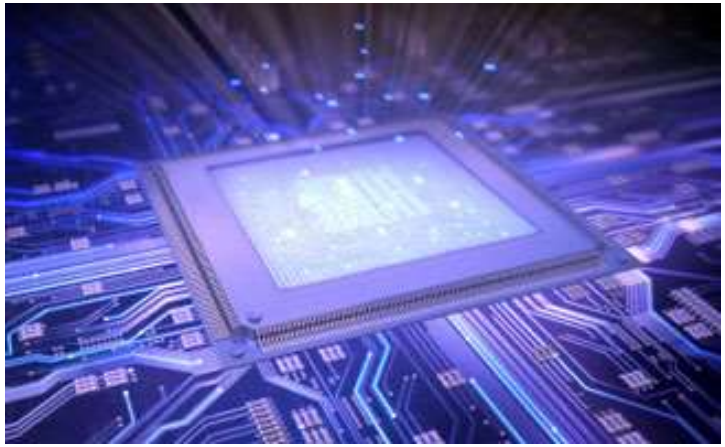
# Parallel Computing

## MPI - I

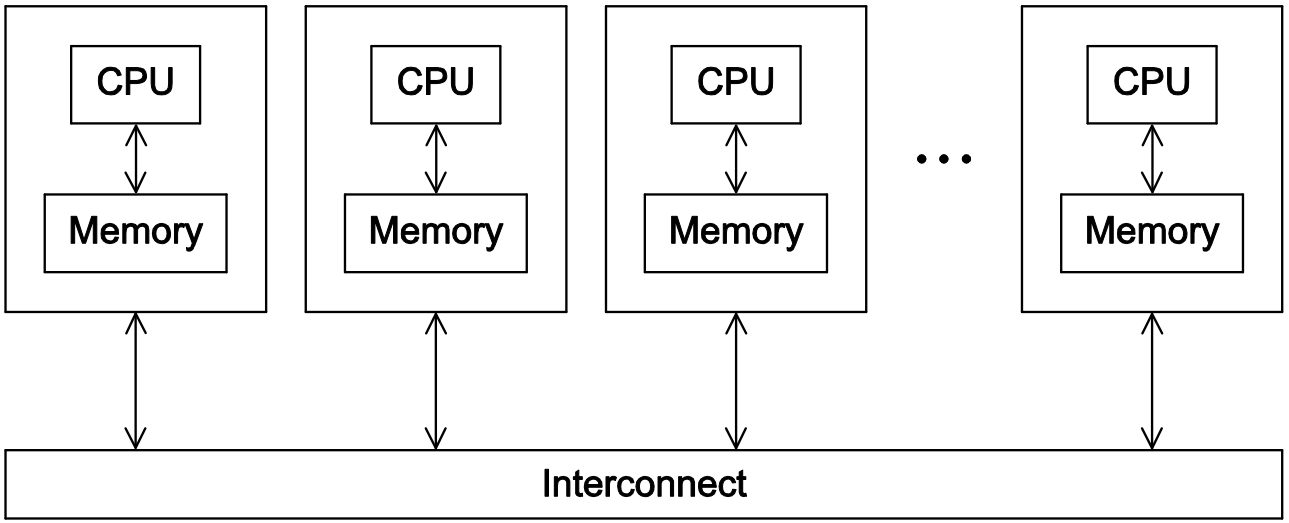
Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Many slides of this lecture are adopted and slightly modified from:

