Point to Point (P2P) Communication

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

1.1 General Concepts

- Point to Point Communication Routines: General Concepts

How to Calculate PI?

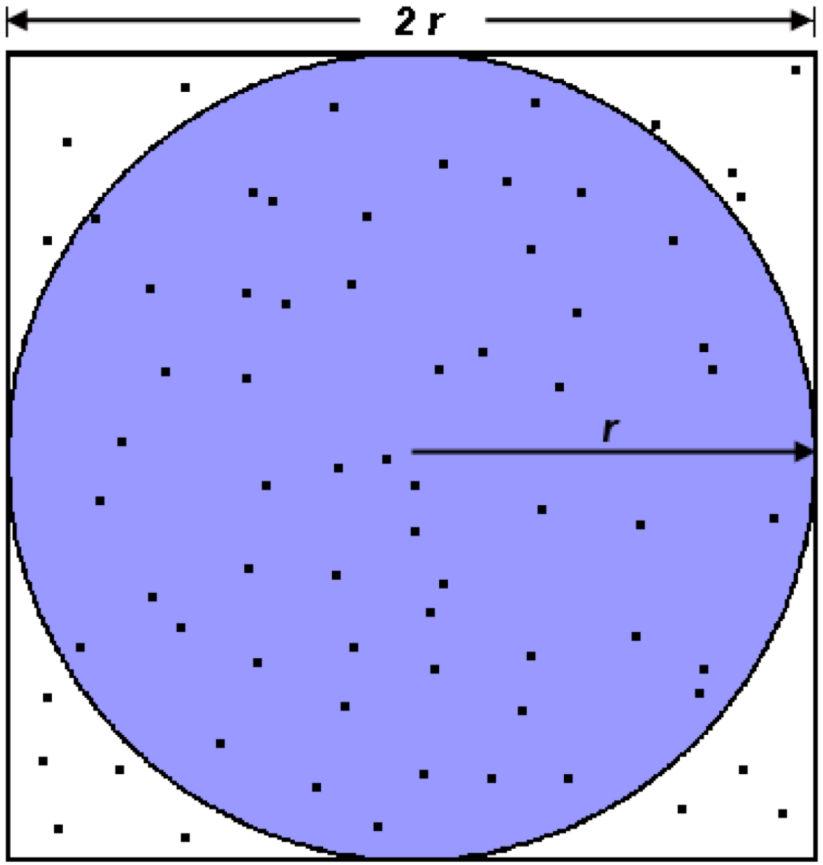
- The value of PI can be calculated in various ways. Consider the Monte Carlo method of approximating PI:
  - Inscribe a circle with radius r in a square with side length of $2r$
  - The area of the circle is πr^2 and the area of the square is $4r^2$
  - The ratio of the area of the circle to the area of the square is:

$$\frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

  - If you randomly generate N points inside the square, approximately $N * \frac{\pi}{4}$ of those points (M) should fall inside the circle.
  - π is then approximated as:

$$\pi = 4 * \frac{M}{N}$$

  - Note that increasing the number of points generated improves the approximation.



$A_S = (2r)^2 = 4r^2$

$A_C = \pi r^2$

$\pi = 4 \times \frac{A_C}{A_S}$

https://computing.llnl.gov/tutorials/mpi/#BuildScripts

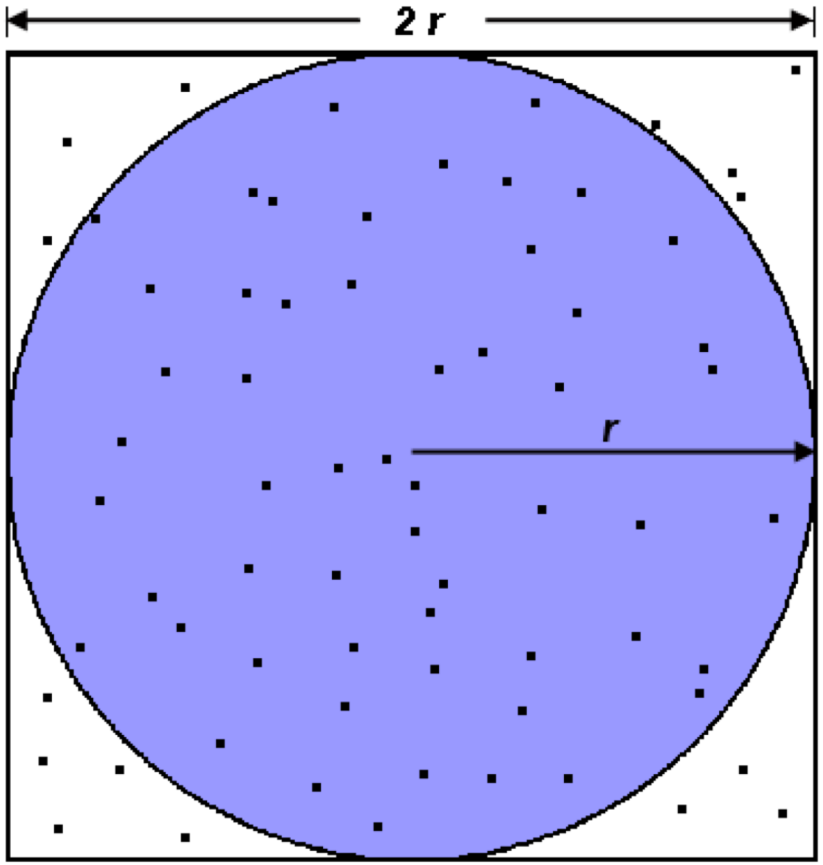
How to Calculate PI?

