number of processes.
  – MPI_Comm_rank reports the rank, a number between 0 and size-1, identifying the calling process
Hello, world!

```c
/* mpi_hello.c */

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv ); /* no MPI calls before this line */
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "Greetings from process %d of %d\n", rank, size );
    MPI_Finalize(); /* no MPI calls after this line */
    return 0;
}
```
Hello, world!

- Compilation:
  
  \texttt{mpicc -Wall mpi_hello.c -o mpi_hello}

- Execution (8 instances on localhost):
  
  \texttt{mpirun -n 8 ./mpi_hello}

- Execution (two instances on host “foo” and one on host “bar”)
  
  \texttt{mpirun -H foo,foo,bar ./mpi_hello}
Hello, world!

$ mpirun -n 8 ./mpi_hello
Greetings from process 7 of 8
Greetings from process 5 of 8
Greetings from process 0 of 8
Greetings from process 3 of 8
Greetings from process 6 of 8
Greetings from process 4 of 8
Greetings from process 1 of 8
Greetings from process 2 of 8
Hostfile

$ cat myhostfile
aa slots=4
bb slots=4
cc slots=4

- The following runs 4 instances on node “aa” and 2 instances on node “bb”

$ mpirun -hostfile myhostfile -n 6 ./mpi_hello

- The following runs 2 instances on “aa”, 2 on “bb” and the remaining 2 on “cc”

$ mpirun -loadbalance -hostfile myhostfile -n 6 ./mpi_hello

- `man mpirun` for additional information
/* mpi_point_to_point.c */
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[])
{
    int rank, buf;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* Process 0 sends and Process 1 receives */
    if (rank == 0) {
        buf = 123456;
        MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    else if (rank == 1) {
        MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                  &status );
        printf( "Received %d\n", buf );
    }

    MPI_Finalize();
    return 0;
}
MPI Basic (Blocking) Send

buf = 123456;
MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

- **MPI_Send**(start, count, datatype, dest, tag, comm)
  - The message buffer is described by (start, count, datatype).
  - count is the number of *items* to send (NOT the number of bytes)
  - The target process is specified by dest, which is the rank of the target process in the communicator specified by comm.
  - When this function returns, the data has been delivered to the system and the buffer can be reused. The message may have not been received by the target process.
MPI Basic (Blocking) Receive

- **MPIRecv**(start, count, datatype, source, tag, comm, status)
  - Waits until a matching (both source and tag) message is received from the system, and the buffer can be used
  - source is rank in communicator specified by comm, or MPI_ANY_SOURCE
  - tag is a tag to be matched on or MPI_ANY_TAG
  - receiving fewer than count occurrences of datatype is OK, but receiving more is an error
  - status contains further information (e.g. size of message)
Some Basic Clarifying Concepts

- **How to organize processes**
  - Processes can be collected into **groups**
  - A group and context together form a **communicator**
  - A process is identified by its **rank** in the group associated with a communicator

- **There is a default communicator whose group contains all initial processes, called**
  **MPI_COMM_WORLD**
MPI Datatypes

• The data in a message to send or receive is described by a triple \((\text{address}, \text{count}, \text{datatype})\), where

• An MPI datatype is recursively defined as:
  – predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  – a contiguous array of MPI datatypes
  – a strided block of datatypes
  – an indexed array of blocks of datatypes
  – an arbitrary structure of datatypes

• There are MPI functions to construct custom datatypes, in particular ones for subarrays
## MPI datatypes

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>float</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying `MPI_ANY_TAG` as the tag in a receive.
Example: Trapezoid rule with MPI
Recall the trapezoid rule

/* Serial trapezoid rule */
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
x_i = a + h;
for (i=1; i<n; i++) {
approx += f(x_i);
x_i += h;
}
return h*approx;

See trap_serial.c
Parallel pseudo-code (naïve)

