CSCI-GA.3033-016
Multicore Processors: Architecture & Programming

Lecture: MPI

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
This is What We Target With MPI

We will talk about **processes**
OpenMP for Shared Memory

We will talk about Threads
Basics
MPI processes

• Identify processes by non-negative integer ranks.

• $p$ processes are numbered $0, 1, 2, \ldots, p-1$
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.
```
#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 */
```
Our first MPI program

```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
", greeting);
        }
    }

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

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

```c
int MPI_Init(
    int* argc_p /* in/out */,
    char*** argv_p /* in/out */);
```
Pointers to the two arguments of `main()`
• Tells MPI we’re done, so clean up anything allocated for this program.
• No MPI function should be called after this.

```c
int MPI_Finalize(void);
```
Basic Outline

```c
#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

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

- `number of processes in the communicator`

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

- `my rank`
  - `(rank of the process making this call)`

- `MPI_COMM_WORLD for now`
Message sent by a process using one communicator cannot be received by a process in another communicator.
## Data types

<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>
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 */);

Copyright © 2010, Elsevier Inc.
All rights Reserved
Message matching

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

recv_buf_sz >= send_buf_sz

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

src = q

dest = r
Scenario 1

What if process 2 message arrives before process 1?

```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 */
```
Scenario 1

Wildcard: **MPI_ANY_SOURCE**

The loop will then be:

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

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

• 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 ..

```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 */
```
status argument

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

MPI_Status* status;

status.MPI_SOURCE
status.MPI_TAG

MPI_SOURCE
MPI_TAG
MPI_ERROR
How much data am I receiving?

```c
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!
Dealing with I/O

In all MPI implementations, all processes in MPI_COMM_WORLD have access to stdout and stderr.

```c
#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 */
```

BUT .. In most of them there is no scheduling of access to output devices!
Running 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!!

• Processes are competing for stdout
• Result: nondeterminism!
How About Input?

- Most MPI implementations only allow process 0 in MPI_COMM_WORLD to access to stdin.

- If there is some input needed:
  - Process 0 must read the data and send to the other processes.
Function for reading user input

```c
void Get_input(
    int    my_rank  /* in */,
    int    comm_sz  /* in */,
    double* a_p     /* out */,
    double* b_p     /* out */,
    int*   n_p      /* out */)
{
    int dest;

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for (dest = 1; dest < comm_sz; dest++) {
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else { /* my_rank != 0 */
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

} /* Get_input */
```
Collective Communication
What if:
Each process calculates part of the solution and you want to combine the partial results?
Reduction

• Reducing a set of numbers into a smaller set of numbers via a function
  – Example: reducing the group [1, 2, 3, 4, 5] with the sum function → 15

• MPI provides a handy function that handles almost all of the common reductions that a programmer needs to do in a parallel application
Every process has an element

Every process has an array of elements
MPI_Reduce

```c
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 */);
```

Examples:

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

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

MPI_Reduce is called by all processes involved. This is why it is called **collective call**.

has size:
```
sizeof(datatype) * count
```

only relevant to dest_process
Predefined reduction operators in MPI

<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>

Location = rank of the process that owns it
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 erroneous.

• The arguments passed by each process to an MPI collective communication must be “compatible.”
  – For example, if one process passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous.
Collective vs. Point-to-Point Communications

- The `output_data_p` argument is only used on `dest_process`.
- However, all of the processes still need to pass in an actual argument corresponding to `output_data_p`, even if it's just `NULL`.
- All collective communication calls are blocking.
Collective vs. Point-to-Point Communications

• Point-to-point communications are matched on the basis of tags and communicators.

• Collective communications don’t use tags.

• They’re matched solely on the basis of the communicator and the order in which they’re called.
Example

Assume:
- all processes use the operator MPI\_SUM
- destination is process 0

What will be the final values of b and d??

<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a = 1; c = 2</td>
<td>a = 1; c = 2</td>
<td>a = 1; c = 2</td>
</tr>
<tr>
<td>1</td>
<td>MPI_Reduce(&amp;a, &amp;b, ...)</td>
<td>MPI_Reduce(&amp;c, &amp;d, ...)</td>
<td>MPI_Reduce(&amp;a, &amp;b, ...)</td>
</tr>
<tr>
<td>2</td>
<td>MPI_Reduce(&amp;c, &amp;d, ...)</td>
<td>MPI_Reduce(&amp;a, &amp;b, ...)</td>
<td>MPI_Reduce(&amp;c, &amp;d, ...)</td>
</tr>
</tbody>
</table>
Yet Another Example

MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);

This is illegal in MPI and the result is non-predictable!
MPI_Allreduce

• Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

