Parallel Programming with MPI

Based on the Peter Pacheto's presentation

Loic Guegan loic.guegan@uit.no

September 1, 2022

UiT The Arctic University of Norway

First glance of MPI

- Vector of numbers $X = [x_1, x_2, ..., x_n]$
- Heavy computations $f(x_i) \approx 1$ day
- Single machine:

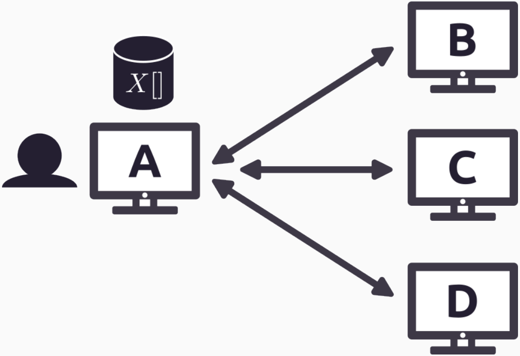
  $t = t_{f(x_1)} + ... + t_{f(x_n)} \approx n$ days

First glance of MPI

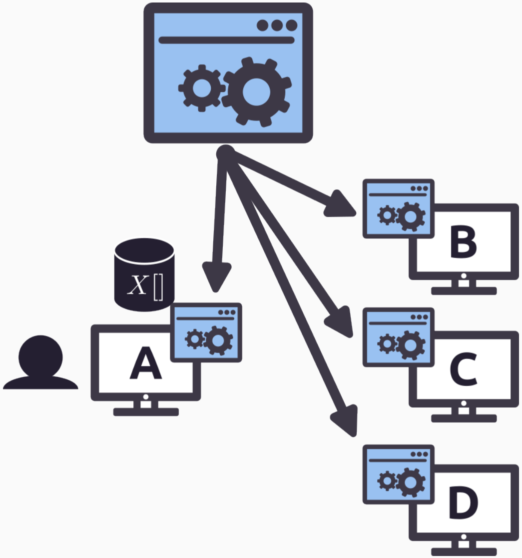
- Vector of numbers $X = [x_1, x_2, ..., x_n]$
- Heavy computations $f(x_i) \approx 1$ day
- Single machine:
  $t = t_{f(x_1)} + ... + t_{f(x_n)} \approx n$ days

Improve t using more machines?

First glance of MPI



First glance of MPI

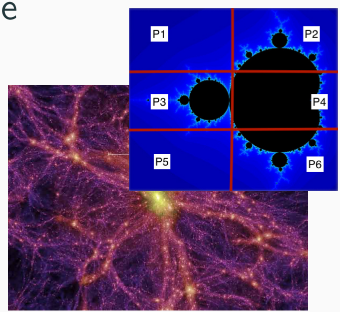


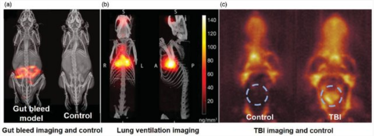
Outline

An introduction to MPI

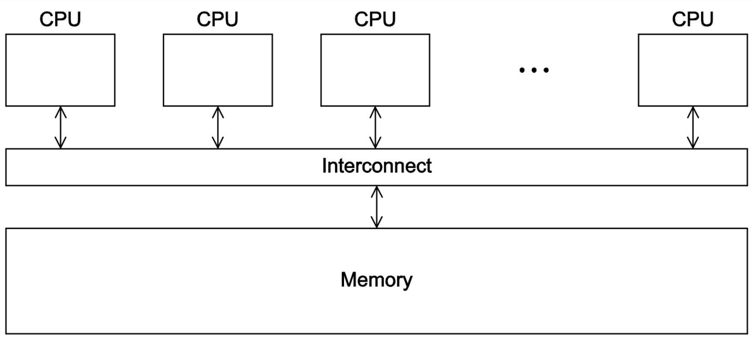
What is MPI?