1: Get a, b, n;
2: \( h = (b-a)/n; \)
3: \( \text{local}_n = n/\text{comm}_sz; \)
4: \( \text{local}_a = a + \text{my}_rank*\text{local}_n*h; \)
5: \( \text{local}_b = \text{local}_a + \text{local}_n*h; \)
6: \( \text{local}_integral = \text{Trap}(\text{local}_a, \text{local}_b, \text{local}_n, h); \)
7: \textbf{if} (\text{my}_rank \neq 0)
8: \quad \text{Send local}_integral \text{ to process 0;}
9: \textbf{else} /* my_rank == 0 */
10: \quad \text{total}_integral = \text{local}_integral;
11: \quad \textbf{for} (\text{proc} = 1; \text{proc} < \text{comm}_sz; \text{proc}++) \{ 
12: \quad \text{Receive local}_integral \text{ from proc;}
13: \quad \text{total}_integral += \text{local}_integral;
14: \quad \}
15: \}
16: \textbf{if} (\text{my}_rank == 0)
17: \quad \text{print result;
Parallel pseudo-code (naïve)

- See file mpi_trap0.c
Collective Communication
MPI_Reduce

MPI_Reduce(
    &partial_result, /* send buffer */
    &result,       /* receive buffer, used only at root */
    1,             /* number of items to send */
    MPI_DOUBLE,    /* data type of elements */
    MPI_SUM,       /* reduction operator */
    0,             /* rank of root (who performs reduction) */
    MPI_COMM_WORLD /* communicator */
);
Predefined reduction operators

<table>
<thead>
<tr>
<th>Operation Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical and</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical or</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive or</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise exclusive or</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location of maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and location of minimum</td>
</tr>
</tbody>
</table>
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 **wrong**, and the program will likely hang or crash

• The receive buffer argument is only used at the destination process (the one performing the reduction)
  – However, all of the processes still need to pass in an actual argument, even if it’s just NULL
Parallel trapezoid
(with reduction primitive)

• See file mpi_trap1.c
MPI_Bcast

Broadcasts a message to all other processes of that group

count = 1;
source = 1;
broadcast originates in task 1
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);

Source: https://computing.llnl.gov/tutorials/mpi/
MPI_Scatter

Sends data from one task to all other tasks in a group

```
sendcnt = 1;
recvcnt = 1;
src = 1;  
task 1 contains the message to be scattered

MPI_Scatter(sendbuf, sendcnt, MPI_INT,
            recvbuf, recvcnt, MPI_INT,
            src, MPI_COMM_WORLD);
```

Source: https://computing.llnl.gov/tutorials/mpi/
MPI_Gather

Gathers together values from a group of processes

sendcnt = 1;
recvcnt = 1;
src = 1;    messages will be gathered in task 1

MPI_Gather(sendbuf, sendcnt, MPI_INT,
            recvbuf, recvcnt, MPI_INT,
            src, MPI_COMM_WORLD);

<table>
<thead>
<tr>
<th>task 0</th>
<th>task 1</th>
<th>task 2</th>
<th>task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

sendbuf (before)

recvbuf (after)

Source: https://computing.llnl.gov/tutorials/mpi/
MPI_Allgather

Gathers together values from a group of processes and distributes to all

sendcnt = 1;
recvcnt = 1;

MPI_Allgather(sendbuf, sendcnt, MPI_INT, recvbuf, recvcnt, MPI_INT, MPI_COMM_WORLD);

<table>
<thead>
<tr>
<th>task 0</th>
<th>task 1</th>
<th>task 2</th>
<th>task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

sendbuf (before)

recvbuf (after)

Source: https://computing.llnl.gov/tutorials/mpi/
MPI_Reduce

Perform and associate reduction operation across all tasks in the group and place the result in one task

count = 1;
dest = 1;
result will be placed in task 1
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, 
dest, MPI_COMM_WORLD);

task 0  task 1  task 2  task 3
1        2        3        4  -> sendbuf (before)

10

recvbuf (after)
**MPI_Allreduce**

Perform and associate reduction operation across all tasks in the group and place the result in all tasks

```c
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
               MPI_COMM_WORLD);
```

<table>
<thead>
<tr>
<th>task 0</th>
<th>task 1</th>
<th>task 2</th>
<th>task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

→ sendbuf (before)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

← recvbuf (after)

Source: [https://computing.llnl.gov/tutorials/mpi/](https://computing.llnl.gov/tutorials/mpi/)
MPI_Allreduce

One possible implementation using tree-structured communication
MPI_Allreduce

Another implementation, using butterfly
MPI_Alltoall

Sends data from all to all processes. Each process performs a scatter operation.

```c
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT,
             recvbuf, recvcnt, MPI_INT,
             MPI_COMM_WORLD);
```