```c
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 */);
```

No destination argument!
Broadcast

• Data belonging to a single process is sent to all of the processes in the communicator.

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

ALL processes in the communicator must call MPI_Bcast()
A tree-structured broadcast.
A version of Get_input that uses MPI_Bcast

```c
void Get_input(
    int my_rank /* in */,
    int comm_sz /* in */,
    double* a_p    /* out */,
    double* b_p    /* out */,
    int* n_p       /* out */) {

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }

    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
/* Get_input */
```
So Far:

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

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 */);

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

```
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 */);

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 */);
```
Data distributions

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

```c
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 */
```
Different partitions of a 12-component vector among 3 processes

<table>
<thead>
<tr>
<th>Process</th>
<th>Block</th>
<th>Cyclic</th>
<th>Block-cyclic Blocksize = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2 3</td>
<td>0 3 6 9</td>
<td>0 1 6 7</td>
</tr>
<tr>
<td>1</td>
<td>4 5 6 7</td>
<td>1 4 7 10</td>
<td>2 3 8 9</td>
</tr>
<tr>
<td>2</td>
<td>8 9 10 11</td>
<td>2 5 8 11</td>
<td>4 5 10 11</td>
</tr>
</tbody>
</table>

- **Block**: Assign blocks of consecutive components to each process.
- **Cyclic**: Assign components in a round robin fashion.
- **Block-cyclic**: Use a cyclic distribution of blocks of components.
Parallel implementation of vector addition

```c
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 */
```

How will you distribute parts of x[] and y[] to processes?
Scatter

• Read an entire vector on process 0
• **MPI_Scatter** sends the needed components to each of the other processes.

```c
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 */
);
```

Important:
• All arguments are important for the source process (process 0 in our example)
• For all other processes, only recv_buf_p, recv_count, recv_type, src_proc, and comm are important
Reading and distributing a vector

```c
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 */
```

process 0 itself also receives data.
• **send**\_buf\_p
  - is not used except by the sender.
  - However, it must be defined or NULL on others to make the code correct.
  - Must have at least communicator size * send\_count elements

• All processes must call **MPI\_Scatter**, not only the sender.

• **send\_count** the number of data items sent to each process.

• **recv**\_buf\_p must have at least send\_count elements

• **MPI\_Scatter** uses block distribution

```c
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 */);
```
```
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 */);
```
**Gather**

- **MPI_Gather** collects all of the components of the vector onto process dest process, ordered in rank order.

```c
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 */);
```

**Important:**
- All arguments are important for the destination process.
- For all other processes, only `send_buf_p`, `send_count`, `send_type`, `dest_proc`, and `comm` are important.
Print a distributed vector (1)

```c
void Print_vector(
    double local_b[], /* in */,
    int local_n, /* in */,
    int n, /* in */,
    char title[], /* in */,
    int my_rank, /* in */,
    MPI_Comm comm /* in */) {

    double* b = NULL;
    int i;
```
if (my_rank == 0) {
    b = malloc(n*sizeof(double));
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
    printf("%s\n", title);
    for (i = 0; i < n; i++)
        printf("%f ", b[i]);
    printf("\n");
    free(b);
} else {
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
}
/* Print_vector */
Allgather

• Concatenates the contents of each process’ `send_buf_p` and stores this in each process’ `recv_buf_p`.

• As usual, `recv_count` is the amount of data being received from each process.