- **M**essage **P**assing **I**nterface
- It **is a specification!**
  - MPICH
  - OpenMPI
  - and more!
- Parallel applications
  - Physics
  - Biology
  - Maths
  - Computer Science

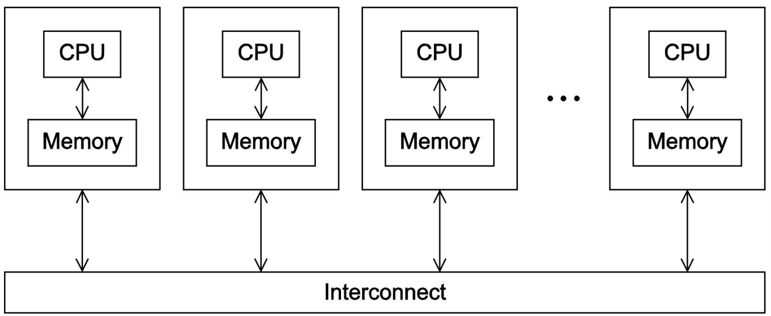




Shared Memory System

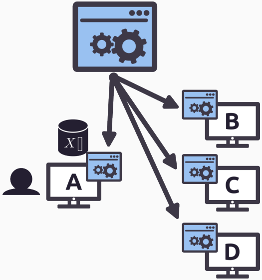


Distributed Memory System

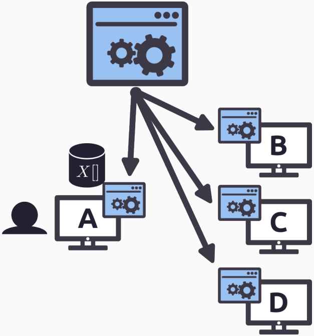


MPI and SPMD

- **S**ingle **P**rogram **M**ultiple **D**ata
- Compile **ONE** program



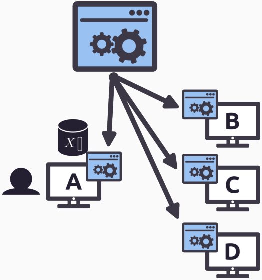
MPI and SPMD



- Each process does "something" different
- **Conditional branching** $\Longrightarrow$ SPMD

Identifying MPI Processes

- Common practice ⇒ Non-negative integers called **ranks**
- So for p processes we have $0, 1, \ldots, p-1$



Input/Output in MPI

Output

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

int main(void) {
   int my_rank, comm_sz;

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   printf("Proc %d of %d > Does anyone have a toothpick?\n",
         my_rank, comm_sz);

   MPI_Finalize();
   return 0;
}  /* main */
```

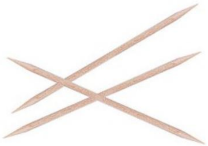
Each process just prints a message.

demo • mpi_dealing_with_io

Run with 6 processes

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
```

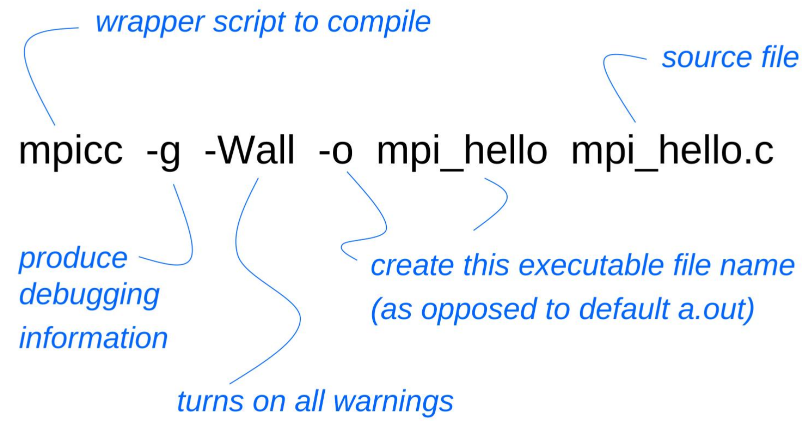
unpredictable output



Inputs

- Most MPI implementations ⇒ only process 0 in **MPI_COMM_WORLD** access to **stdin**
- Process 0:
    1. Read the data (scanf)
    2. Send the data to the other process