<table>
<thead>
<tr>
<th>task 0</th>
<th>task 1</th>
<th>task 2</th>
<th>task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

(sendbuf (before))

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

(recvbuf (after))

Source: https://computing.llnl.gov/tutorials/mpi/
MPI_Scan

Computes the scan (partial reductions) of data on a collection of processes

count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

<table>
<thead>
<tr>
<th>task 0</th>
<th>task 1</th>
<th>task 2</th>
<th>task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

sendbuf (before)

| 1 | 3 | 6 | 10 |

recvbuf (after)

Source: https://computing.llnl.gov/tutorials/mpi/
Collective Communication Routines / 1

**MPI_Barrier (comm)**
- Synchronization operation. Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call. Then all tasks are free to proceed.

**MPI_Bcast (&buffer, count, datatype, root, comm)**
- Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.

**MPI_Scatter (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)**
- Distributes distinct messages from a single source task to each task in the group

**MPI_Gather (&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype, root, comm)**
- Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter

**MPI_Allgather (&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)**
- Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group

**MPI_Reduce (&sendbuf, &recvbuf, count, datatype, op, root, comm)**
- Applies a reduction operation on all tasks in the group and places the result in one task
Collective Communication Routines / 2

MPI_Reduce_scatter (&sendbuf, &recvbuf, recvcount, datatype, op, comm)
- Collective computation operation + data movement. First does an element-wise reduction on a vector across all tasks in the group. Next, the result vector is split into disjoint segments and distributed across the tasks. This is equivalent to an MPI_Reduce followed by an MPI_Scatter operation.

MPI_Scan (&sendbuf, &recvbuf, count, datatype, op, comm)
- Performs a scan operation with respect to a reduction operation across a task group.

MPI_Alltoall (&sendbuf, sendcount, sendtype, &recvbuf, recvcnt, recvtype, comm)
- Data movement operation. Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.
Vector sum

\[ \mathbf{x} + \mathbf{y} = (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \]
\[ = (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \]
\[ = (z_0, z_1, \ldots, z_{n-1}) \]
\[ = \mathbf{z} \]

```c
void sum( double* x, double* y, double* z, int n )
{
    int i;
    for (i=0; i<n; i++) {
        z[i] = x[i] + y[i];
    }
}
```
Parallel Vector Sum

Processor 0  Processor 1  Processor 2  Processor 3

\[ x[] + + + + \]

\[ y[] = = = = \]

\[ z[] = = = = \]
Parallel Vector Sum

Processor 0
x[]
y[]

Processor 1
x[]
y[]
local_x[]
local_y[]
local_z[]

Processor 2
x[]
y[]
local_x[]
local_y[]
local_z[]

Processor 3
x[]
y[]
local_x[]
local_y[]
local_z[]

MPI_Scatter
MPI_Gather
Parallel Vector Sum

• See file mpi_vecsum.c
Requirement for MPI_Scatter

- Contiguous data
- Uniform message size
MPI_Scatterv / MPI_Gatherv

- Gaps are allowed between messages in source data
- Irregular message sizes are allowed
- Data can be distributed to processes in any order
MPI_Scatterv

**Function Description:**

MPI_Scatterv is a message passing interface (MPI) function used for scatter operations in parallel computing. It allows processes to scatter data from a single source to multiple destinations.

**Function Signature:**

```c
int MPI_Scatterv (void *sendbuf, int *sendcnts, int *displs, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- **sendbuf:** Pointer to the send buffer.
- **sendcnts:** Array of integers indicating the number of elements to be sent to each process.
- **displs:** Array of integers indicating the starting offset for each process.
- **sendtype:** The type of the data to be sent.
- **recvbuf:** Pointer to the receive buffer.
- **recvcnt:** Total number of elements to be received.
- **recvtype:** The type of the data to be received.
- **root:** The root process responsible for sending data.
- **comm:** The communicator representing the set of processes involved in the operation.

**Arguments:**

- `sendbuf`: The buffer containing the data to be scattered.
- `sendcnts`: An array specifying the number of elements each process will receive.
- `displs`: An array indicating the starting position for each element.
- `sendtype`: The type of the data being sent.
- `recvbuf`: The buffer where the scattered data will be placed.
- `recvcnt`: The total number of elements that will be scattered.
- `recvtype`: The type of the data being received.
- `root`: The process responsible for initiating the scatter operation.
- `comm`: The communicator involving the processes.

**Notes:**

- `sendcnts` specifies the number of elements to be sent to each process, not the size in bytes.
- `displs` indicates the starting position for each element in the send buffer.
- This function is used for efficient data distribution in parallel computing environments.
Timing MPI programs

double local_start, local_finish, local_elapsed, elapsed;

MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */

local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm);

if (my_rank == 0)
printf("Elapsed time = %e seconds\n", elapsed);
Odd-Even sort with MPI
Serial Odd-Even Transposition Sort

- Variant of bubble sort
- Compare all (even, odd) pairs of adjacent elements, and exchange them if in the wrong order
- Then compare all (odd, even) pairs, exchanging if necessary; repeat the step above