- Serial pseudo code for this procedure:

```
npoints = 10000
circle_count = 0

do j = 1,npoints
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```



$$A_S = (2r)^2 = 4r^2$$

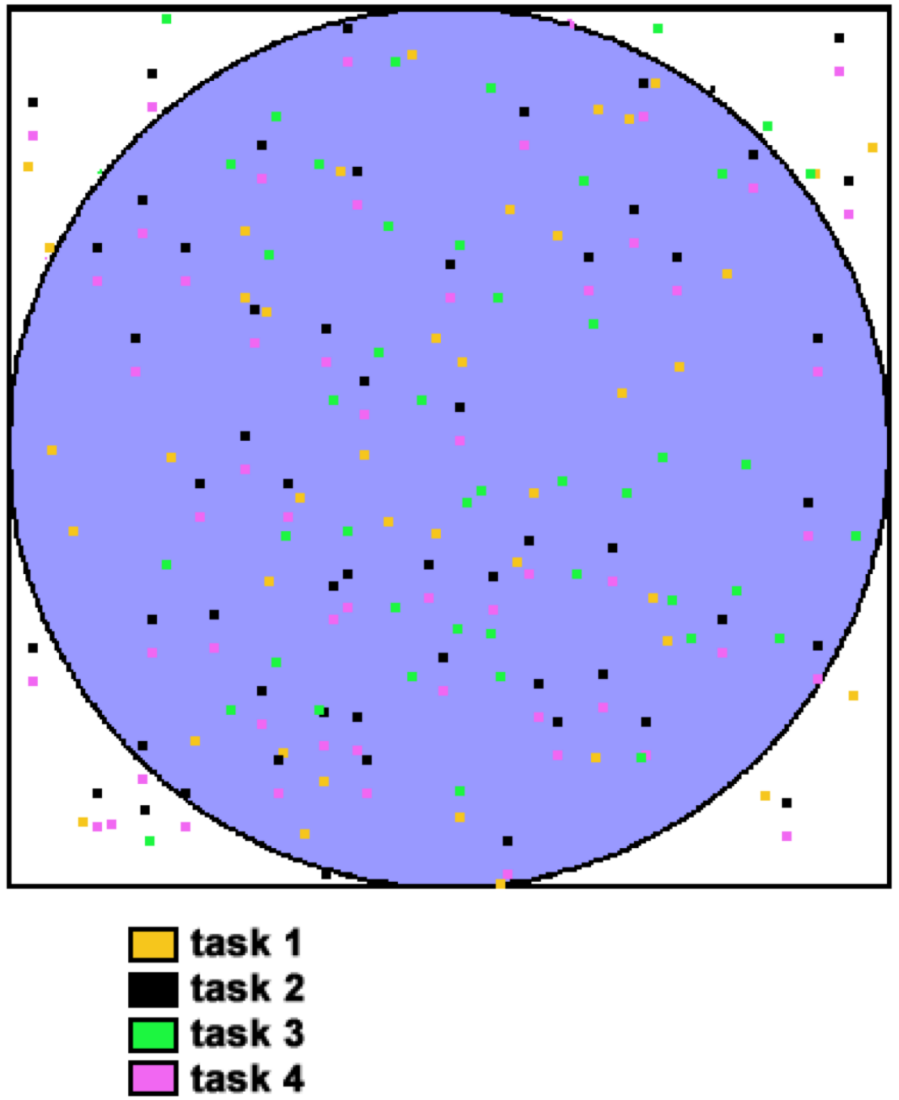
$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

https://computing.llnl.gov/tutorials/mpi/#BuildScripts

How to Calculate PI in Parallel?

- Leads to an "embarrassingly parallel" solution:
  - Break the loop iterations into chunks that can be executed by different tasks simultaneously.
  - Each task executes its portion of the loop a number of times.
  - Each task can do its work without requiring any information from the other tasks (there are no data dependencies).
  - Master task receives results from other tasks **using send/receive point-to-point operations**



https://computing.llnl.gov/tutorials/mpi/#BuildScripts

How to Calculate PI in Parallel?

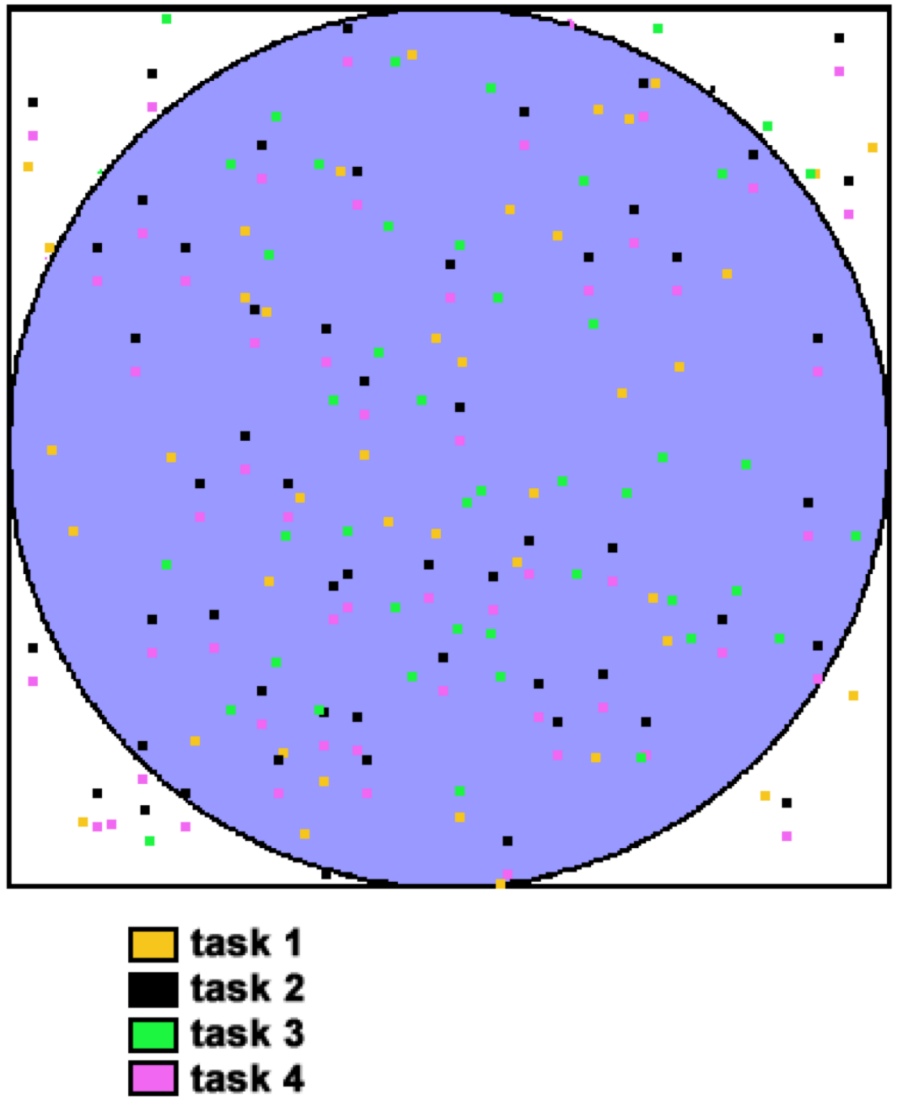
```
npoints = 10000
circle_count = 0