Compilation



Execution

mpiexec -n <number of processes> <executable>

---

mpiexec -n 1 ./mpi_hello

run with 1 process

mpiexec -n 4 ./mpi_hello

run with 4 processes

Execution

```
mpiexec -n 1 ./mpi_hello

    Greetings from process 0 of 1 !
```

```
mpiexec -n 4 ./mpi_hello

    Greetings from process 0 of 4 !
    Greetings from process 1 of 4 !
    Greetings from process 2 of 4 !
    Greetings from process 3 of 4 !
```

Recap

- Written in C
- Uses *stdio.h*, *string.h*, etc.
- Need to add **mpi.h** header file
- MPI identifiers start with **"MPI_"**
- First letter following underscore is uppercase
    - Function names and types
    - Avoid confusion

MPI Components

- MPI_Init
  - Tell MPI to setup

```
int MPI_Init(
        int*      argc_p  /* in/out */,
        char***   argv_p  /* in/out */);
```

- MPI_Finalize
  - Tell MPI to cleanup

```
int MPI_Finalize(void);
```

Basic Outline

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
   . . .
   /* No MPI calls before this */
   MPI_Init(&argc, &argv);
   . . .
   MPI_Finalize();
   /* No MPI calls after this */
   . . .
   return 0;
}
```

Point-to-point Communications

Communications: Communicators

- Communicators = **a reference to processes that can communicate together**
- MPI_Init create one for us!
- Called **MPI_COMM_WORLD**
- Contains in all the processes

Communications: Communicators

```
int MPI_Comm_size(
      MPI_Comm   comm        /* in   */,
      int*       comm_sz_p   /* out  */);
```

number of processes in the communicator

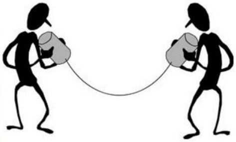
```
int MPI_Comm_rank(
      MPI_Comm   comm        /* in   */,
      int*       my_rank_p   /* out  */);
```

my rank
(the process making this call)

Communications: Send

```
int MPI_Send(

   void*          msg_buf_p     /* in */,
   int            msg_size      /* in */,
   MPI_Datatype   msg_type      /* in */,
   int            dest          /* in */,
   int            tag           /* in */,
   MPI_Comm       communicator  /* in */);
```

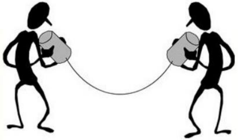


Communications: Datatypes

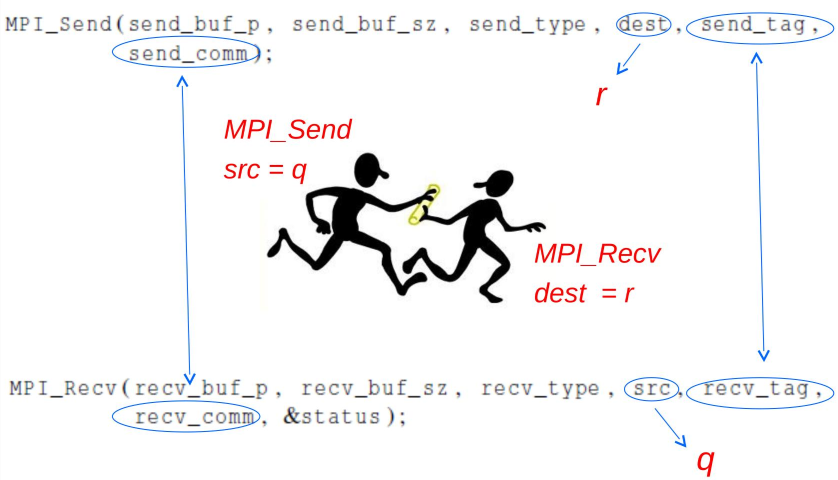
MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Communications: Receive