- Gerassimos Barlas
- Peter S. Pacheco

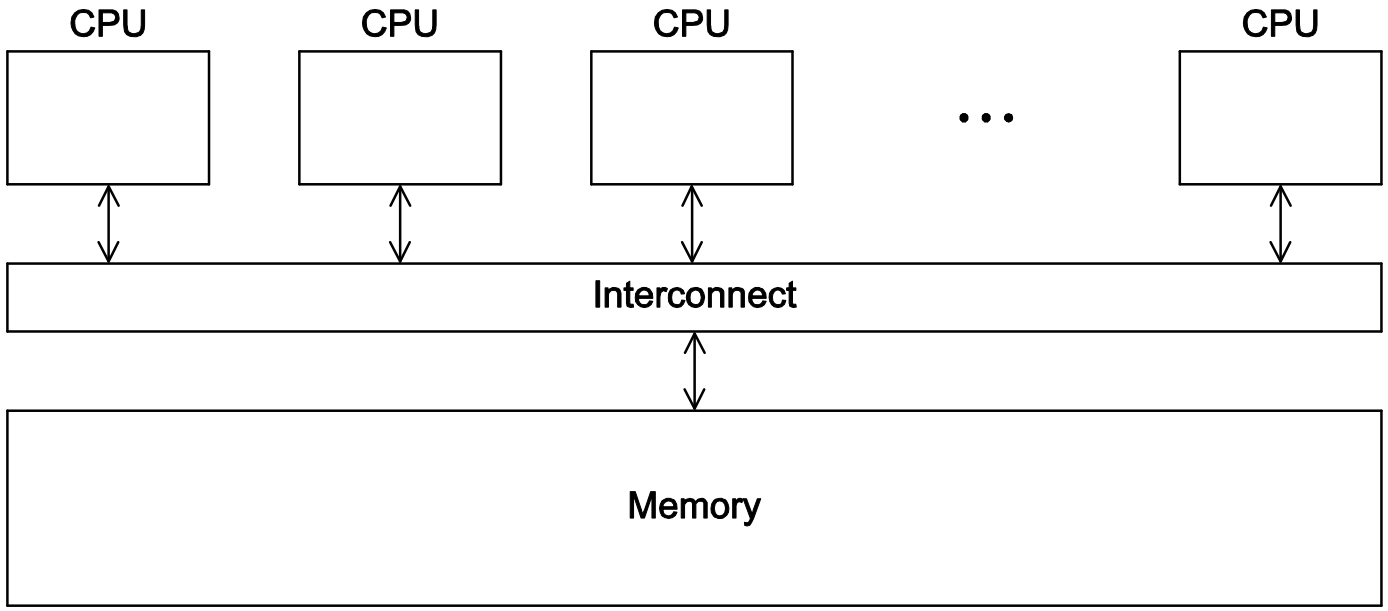


# This is What We Target With MPI




We will talk about processes

# We Will Study OpenMP for This



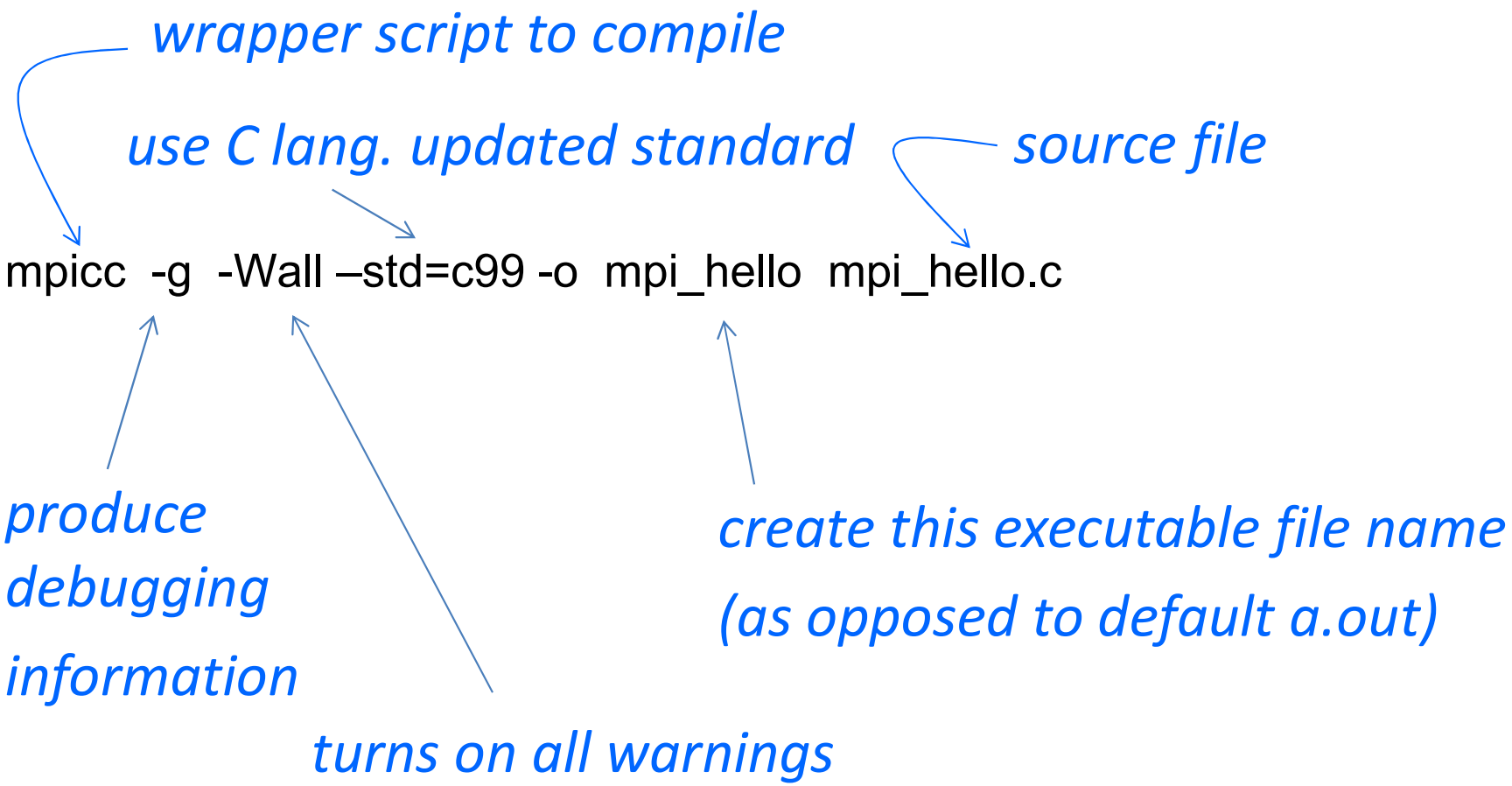

We will talk about Threads

# MPI processes

- Identify processes by non-negative integer ranks.

- *p* processes are numbered *0, 1, 2, .. p-1*

# Compilation




MPI is NOT a language.
Just libraries called from
C/C++, Fortran, and any
language that can call libraries from those.

# Execution

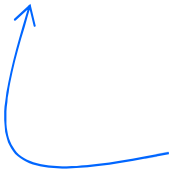
mpiexec -n <number of processes> <executable>

---

mpiexec -n 1 ./mpi_hello

*run with 1 process*

mpiexec -n 4 ./mpi_hello



*run with 4 processes*

You can use mpirun instead of mpiexec and -np instead of -n.

# Our first MPI program

```
1  #include <stdio.h>
2  #include <string.h>  /* For strlen             */
3  #include <mpi.h>     /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8     char        greeting[MAX_STRING];
9     int         comm_sz;  /* Number of processes */
10    int         my_rank;  /* My process rank     */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17       sprintf(greeting, "Greetings from process %d of %d!",
18             my_rank, comm_sz);
19       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
23       for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27       }
28    }
29
30    MPI_Finalize();
31    return 0;
32 }  /* main */
```

# Our first MPI program

```
1  #include <stdio.h>
2  #include <string.h>  /* For strlen             */
3  #include <mpi.h>     /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8     char        greeting[MAX_STRING];
9     int         comm_sz;  /* Number of processes */
10    int         my_rank;  /* My process rank     */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17       sprintf(greeting, "Greetings from process %d of %d!",
18               my_rank, comm_sz);
19       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
23       for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27       }
28    }
29
30    MPI_Finalize();
31    return 0;
32 }  /* main */
```