```c
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

\[ A = (a_{ij}) \text{ is an } m \times n \text{ matrix} \]

\[ \mathbf{x} \text{ is a vector with } n \text{ components} \]

\[ \mathbf{y} = A\mathbf{x} \text{ is a vector with } m \text{ components} \]

\[ y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{i,n-1}x_{n-1} \]

\[ i\text{-th component of } \mathbf{y} \]

\[ \text{Dot product of the } i\text{th row of } A \text{ with } \mathbf{x}. \]
Matrix-vector multiplication

\[
\begin{array}{cccc}
  a_{00} & a_{01} & \cdots & a_{0,n-1} \\
  a_{10} & a_{11} & \cdots & a_{1,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\end{array}
\]

\[
\begin{array}{c}
  x_0 \\
  x_1 \\
  \vdots \\
  x_{n-1} \\
\end{array}
\]

\[
\begin{array}{c}
  y_0 \\
  y_1 \\
  \vdots \\
  y_{m-1} \\
\end{array}
\]

\[
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}
\]

/* 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];
}

Pseudo-code Serial Version
C style arrays

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

stored as

0 1 2 3 4 5 6 7 8 9 10 11
Serial matrix-vector multiplication

```c
void Mat_vec_t mult(
    double A[] /* in */,
    double x[] /* in */,
    double y[] /* out */,
    int m /* in */,
    int n /* in */) {
    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
} /* Mat_vec_t mult */
```

Let's assume x[] is distributed among the different processes
An MPI matrix-vector multiplication function (1)

```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;
```
An MPI matrix-vector multiplication function (2)

```c
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_vec_mult */
```
Keep in mind ...

• In distributed memory systems, communication is more expensive than computation.

• Distributing a fixed amount of data among several messages is more expensive than sending a single big message.
Communicative Derived Data Types
Derived datatypes

• Used to represent any collection of data items

• If a function that sends data knows this information about a collection of data items, it can collect the items from memory before they are sent.

• A function that receives data can distribute the items into their correct destinations in memory when they’re received.
Derived datatypes

- A sequence of basic MPI data types together with a displacement for each of the data types.

Address in memory where the variables are stored

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>24</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
</tr>
<tr>
<td>n</td>
<td>48</td>
</tr>
</tbody>
</table>

\{(\text{MPI\_DOUBLE}, 0), (\text{MPI\_DOUBLE}, 16), (\text{MPI\_INT}, 24)\}

displacement from the beginning of the type
(We assume we start with a.)
**MPI_Type create_struct**

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

```c
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 */);
```

- An integer type that is big enough to store an address on the system.

```c
int MPI_Get_address(
    void* location_p /* in */,
    MPI_Aint* address_p /* out */);
```
Before you start using your new data type

```c
int MPI_Type_commit(MPI_Datatype* new_mpi_t_p /* in/out */);
```

Allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.
When you are finished with your new type

```c
int MPI_Type_free(MPI_Datatype* old_mpi_t_p /* in/out */);
```

This frees any additional storage used.
```c
void Build_mpi_type(
    double* a_p, /* in */,
    double* b_p, /* in */,
    int* n_p, /* in */,
    MPI_Datatype* input_mpi_t_p /* out */) {

    int array_of_blocklengths[3] = {1, 1, 1};
    MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
    MPI_Aint a_addr, b_addr, n_addr;
    MPI_Aint array_of_displacements[3] = {0};
```
Example (2)

MPI_Get_address(a_p, &a_addr);
MPI_Get_address(b_p, &b_addr);
MPI_Get_address(n_p, &n_addr);
array_of_displacements[1] = b_addr - a_addr;
MPI_Type_create_struct(3, array_of_blocklengths,
                      array_of_displacements, array_of_types,
                      input_mpi_t_p);
MPI_Type_commit(input_mpi_t_p);

} /* Build_mpi_type */
Example (3)

```c
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
               int* n_p) {
    MPI_Datatype input_mpi_t;

    Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

    if (my_rank == 0) {
        printf("Enter a, b, and n\n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

    MPI_Type_free(&input_mpi_t);
} /* Get_input */
```

The receiving end can use the received complex data item as if it is a structure.
MEASURING TIME IN MPI
We have seen in the past ...

- `time` in Linux
- `clock()` inside your code
- Does MPI offer anything else?
Elapsed parallel time

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

```c
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);
```
How to Sync Processes?
MPI_Barrier

- Ensures that no process will return from calling it until every process in the communicator has started calling it.

```c
int MPI_Barrier(MPI_Comm comm /* in */);
```
Performance Analysis
Let's see how we can analyze the performance of an MPI program.