```
int MPI_Recv(
        void*          msg_buf_p     /* out */,
        int            buf_size      /* in  */,
        MPI_Datatype   buf_type      /* in  */,
        int            source        /* in  */,
        int            tag           /* in  */,
        MPI_Comm       communicator  /* in  */,
        MPI_Status*    status_p      /* out */);
```



Communications: Message Matching



Our first communications

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

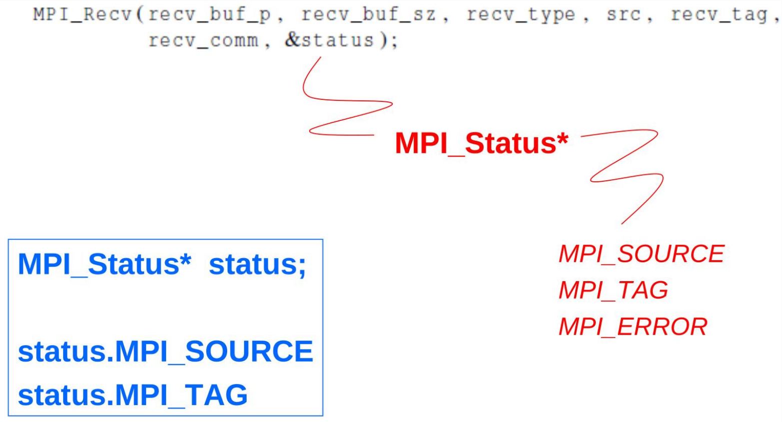
demo • mpi_first_com

Communications: Receiving

A receiver can receive a message **without** knowing:

- Message size
- The sender ⇒ MPI_ANY_SOURCE
- The tag ⇒ MPI_ANY_TAG

Communications: status_p argument



Communications: How much data?

```
int MPI_Get_count(
        MPI_Status*   status_p  /* in  */,
        MPI_Datatype  type      /* in  */,
        int*          count_p   /* out */);
```

Communications: Any issues?

MPI_Send and MPI_Recv:

- MPI_Recv always block
- MPI_Send behave differently according to buffer size
    - Cutoffs/Blocking
- Depends of the implementation!
- Solution $\Rightarrow$ **Know your implementation!**

Safety in MPI programs

Notion of Safety in MPI programs

MPI_Send behave in **2 ways**:

- **Buffering**: copy the data in the send buffer and return
- **Blocking**: block until a matching MPI_Recv call

Notion of Safety in MPI programs

A threshold is used to switch from buffering to blocking:

- Relatively small messages will be buffered by MPI_Send
- Larger messages will cause it to block

Notion of Safety in MPI programs

- If every processes do a MPI_Send ⇒ **program will hang or deadlock** since MPI_Recv not reached
- Each process is blocked waiting for an event that will never happen

Notion of Safety in MPI programs

- A program is **unsafe** if it relies on MPI buffering to work
- Works for various inputs
- Hang for others

How to check if a program is safe ?

- Use **MPI_Ssend** instead
- "s" $\equiv$ synchronous
- Block until a matching MPI_Recv

```
int MPI_Ssend(
      void*          msg_buf_p     /* in */,
      int            msg_size      /* in */,
      MPI_Datatype   msg_type      /* in */,
      int            dest          /* in */,
      int            tag           /* in */,
      MPI_Comm       communicator  /* in */);
```

How to make a program safe ?

```
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);
}
```

How to make a program safe ?

- Using **MPI_Sendrecv**
- Scheduling handled by MPI
- Blocking send + receive
- *dest* and *source* can be equal

How to make a program safe ?