```
                  Compare and exchange
```

Time
Serial Odd-Even Transposition Sort

- Variant of bubble sort
- Compare all (even, odd) pairs of adjacent elements, and exchange them if in the wrong order
- Then compare all (odd, even) pairs, exchanging if necessary; repeat the step above

```c
for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0)
        for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) Swap(&a[i-1], &a[i]);
    else
        for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```
Choosing Granularity

- Communication in distributed memory systems is very expensive, therefore working on single array elements does not make any sense.
- We can achieve a coarser granularity by splitting the array in chunks and applying odd-even sort at the chunk level.
Odd-Even Sort

- The vector is split in chunks that are assigned to nodes
- Each node sorts its chunk (e.g., using qsort)
- At each exchange-swap step:
  - Node i sends a copy of its chunk to node i+1
  - Node i+1 sends a copy of its chunk to node i
  - Node i merges the chunks, discards upper half
  - Node i+1 merges the chunks, discards lower half
Communication pattern for odd-even sort

phase

0
Proc 0  Proc 1  Proc 2  Proc 3

1
Proc 0  Proc 1  Proc 2  Proc 3

2
Proc 0  Proc 1  Proc 2  Proc 3

3
Proc 0  Proc 1  Proc 2  Proc 3
Odd-Even Sort

10 7 12 33 9

7 9 10 12 33

sort

13 52 14 97 31

13 14 31 52 97

Send to neighbor

merge

7 9 10 12 33 13 14 31 52 97

7 9 10 12 33 13 14 31 52 97

merge

Discard upper half

7 9 10 12 13 14 31 33 52 97

7 9 10 12 13 14 31 33 52 97

Discard lower half

7 9 10 12 13

14 31 33 52 97
Beware of the (possible) deadlock!!

Sort local keys;

for (phase = 0; phase < comm_sz; phase++) {
    partner = Compute_partner(phase, my_rank);
    if (I'm not idle) {
        Send my keys to partner;
        Receive keys from partner;
        if (my_rank < partner)
            Keep smaller keys;
        else
            Keep larger keys;
    }
}
Solution 1 (bad): restructure communication

```c
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz, 0, comm, MPI_STATUS_IGNORE.

if (my_rank % 2 == 0) {
    MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
    MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz, 0, comm, MPI_STATUS_IGNORE.
} else {
    MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz, 0, comm, MPI_STATUS_IGNORE.
    MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
}
```
Solution 2 (better): use MPI_Sendrecv

- Carries out a blocking send and a receive in a single call
  - The dest and the source can be the same or different.
  - Especially useful because MPI schedules the communications so that the program won’t hang or crash.

```c
int MPI_Sendrecv(
    void* send_buf_p /* in */,
    int send_buf_size /* in */,
    MPI_Datatype send_buf_type /* in */,
    int dest /* in */,
    int send_tag /* in */,
    void* recv_buf_p /* out */,
    int recv_buf_size /* in */,
    MPI_Datatype recv_buf_type /* in */,
    int source /* in */,
    int recv_tag /* in */,
    MPI_Comm communicator /* in */,
    MPI_Status* status_p /* in */);
```
MPI Odd-Even transposition sort

- See file odd_even_mpi.c
Concluding Remarks

- MPI is a library of functions that can be called from C, C++, or Fortran programs.
- Many parallel programs use the single-program multiple data or SPMD approach.
- An MPI program is unsafe if its correct behavior depends on the fact that MPI_Send is buffering its input.