p = number of tasks
num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

if I am MASTER
  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)
else if I am WORKER
  send to MASTER circle_count
endif
```



https://computing.llnl.gov/tutorials/mpi/#BuildScripts

- **Key Concept:** Divide work between available tasks which communicate data via point-to-point message passing calls.

Types of Point-to-Point Operations

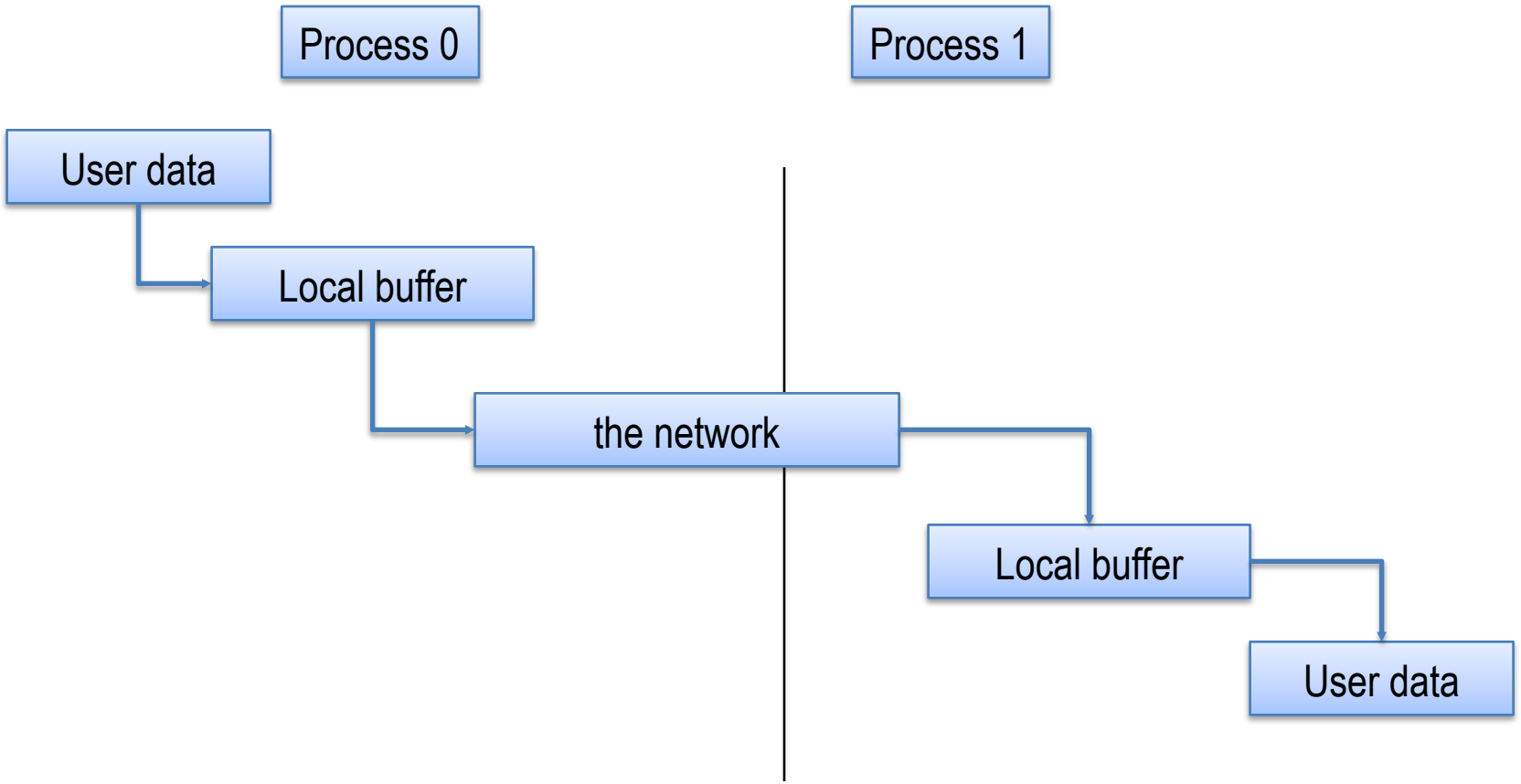
- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.
- There are different types of send and receive routines used for different purposes.
  - Synchronous send
  - Blocking send / blocking receive
  - Non-blocking send / non-blocking receive
  - Buffered send
  - Combined send/receive
  - “Ready” send

Buffering

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. Somehow or other, the MPI implementation must be able to deal with storing data when the two tasks are out of sync.
- Consider the following two cases:
  - A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
  - Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are “backing up”?
- The MPI implementation (not the MPI standard) decides what happens to data in these types of cases.

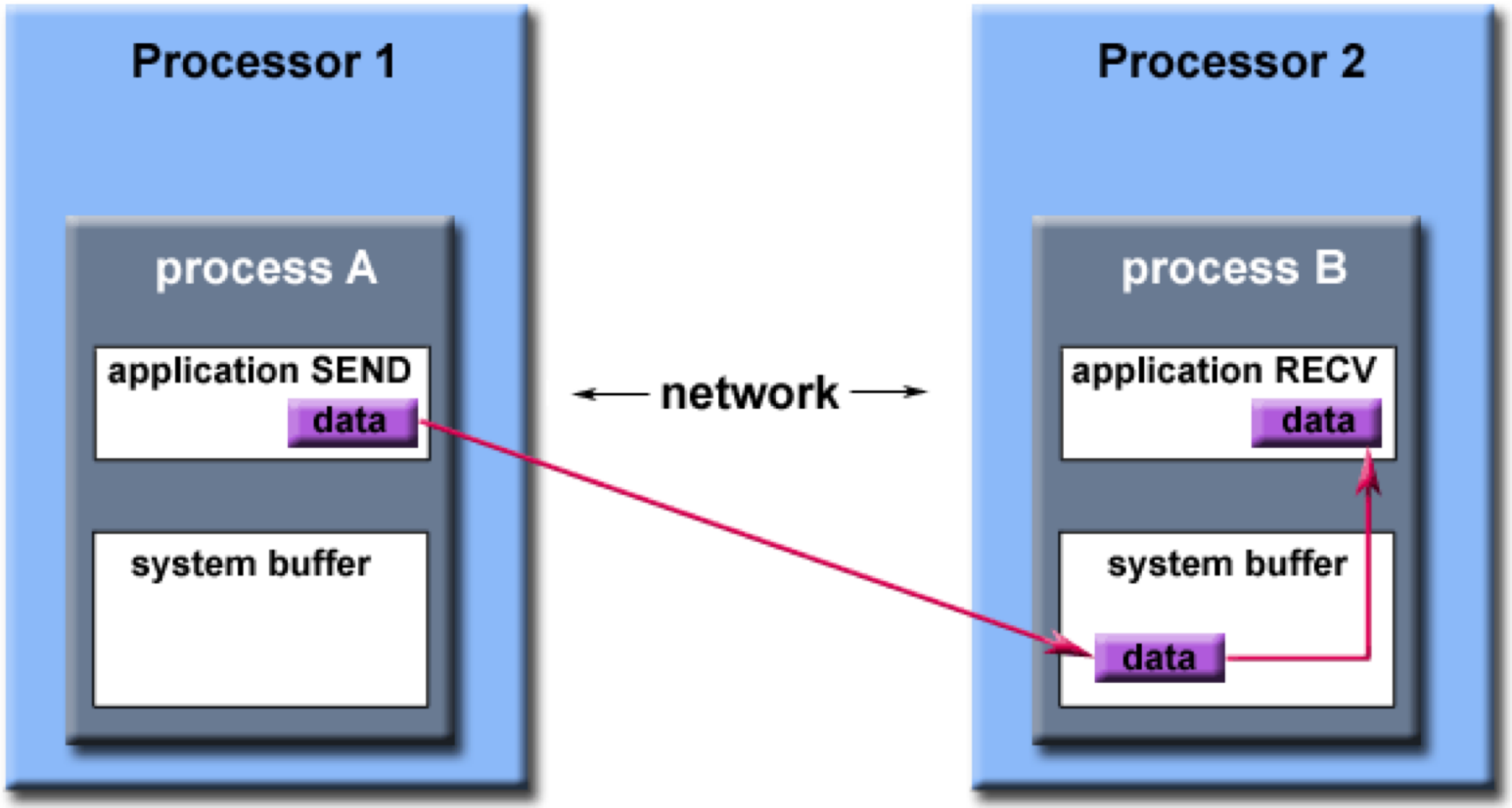
Buffers

- When you send data, where does it go? One possibility is:



Buffering

Typically, a system buffer area is reserved to hold data in transit. For example:



Path of a message buffered at the receiving process

Blocking vs. Non-blocking

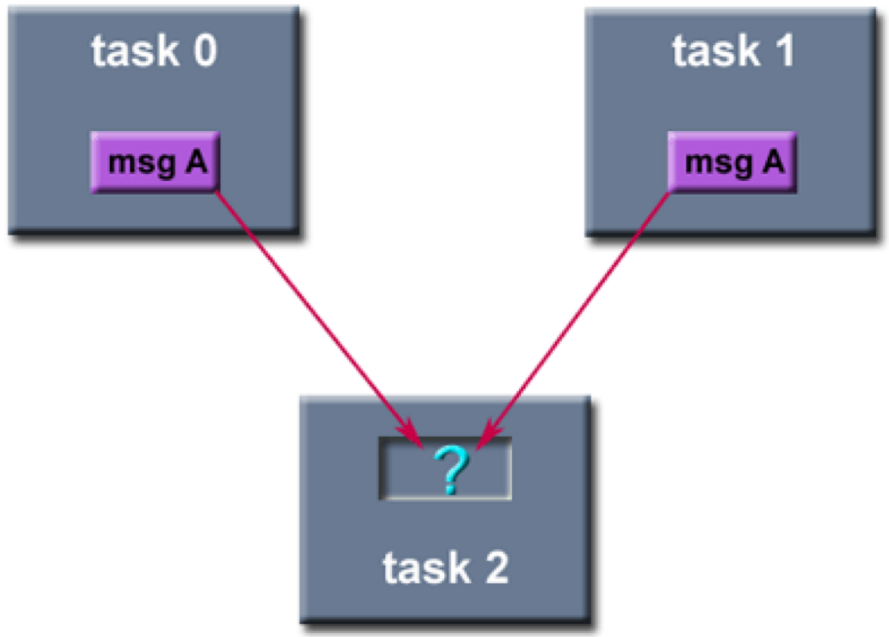
- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.
- Blocking:
  - A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
  - A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
  - A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program.

Blocking vs. Non-blocking

- Non-blocking:
  - Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
  - Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
  - It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
  - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

Fairness

- MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".
- Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.



1.2 MPI Message Passing Routine Arguments

- MPI Message Passing Routine Arguments

Point-to-point Operations

- The “Hello World” example did not contain any real communication.
- MPI Message: an array of elements of a particular MPI data type.