```
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  */);
```

Collective Communications

Scenario

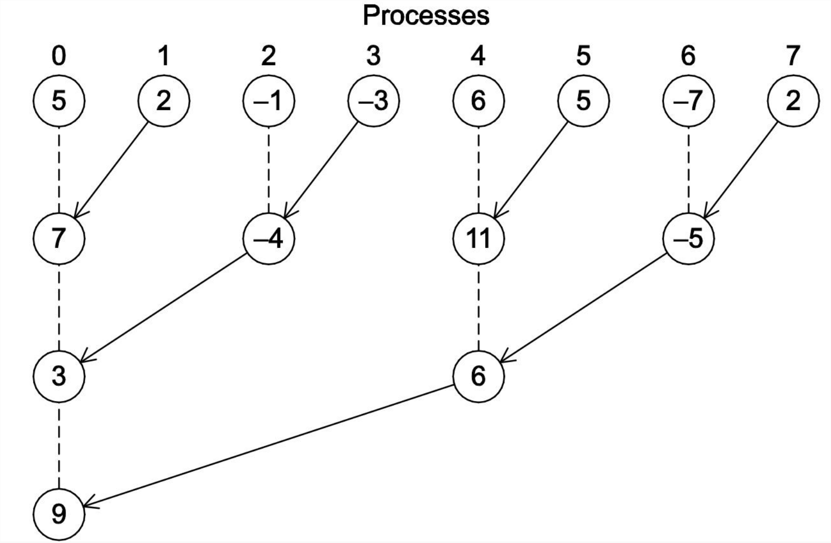
- 8 processes
- Each one hold a number
- How to perform a global sum?

Idea:

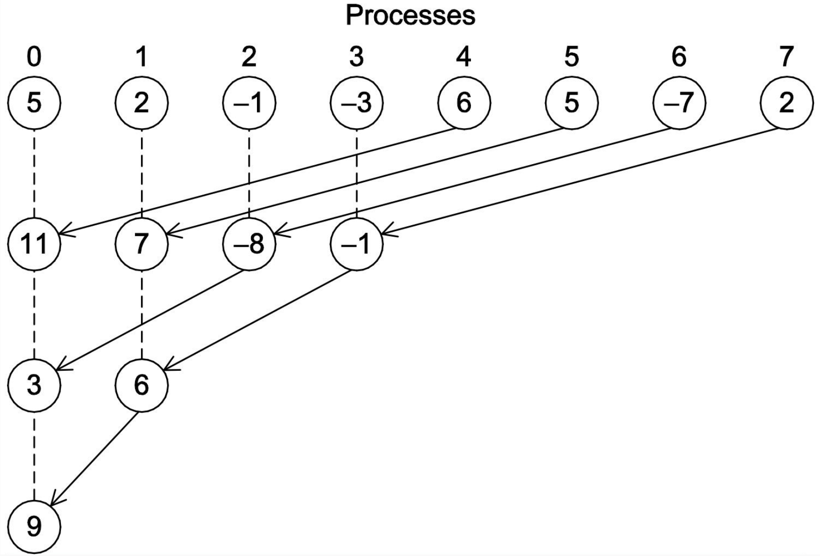
- Send all the numbers to process 0
- Perform the sum
- Print the result

Problem $\Longrightarrow$ **NOT FAIR/OPTIMIZED**

Tree-structured global sum



Tree-structured global sum alternative



Our first collective communication