The matrix-vector multiplication:

```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);
```
Run-times of serial and parallel matrix-vector multiplication

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>Order of Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1024</td>
</tr>
<tr>
<td>1</td>
<td>4.1</td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
</tr>
<tr>
<td>4</td>
<td>2.0</td>
</tr>
<tr>
<td>8</td>
<td>1.7</td>
</tr>
<tr>
<td>16</td>
<td>1.7</td>
</tr>
</tbody>
</table>

(Seconds)
**Speedups of Parallel Matrix-Vector Multiplication**

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16,384</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>1.9</td>
<td>1.9</td>
<td>1.9</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>2.1</td>
<td>3.1</td>
<td>3.6</td>
<td>3.9</td>
<td>3.9</td>
</tr>
<tr>
<td>8</td>
<td>2.4</td>
<td>4.8</td>
<td>6.5</td>
<td>7.5</td>
<td>7.9</td>
</tr>
<tr>
<td>16</td>
<td>2.4</td>
<td>6.2</td>
<td>10.8</td>
<td>14.2</td>
<td>15.5</td>
</tr>
</tbody>
</table>
Efficiencies of Parallel Matrix-Vector Multiplication

<table>
<thead>
<tr>
<th>comm_sz</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
<th>16,384</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.89</td>
<td>0.94</td>
<td>0.97</td>
<td>0.96</td>
<td>0.98</td>
</tr>
<tr>
<td>4</td>
<td>0.51</td>
<td>0.78</td>
<td>0.89</td>
<td>0.96</td>
<td>0.98</td>
</tr>
<tr>
<td>8</td>
<td>0.30</td>
<td>0.61</td>
<td>0.82</td>
<td>0.94</td>
<td>0.98</td>
</tr>
<tr>
<td>16</td>
<td>0.15</td>
<td>0.39</td>
<td>0.68</td>
<td>0.89</td>
<td>0.97</td>
</tr>
</tbody>
</table>
Safety: Avoiding Deadlock
Safety in MPI programs

• The MPI standard allows MPI_Send to behave in two different ways:
  – it can simply copy the message into an MPI managed buffer and return,
  – or it can block until the matching call to MPI_Recv starts.
Safety in MPI programs

• Many implementations of MPI set a threshold at which the system switches from buffering to blocking.
  – Relatively small messages will be buffered by MPI_Send.
  – Larger messages, will cause it to block.
Safety in MPI programs

• If the MPI_Send executed by each process blocks, no process will be able to start executing a call to MPI_Recv, and the program will hang or deadlock.

• Each process is blocked waiting for an event that will never happen.
Safety in MPI programs

• A program that relies on MPI provided buffering is said to be unsafe.

• Such a program may run without problems for various sets of input, but it may hang or crash with other sets.