- MPI point-to-point communication routines generally have an argument list that takes one of the following formats:

Blocking sends	MPI_Send(buffer,count,type,dest,tag,comm)
Non-blocking sends	MPI_Isend(buffer,count,type,dest,tag,comm,request)
Blocking receive	MPI_Recv(buffer,count,type,source,tag,comm,status)
Non-blocking receive	MPI_Irecv(buffer,count,type,source,tag,comm,request)

Buffer

- Program (application) address space that references the data that is to be sent or received.
- In most cases, this is simply the variable name that is be sent/received.
- For C or C++ programs, this argument is passed by reference and usually must be prepended with an ampersand: **&var1**.

Data Count

- Indicates the number of data elements of a particular type to be sent.

MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype)

- 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

- MPI functions to construct custom datatypes,
  - in particular ones for subarrays

MPI Datatypes

C Data Types		Fortran Data Types	
MPI_CHAR	char	MPI_CHARACTER	character(1)
MPI_WCHAR	wchar_t - wide character		
MPI_SHORT	signed short int		
MPI_INT	signed int	MPI_INTEGER MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4	integer integer*1 integer*2 integer*4
MPI_LONG	signed long int		
MPI_LONG_LONG_INT MPI_LONG_LONG	signed long long int		
MPI_SIGNED_CHAR	signed char		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED_SHORT	unsigned short int		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long int		
MPI_UNSIGNED_LONG_LONG	unsigned long long int		
MPI_FLOAT	float	MPI_REAL MPI_REAL2 MPI_REAL4 MPI_REAL8	real real*2 real*4 real*8
MPI_DOUBLE	double	MPI_DOUBLE_PRECISION	double precision
MPI_LONG_DOUBLE	long double		
MPI_C_COMPLEX MPI_C_FLOAT_COMPLEX	float _Complex	MPI_COMPLEX	complex
MPI_C_DOUBLE_COMPLEX	double _Complex	MPI_DOUBLE_COMPLEX	double complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex		
MPI_C_BOOL	_Bool	MPI_LOGICAL	logical
MPI_INT8_T MPI_INT16_T MPI_INT32_T MPI_INT64_T	int8_t int16_t int32_t int64_t		
MPI_UINT8_T MPI_UINT16_T MPI_UINT32_T MPI_UINT64_T	uint8_t uint16_t uint32_t uint64_t		
MPI_BYTE	8 binary digits	MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack	MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

Destination

- An argument to send routines that indicates the process where a message should be delivered.
- Specified as the rank of the receiving process.

Source

- An argument to receive routines that indicates the originating process of the message.