```
int MPI_Reduce(
      void*          input_data_p   /* in  */,
      void*          output_data_p  /* out */,
      int            count          /* in  */,
      MPI_Datatype   datatype       /* in  */,
      MPI_Op         operator       /* in  */,
      int            dest_process   /* in  */,
      MPI_Comm       comm           /* in  */);
```

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

demo • mpi_reduce

Other reduction operators

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Collective vs Point-To-Point Communications

In collective communications:

- **All** processes **MUST** call the **SAME** collective function
- Example with 2 processes:
  - p1: MPI_Reduce()
  - p2: MPI_Recv()
  - Processes will **CRASH**, **HANG** or ...

Collective vs Point-To-Point Communications

In collective communications:

- All arguments must **BE COMPATIBLE**
- Example with 2 processes:
  - p1: MPI_Reduce() with dest_process=0
  - p2: MPI_Reduce() with dest_process=1
  - Processes will **CRASH**, **HANG** or ...

Collective vs Point-To-Point Communications

- **output_data_p** only used on **dest_process**
- All of the processes still need to pass in an actual argument corresponding to **output_data_p** even if NULL.

Collective vs Point-To-Point Communications

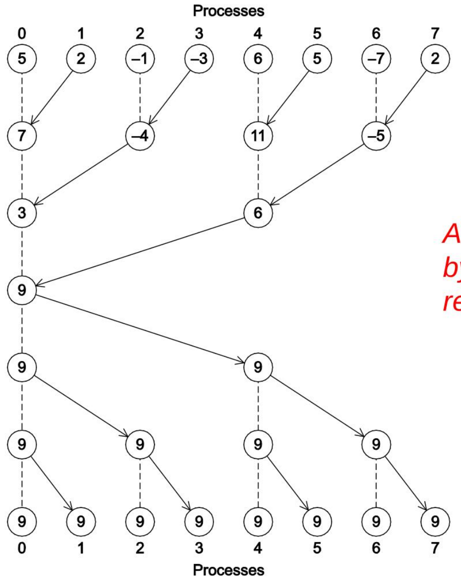
- P2P communications are matched using:
  - Communicators
  - Tags
- Collective communications = NO TAGS!
- Collective communications are matched using:
  - Communicators
  - Call order! $\Rightarrow$ All processes use the same collective calls order

MPI_Allreduce

- What if the result should be available to **all** the processes ?

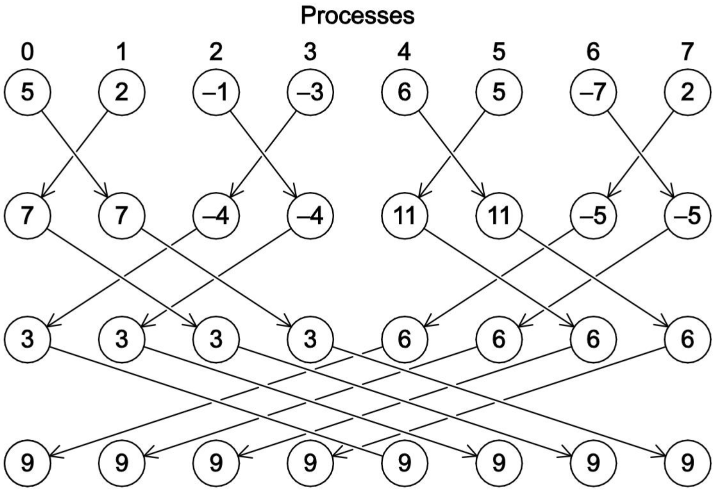
```
int MPI_Allreduce(
        void*          input_data_p    /* in  */,
        void*          output_data_p   /* out */,
        int            count           /* in  */,
        MPI_Datatype   datatype        /* in  */,
        MPI_Op         operator        /* in  */,
        MPI_Comm       comm            /* in  */);
```

MPI_Allreduce: Tree-structured



A global sum followed by distribution of the result.

MPI_Allreduce: Tree-structured



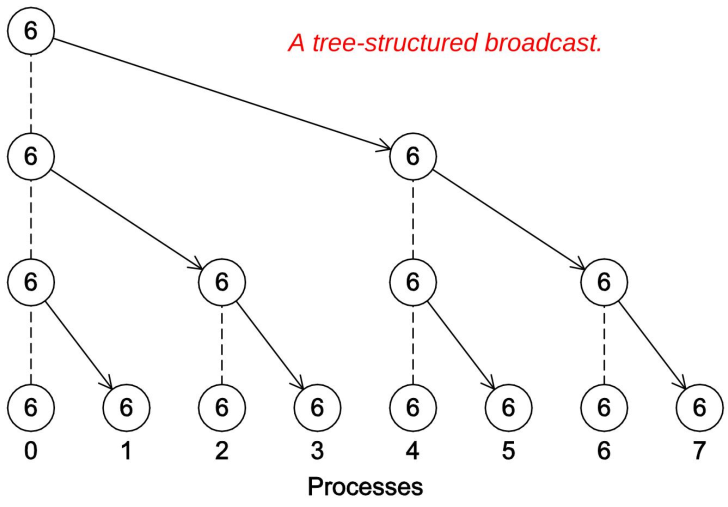
A butterfly-structured global sum.

Broadcast