# 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 !
```

# MPI Programs

- Used mainly with C/C++ and Fortran
  - With some efforts with other languages going on and off.
  - But any language that can call libraries from the above can use MPI capabilities.
- Need to add mpi.h header file.
- Identifiers defined by MPI start with "**MPI_**".
  - First letter following underscore is uppercase.
    - For function names and MPI-defined types.
    - Helps to avoid confusion.
  - All letters following underscore are uppercase.
    - MPI defined macros
    - MPI defined constants

# MPI Components

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

Pointers to
the two arguments
of main()

Tells MPI to do all the necessary setup.
No MPI functions should be called before this.

# MPI Components

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

- Tells MPI we're done, so clean up anything allocated for this program.
- No MPI function should be called after this.

# 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;
}
```

# Communicators

- A collection of processes that can send messages to each other.
- MPI_Init defines a communicator that consists of all the processes created when the program started.
- Called MPI_COMM_WORLD.

# Communicators

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

*number of processes in the communicator*

*MPI_COMM_WORLD for now*

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

*my rank*
*(rank of the process making this call)*

# Communication

```
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 */);
```

num of elements in msg_buf

type of each element in msg_buf

To distinguish messages

rank of the receiving process

Message sent by a process using one communicator cannot be received by a process in another communicator.

# Data types

| 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 | |

# Communication

```
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 */);
```

# Message matching

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

*r*

recv_buf_sz
>=
send_buf_sz

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

*q*

*MPI_Send*

*src = q*

*MPI_Recv*

*dest = r*

# Scenario 1

```
1  #include <stdio.h>
2  #include <string.h>  /* For strlen             */
3  #include <mpi.h>     /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8     char        greeting[MAX_STRING];
9     int         comm_sz;  /* Number of processes */
10    int         my_rank;  /* My process rank     */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17       sprintf(greeting, "Greetings from process %d of %d!",
18             my_rank, comm_sz);
19       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
23       for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27       }
28    }
29
30    MPI_Finalize();
31    return 0;
32 }  /* main */
```

What if process 2 message arrives before process 1?

# Scenario 1

Wildcard: MPI_ANY_SOURCE

The loop will then be:

```
for (i = 1; i < comm_sz; i++) {
    MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,
            result_tag, comm, MPI_STATUS_IGNORE);
    Process_result(result);
}
```

# Scenario 2

What if process 1 sends to process 0 several messages but they arrive out of order.

- Process 0 is waiting for a message with tag = 0 but tag = 1 message arrives instead!

# Scenario 2

Wildcard: MPI_ANY_TAG

The loop will then be:

```
for(q = 1; q < comm_sz; q++) {
   MPI_Recv(result, result_sz, result_type,
   q,
   MPI_ANY_TAG, comm,
   MPI_STATUS_IGNORE);
}
```

# Receiving messages

- By using these wildcards, a receiver can get a message without knowing:
  - the amount of data in the message,
  - the sender of the message,
  - or the tag of the message.

# How will the output be different if ..

```
1  #include <stdio.h>
2  #include <string.h>  /* For strlen             */
3  #include <mpi.h>     /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8     char       greeting[MAX_STRING];
9     int        comm_sz;  /* Number of processes */
10    int        my_rank;  /* My process rank     */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17       sprintf(greeting, "Greetings from process %d of %d!",
18             my_rank, comm_sz);
19       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
23       for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27       }
28    }
29
30    MPI_Finalize();
31    return 0;
32 }  /* main */
```

- use MPI_ANY_SOURCE
- use MPI_ANY_TAG

# status argument

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

**Pointer to**

**MPI_Status struct**

*MPI_SOURCE*

*MPI_TAG*

*MPI_ERROR*

**MPI_Status status;**

**status.MPI_SOURCE**

**status.MPI_TAG**

# How much data am I receiving?

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

# Issues

- MPI_Send() is implementation dependent: can buffer or block .. or both!
- MPI_Recv() always blocks
  - So, if it returns we are sure the message has been received.
  - Be careful: don't make it block forever!

# Conclusions

- MPI is the choice when we have distributed memory organization.
- MPI is built around messages.
- Your goal: How to reduce messages yet increase concurrency?