- Specified as the rank of the sending process.
- This may be set to the wild card **`MPI_ANY_SOURCE (C++: MPI::ANY_SOURCE)`** to receive a message from any task.

Tag

- Arbitrary non-negative integer assigned by the programmer to uniquely identify a message.
- Send and receive operations should match message tags.
- For a receive operation, the wild card **`MPI_ANY_TAG (C++ MPI::ANY_TAG)`** can be used to receive any message regardless of its tag.
- The MPI standard guarantees that integers 0-32767 can be used as tags, but most implementations allow a much larger range than this.

Communicator

- Indicates the communication context, or set of processes for which the source or destination fields are valid.
- Unless the programmer is explicitly creating new communicators, the predefined communicator `MPI_COMM_WORLD (MPI::COMM_WORLD)` is usually used.

Status

- The source or tag of a received message may not be known if wildcard values were used in the receive operation.
- Also, if multiple requests are completed by a single MPI function , a distinct error code may need to be returned for each request.
- The information is returned by the status argument of MPI_RECV.
- The status argument also returns information on the length of the message received.
- However, this information is not directly available as a field of the status variable and a call to MPI_GET_COUNT is required to ``decode'' this information.

Request

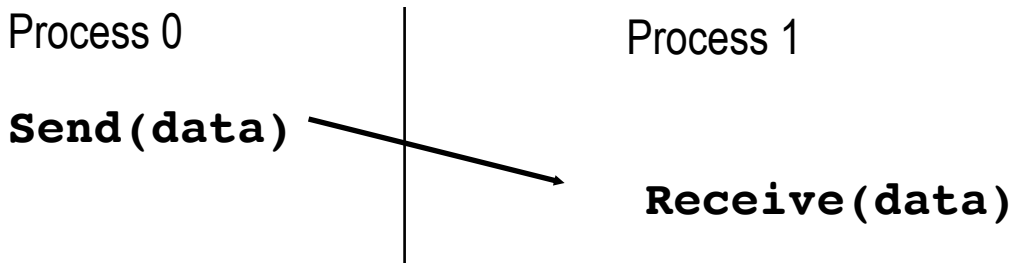
- Used by non-blocking send and receive operations.
- Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique “request number”.
- The programmer uses this system assigned “handle” later (in a WAIT type routine) to determine completion of the non-blocking operation.
- In C, this argument is a pointer to a predefined structure MPI_Request. In Fortran, it is an integer.

1.3 Blocking Message Passing Routines

- Blocking Message Passing Routines

MPI Basic Send/Receive

- We need to fill in the details in



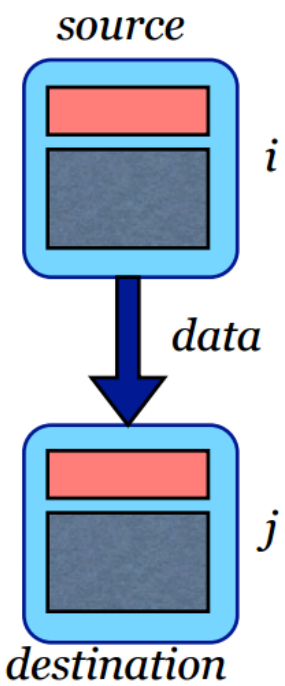
- Things that need specifying:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Blocking Point-to-point Operations

- Synchronous instructions to send a message from one *source* rank to a *destination* rank:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
     int  source, int tag, MPI_Comm comm, MPI_Status *status)
```




```
MPI_Send(&data, count, type, j, tag, comm);

MPI_Recv(&data, count, type, i, tag, comm, &status);
```