```
int MPI_Bcast(
        void*          data_p        /* in/out */,
        int            count         /* in     */,
        MPI_Datatype   datatype      /* in     */,
        int            source_proc   /* in     */,
        MPI_Comm       comm          /* in     */);
```

- One process send its data to all the others in a communicator

Broadcast Tree-Structured



Data Distribution

$$
\begin{aligned}
\boldsymbol{x} + \boldsymbol{y} &= (x_0, x_1, ..., x_{n-1}) + (y_0, y_1, ..., y_{n-1}) \\
&= (x_0 + y_0, x_1 + y_1, ..., x_{n-1} + y_{n-1}) \\
&= (z_0, z, ..., z_{n-1}) \\
&= \boldsymbol{z}
\end{aligned}
$$

How to implement a parallel vector sum?

Data Distribution: Serial Vector Sum

A serial version:

```
void Vector_sum(double x[], double y[], double z[], int n) {
   int i;

   for (i = 0; i < n; i++)
      z[i] = x[i] + y[i];
}  /* Vector_sum */
```

Data Distribution

- Block partitionning
  - Assign blocks of consecutive components to each process
- Cyclic partitioning
  - Assign components in a round robin fashion
- Block-cyclic partitioning
  - Use a cyclic distribution of blocks of components

Data Distribution

Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Data Distribution: Parallel Vector Sum

```
void Parallel_vector_sum(
      double  local_x[]  /* in  */,
      double  local_y[]  /* in  */,
      double  local_z[]  /* out */,
      int     local_n    /* in  */) {
   int local_i;

   for (local_i = 0; local_i < local_n; local_i++)
      local_z[local_i] = local_x[local_i] + local_y[local_i];
}  /* Parallel_vector_sum */
```

Data Distribution: Scatter

MPI_Scatter allows to **read an entire vector on a process** and **send the required components to each process**

```
int MPI_Scatter(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            src_proc    /* in  */,
      MPI_Comm       comm        /* in  */);
```

Data Distribution: Parallel Vector Sum

```
void Read_vector(
      double     local_a[]   /* out */,
      int        local_n     /* in  */,
      int        n           /* in  */,
      char       vec_name[]  /* in  */,
      int        my_rank     /* in  */,
      MPI_Comm   comm        /* in  */) {

   double* a = NULL;
   int i;

   if (my_rank == 0) {
      a = malloc(n*sizeof(double));
      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]);
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
      free(a);
   } else {
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Read_vector */
```

demo • mpi_vector_addition

Gather

MPI_Gather allows **from one process** to **collect data of all the other processes**

```
int MPI_Gather(
      void*          send_buf_p   /* in   */,
      int            send_count   /* in   */,
      MPI_Datatype   send_type    /* in   */,
      void*          recv_buf_p   /* out  */,
      int            recv_count   /* in   */,
      MPI_Datatype   recv_type    /* in   */,
      int            dest_proc    /* in   */,
      MPI_Comm       comm         /* in   */);
```

Allgather

MPI_Allgather allows to **collect data from all the processes** on **all the processes**

```
int MPI_Allgather(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      MPI_Comm       comm         /* in  */);
```

Allgather

- Concatenates the content of **send_buf_p** of each process and stores it in each process **recv_buf_p**
- **recv_count** is the amount of data being received from each process

```
int MPI_Allgather(
        void*           send_buf_p   /* in  */,
        int             send_count   /* in  */,
        MPI_Datatype    send_type    /* in  */,
        void*           recv_buf_p   /* out */,
        int             recv_count   /* in  */,
        MPI_Datatype    recv_type    /* in  */,
        MPI_Comm        comm         /* in  */);
```

Matrix Vector Multiplication

$$\begin{array}{|c|c|c|c|} \hline a_{00} & a_{01} & \cdots & a_{0,n-1} \\ \hline a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\ \hline \end{array} \quad \begin{array}{|c|} \hline x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{n-1} \\ \hline \end{array} = \begin{array}{|c|} \hline y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \hline \vdots \\ \hline y_{m-1} \\ \hline \end{array}$$

Matrix Vector Multiplication: Serial Loop