MPI Blocking Send

```
NAME
       MPI_Send -  Performs a basic send

SYNOPSIS
       #include "mpi.h"
       int MPI_Send( void *buf, int count, MPI_Datatype datatype, int
dest,
                     int tag, MPI_Comm comm )

INPUT PARAMETERS
       buf     - initial address of send buffer (choice)
       count   - number of elements in send buffer (nonnegative integer)
       datatype
               - datatype of each send buffer element (handle)
       dest    - rank of destination (integer)
       tag     - message tag (integer)
       comm    - communicator (handle)
```

MPI Blocking Recv

```
NAME
       MPI_Recv -  Basic receive

SYNOPSIS
       #include "mpi.h"
       int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source,
                     int tag, MPI_Comm comm, MPI_Status *status )

OUTPUT PARAMETERS
       buf    - initial address of receive buffer (choice)
       status - status object (Status)


INPUT PARAMETERS
       count  - maximum number of elements in receive buffer (integer)
       datatype
              - datatype of each receive buffer element (handle)
       source - rank of source (integer)
       tag    - message tag (integer)
       comm   - communicator (handle)
```

MPI Blocking Sendrecv

```
NAME
       MPI_Sendrecv -  Sends and receives a message

SYNOPSIS
       #include "mpi.h"
       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 )

INPUT PARAMETERS
       sendbuf
              - initial address of send buffer (choice)
       sendcount
              - number of elements in send buffer (integer)
       sendtype
              - type of elements in send buffer (handle)
       dest   - rank of destination (integer)
       sendtag
              - send tag (integer)
       recvcount
              - number of elements in receive buffer (integer)
       recvtype
              - type of elements in receive buffer (handle)
       source - rank of source (integer)
       recvtag
              - receive tag (integer)
       comm   - communicator (handle)
```

MPI_Ssend

- Synchronous blocking send
- Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.
- MPI_Ssend (&buf,count,datatype,dest,tag,comm)

Blocking operation

- So far we have been using *blocking* communication:
  - `MPI_Recv` does not complete until the buffer is full (available for use).
  - `MPI_Send` does not complete until the buffer is empty (available for use).
- Completion depends on size of message and amount of system buffering.

Lab: Simple Blocking P2P (mpi_p2p_block.cpp)

```cpp
#include <iostream>
#include <mpi.h>        // MPI header file
#define MASTER 0

using namespace std;

int main(int argc, char **argv) {
  int nprocs, rank;

  // initialize for MPI
  MPI_Init(&argc, &argv);
  // get number of processes
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  // get the rank = this process's number (ranges from 0 to nprocs - 1)
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // print a greeting
  cout << "Hello from process rank = " << rank << endl;

  int val = 0;
  // master send value to workers
  if (rank == MASTER) {
    val = 100;
    for (int dest_rank = 1; dest_rank < nprocs; dest_rank++) {
      MPI_Send(&val, 1, MPI_INT, dest_rank, 0, MPI_COMM_WORLD);
    }
    cout << "Process rank = " << rank << " has a value of " << val << endl;
  }
  // workers get the message
  else {
    MPI_Status status;
    MPI_Recv(&val, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    cout << "Process rank = " << rank << " has a value of " << val << endl;
    val = val + rank;
    cout << "Process rank = " << rank << " has a updated value of " << val << endl;
  }

  // clean up for MPI
  MPI_Finalize();   // C style

  return 0;
}
```

Lab: Simple Blocking P2P (mpi_p2p_block.cpp)

```cpp
#include <iostream>
#include <mpi.h>        // MPI header file
#define MASTER 0

using namespace std;

int main(int argc, char **argv) {
  int nprocs, rank;

  // initialize for MPI
  MPI_Init(&argc, &argv);
  // get number of processes
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  // get the rank = this process's number (
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // print a greeting
  cout << "Hello from process rank = " << r

  int val = 0;
  // master send value to workers
  if (rank == MASTER) {
    val = 100;
    for (int dest_rank = 1; dest_rank < npr
      MPI_Send(&val, 1, MPI_INT, dest_rank,
    }
    cout << "Process rank = " << rank << " (
  }
  // workers get the message
  else {
    MPI_Status status;
    MPI_Recv(&val, 1, MPI_INT, MPI_ANY_SOUR
    cout << "Process rank = " << rank << " has a value of " << val << endl;
    val = val + rank;
    cout << "Process rank = " << rank << " has a updated value of " << val << endl;
  }

  // clean up for MPI
  MPI_Finalize();   // C style

  return 0;
}
```

```
ai@ubuntu-20-04:~/Lab/MPI$ mpic++ mpi_p2p_block.cpp -o mpi_p2p_block
ai@ubuntu-20-04:~/Lab/MPI$ mpirun -np 4 ./mpi_p2p_block
Hello from process rank = 0
Hello from process rank = 1
Hello from process rank = 2
Hello from process rank = 3
Process rank = 1 has a value of 100
Process rank = 1 has a updated value of 101
Process rank = 2 has a value of 100
Process rank = 2 has a updated value of 102
Process rank = 0 has a value of 100
Process rank = 3 has a value of 100
Process rank = 3 has a updated value of 103
```

Status

- The source or tag of a received message may not be known if wildcard values were used in the receive operation.
- Also, if multiple requests are completed by a single MPI function , a distinct error code may need to be returned for each request.
- The information is returned by the status argument of `MPI_RECV`.
- As covered in the previous lesson, the `MPI_Recv` operation takes the address of an `MPI_Status` structure as an argument (which can be ignored with `MPI_STATUS_IGNORE`).

Status

- **The rank of the sender**. The rank of the sender is stored in the **`MPI_SOURCE`** element of the structure.
  - the rank can be accessed with **`status.MPI_SOURCE`**
- **The tag of the message**. The tag of the message can be accessed by the MPI_TAG element of the structure.
  - the tag can be accessed with **`status.MPI_TAG`**

Status Lab

- Open a terminal and copy the previous lab file (mpi_p2p_block.cpp) to a new file (mpi_p2p_block_status.cpp)
  - $ cp mpi_p2p_block.cpp mpi_p2p_block_status.cpp

Status Lab

- Edit the file as highlighted (Red box):

```
20  int val = 0;
21  // master send value to workers
22  if (rank == MASTER) {
23    val = 100;
24    for (int dest_rank = 1; dest_rank < nprocs; dest_rank++) {
25      int tag = dest_rank;
26      MPI_Send(&val, 1, MPI_INT, dest_rank, tag, MPI_COMM_WORLD);
27    }
28    cout << "Process rank = " << rank << " has a value of " << val << endl;
29  }
30  // workers get the message
31  else {
32    MPI_Status status;
33    MPI_Recv(&val, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
34    cout << "Process rank = " << rank << " has a value of " << val
35         << " from a source rank " << status.MPI_SOURCE
36         << " with a Tag ID " << status.MPI_TAG << endl;
37    val = val + rank;
38    cout << "Process rank = " << rank << " has a updated value of " << val << endl;
39  }
40
41  // clean up for MPI
42  MPI_Finalize();   // C style
```