```
/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

Matrix Vector Multiplication: Matrix

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

Stored as **contiguous memory location**:

0 1 2 3 4 5 6 7 8 9 10 11

Matrix Vector Multiplication: Parallel Code

```c
void Mat_vect_mult(
      double    local_A[]  /* in  */,
      double    local_x[]  /* in  */,
      double    local_y[]  /* out */,
      int       local_m    /* in  */,
      int       n          /* in  */,
      int       local_n    /* in  */,
      MPI_Comm  comm       /* in  */) {
   double* x;
   int local_i, j;
   int local_ok = 1;

   x = malloc(n*sizeof(double));
   MPI_Allgather(local_x, local_n, MPI_DOUBLE,
         x, local_n, MPI_DOUBLE, comm);

   for (local_i = 0; local_i < local_m; local_i++) {
      local_y[local_i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[local_i] += local_A[local_i*n+j]*x[j];
   }
   free(x);
}  /* Mat_vect_mult */
```

demo • mpi_matrix_vector_multiplication

Derived Datatypes

What are Derived Datatypes?

- Allows to **represent any collection of data items in memory** by storing **the types of the items** and **their relative locations in memory**
- Allows MPI communication functions to **handle custom user types properly**
- Works for both **send** and **receive** cases

What are Derived Datatypes?

Derived Datatypes $\equiv$ sequence of basics MPI types

Variable	Address
x	24
y	40
z	48

{(MPI_INTEGER,0),(MPI_INTEGER,16),(MPI_INTEGER,24)}

MPI_Type_create_struct

Builds a derived datatype that consists of individual elements that have **different basic types**

```
int MPI_Type_create_struct(
      int              count                      /* in    */,
      int              array_of_blocklengths[]    /* in    */,
      MPI_Aint         array_of_displacements[]   /* in    */,
      MPI_Datatype     array_of_types[]           /* in    */,
      MPI_Datatype*    new_type_p                 /* out   */);
```

Steps to Create a Derived Datatype

1. Found the right MPI types (MPI_DOUBLE, MPI_INT, ...)
2. Define their dimensions (usually 1)
3. Compute their relative displacement
4. Now ready to call **MPI_Type_create_struct**
5. **Commit** your type using MPI_Type_commit
6. Use your type
7. **FREE YOUR TYPE** using MPI_Type_free

How to compute displacement?

```
int MPI_Get_address(
      void*        location_p  /* in  */,
      MPI_Aint*    address_p   /* out */);
```

- Returns the address of the memory location referenced by **location_p**
- **MPI_Aint** is a big enough type to store addresses

Derived Datatypes Example

demo • mpi_derived_types

Performance Evaluation

Elapsed Parallel Time

Returns the number of seconds that have elapsed since some time in the past

```
double MPI_Wtime(void);

      double start, finish;
      . . .
      start = MPI_Wtime();
      /* Code to be timed */
      . . .
      finish = MPI_Wtime();
      printf("Proc %d > Elapsed time = %e seconds\n"
            my_rank, finish-start);
```

Barriers

- Synchronize processes $\Longrightarrow$ **use barriers**
- It ensures that no process will return from calling it until every process in the communicator has started calling it

```
int MPI_Barrier(MPI_Comm    comm    /* in */);
```

Barriers: Back to Performance

```c
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);
```

Conclusion

Takeaway messages

- MPI $\equiv$ **M**essage **P**assing **I**nterface
- MPI uses **SPMD**
- Communicator = **a reference to processes** that can communicate together
- Collective communications involve **ALL** processes of a communicator
- To measure performance we use **wall clock time**
- **Program unsafe** if correct behavior depends on buffering of MPI_Send