Status Lab

- Run:

```
ai@ubuntu-20-04:~/Lab/MPI$ mpic++ mpi_p2p_block_status.cpp -o mpi_p2p_block_status
ai@ubuntu-20-04:~/Lab/MPI$ mpirun -np 4 ./mpi_p2p_block_status
Hello from process rank = 1
Hello from process rank = 2
Hello from process rank = 0
Hello from process rank = 3
Process rank = 1 has a value of 100 from a source rank 0 with a Tag ID 1
Process rank = 1 has a updated value of 101
Process rank = 2 has a value of 100 from a source rank 0 with a Tag ID 2
Process rank = 2 has a updated value of 102
Process rank = 3 has a value of 100 from a source rank 0 with a Tag ID 3
Process rank = 3 has a updated value of 103
Process rank = 0 has a value of 100
```

Array Lab

- Open a terminal and copy the previous lab file (mpi_p2p_block.cpp) to a new file (mpi_p2p_block_array.cpp)
  - $ cp mpi_p2p_block.cpp mpi_p2p_block_array.cpp

Array Lab

```
1 #include <iostream>
2 #include <mpi.h>        // MPI header file
3 #define MASTER 0
4 #define LEN_VAL 100
5
6 using namespace std;
7
8 int main(int argc, char **argv) {
9   int nprocs, rank;
10
11   // initialize for MPI
12   MPI_Init(&argc, &argv);
13   // get number of processes
14   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
15   // get the rank = this process's number (ranges from 0 to nprocs - 1)
16   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
17   // MPI status
18   MPI_Status status;
19
20   // consider chunk size by nprocs
21   int val[LEN_VAL] = {0};
22   int chunk_size = LEN_VAL / nprocs;    // assume no remainder
23   int offset = 0, partial_sum = 0, total_sum = 0;
24
25   // master
26   if (rank == MASTER)
27   {
28     // master initializes all elements of the array
29     for (int i=0; i<LEN_VAL; i++)
30       val[i] = i+1;
31
32     // master sends chunk of the array
33     for (int dest_rank = 1; dest_rank < nprocs; dest_rank++) {
34       offset = dest_rank * chunk_size;
35       MPI_Send(&val[offset], chunk_size, MPI_INT, dest_rank, 0, MPI_COMM_WORLD);
36     }
37
```

Array Lab

```
37
38      // sum of chunk elements on master process
39      cout << "Process rank = " << rank << " has a value of ";
40      for (int i=0; i < chunk_size; i++) {
41        cout << val[i] << " ";
42        total_sum += val[i];
43      }
44      cout << endl;
45
46      // master receives each partial sum from a worker and add it to total_sum
47      for (int src_rank = 1; src_rank < nprocs; src_rank++) {
48        MPI_Recv(&partial_sum, 1, MPI_INT, src_rank, 0, MPI_COMM_WORLD, &status);
49        total_sum += partial_sum;
50      }
51
52      // print out total sum
53      cout << "Total sum of the array elements is " << total_sum << endl;
54    }
55    // workers
56    else
57    {
58      // each worker receives the array chunk (it goes to index 0 ~ chink size)
59      MPI_Recv(&val, chunk_size, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
60      cout << "Process rank = " << rank << " has a value of ";
61
62      // calculate partial sum
63      for (int i=0; i<chunk_size; i++) {
64        cout << val[i] << " ";
65        partial_sum += val[i];
66      }
67      cout << endl;
68
69      // each worker send the partial sum to master
70      MPI_Send(&partial_sum, 1, MPI_INT, MASTER, 0, MPI_COMM_WORLD);
71    }
72
73    // clean up for MPI
74    MPI_Finalize();
75
76    return 0;
77  }
```

Array Lab

```
ai@ubuntu-20-04:~/Lab/MPI$ mpic++ mpi_p2p_block_array.cpp -o mpi_p2p_block_array
ai@ubuntu-20-04:~/Lab/MPI$ mpirun -np 4 ./mpi_p2p_block_array
Process rank = 0 has a value of 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Process rank = 1 has a value of 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
Process rank = 2 has a value of 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
Process rank = 3 has a value of 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Total sum of the array elements is 5050
```

Consider: Sum of the first N Integers

```
// serial solution
int main() {
  int N = 100000000, sum = 0;
  for(int i = 1; i<=N ; i++) {
    sum += i ;
  }
  cout << "The sum from 1 to << N << " is " << sum << endl;
}
```

- How to parallelize?

Develop your logic for Sum of the first N Integers in Parallel

- Each process (rank) do “partial sum” with
  - start_val = rank * (N / nprocs) + 1
  - end_val = (rank+1) * (N / nprocs)
  - E.g., if nprocs = 4, then
    - rank 0 sums from 1 to 25,000,000
    - rank 1 sums from 25,000,001 to 50,000,000
    - rank 2 sums from 50,000,001 to 75,000,000
    - rank 3 sums from 75,000,001 to 100,000,000
- Then, master (rank 0) receives each partial sum and add them.

Develop your logic for Sum of the first N Integers in Parallel

- Assume N is large and divisible by nproces without a remainder.
- Each process (rank) do “partial sum” with
  - start_val = rank * (N / nprocs) + 1
  - end_val = (rank+1) * (N / nprocs)
  - E.g., if nprocs = 4, then
    - rank 0 sums from 1 to 25,000,000
    - rank 1 sums from 25,000,001 to 50,000,000
    - rank 2 sums from 50,000,001 to 75,000,000
    - rank 3 sums from 75,000,001 to 100,000,000
- Then, master (rank 0) receives each partial sum and add them to print out total sum.
- Let me give you 15 mins to implement this program and let’s check together.