So ... What can we do?
MPI_Ssend

- An alternative to MPI_Send defined by the MPI standard.
- The extra “s” stands for synchronous and MPI_Ssend is guaranteed to block until the matching receive starts.

```c
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 does MPI_Ssend help?

- Replace all MPI_Send calls in your code with MPI_Ssend
- If your program does not hang or crash → the original program is safe
- What do we do if we find out that our program is not safe?
- The main problem is due to the fact that all processes send then receive... Let’s change that!
Restructuring communication

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);
}

Note: The above two versions show a ring communication
(i.e. processor comm_sz-1 sends to process 0.)
MPI_Sendrecv

- An alternative to scheduling the communications ourselves.
- Carries out a blocking send and a receive in a single call.
- Especially useful because MPI schedules the communications so that the program won’t hang or crash.
- Replaces a pair of consecutive send and receive calls.
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 */);
int MPI_Sendrecv_replace{
    void * buf_p,
    int buf_size,
    MPI_Datatype buf_type,
    int dest,
    int send_tag,
    int recv_tag,
    MPI_Comm communicator,
    MPI_Status * status_p
};

In this case, what is in buf_p will be sent and replaced by what is received.
PREFIX-SUM
What is that?

• Generalization of global sum
• Input: vector $x[]$ of $n$ elements
• Output: vector prefix_sum[] of $n$ elements, such that:
  - $\text{prefix\_sum}[0] = x[0]$
  - $\text{prefix\_sum}[1] = x[0] + x[1]$
  - $\text{prefix\_sum}[n-1] = x[0] + x[1] + \ldots + x[n-1]$
/* First compute prefix sums of my local vector */
loc_prefix_sums[0] = loc_vect[0];
for (loc_i = 1; loc_i < loc_n; loc_i++)
    loc_prefix_sums[loc_i] = loc_prefix_sums[loc_i-1] + loc_vect[loc_i];

if (my_rank != 0) {
    /* If I'm not 0 receive sum of preceding components */
    MPI_Recv(&sum_of_preceding, 1, MPI_DOUBLE, my_rank-1, 0, comm, 
              MPI_STATUS_IGNORE);

    /* Add in sum of preceding components to my prefix sums */
    for (loc_i = 0; loc_i < n; loc_i++)
        loc_prefix_sums[loc_i] += sum_of_preceding;
}

/* Now send my last element to the next process */
if (my_rank != comm_sz - 1)
    MPI_Send(&loc_prefix_sums[loc_n-1], 1, MPI_DOUBLE, my_rank+1, 0, comm);
MPI_Scan

int MPI_Scan(
    void * sendbuf,
    void * recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm comm);

Returns for process of rank i, the prefix reduction values for elements 0 ... i
Be Careful

Assume 4 processes:

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
10 & 11 & 12 \\
\end{array}
\]

The output of MPI_Scan, for MPI_SUM, is:

\[
\begin{array}{ccc}
1 & 2 & 3 \\
5 & 7 & 9 \\
12 & 15 & 18 \\
22 & 26 & 30 \\
\end{array}
\]

NOT:

\[
\begin{array}{ccc}
1 & 3 & 6 \\
10 & 15 & 21 \\
28 & 36 & 45 \\
55 & 66 & 78 \\
\end{array}
\]
Reconsidering the Communicator
The Communicator(s)

- We are familiar with the communicator `MPI_COMM_WORLD`
- A communicator can be thought of a handle to a group of an ordered set of processes
- For many applications maintaining different groups is appropriate
- Groups allow collective operations to work on a subset of processes
int MPI_Comm_split(
    MPI_Comm comm,
    int color,
    int key,
    MPI_Comm * newcomm);

Called by all processes in comm
Must be non-negative
Rank of the process in newcomm

The original communicator does not go away!
MPI_Comm_split

- Partitions the group associated with comm into disjoint subgroups
- Processes with the same color will be in the same group
- Within each subgroup, the processes are ranked in the order defined by the value of the argument key
  - with ties broken according to their rank in the old group
**MPI_Comm_split**

- If a process uses the color `MPI_UNDEFINED` it won’t be included in the new communicator.
MPI_Comm_free

int MPI_Comm_free(
    MPI_Comm * newcomm);

• Deallocation of created communicator
• Better do it if you are not using the comm again.
Example

Split a Large Communicator Into Smaller Communicators

Source: http://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/
Example

// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
int color = world_rank / 4;

// Determine color based on row
// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);
printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
     world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);

Source: http://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/
Output:

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

Source: http://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/
MPI_Comm_dup

int MPI_Comm_dup(
    MPI_Comm comm,
    MPI_Comm * newcomm);

• Creates an exact copy of comm in newcomm
Groups and Communicators

• In reality, processes are ordered in groups
• Communicators are the mean by which processes communicate
• A process can belong to more than one group, with different rank in each.
• ... But we will not got deeper than that here!
Words of Wisdom!
Don’t Forget!

• **MPI is a library**
  → Any MPI operation requires one or more function calls.
  → Not very efficient for very short data transfers.
  → Communication should be aggregated as much as possible.

• **Avoid unnecessary synchronizations.**
When to use MPI

• Portability and Performance
• Irregular Data Structures
• Building Tools for Others
  – Libraries
• Need to Manage memory on a per process basis
When not to use MPI

- Programs that have irregular communication patterns are often difficult to express in MPI's message-passing model.
- Domain-specific applications with an API tailored to that application
- Require Fault Tolerance
Strengths of MPI

• **Small**
  - Many programs can be written with only 6 basic functions

• **Large**
  - MPI’s extensive functionality (MPI-1 contains about 125 API, let alone MPI-2 and MPI-3)

• **Scalable**
  - Point-to-point communication

• **Flexible**
  - Don’t need to rewrite parallel programs across platforms
Hybrid OpenMP and MPI
MPI vs. OpenMP

- **Pure MPI Pro:**
  - Portable to distributed and shared memory machines.
  - Scales beyond one node
  - No race condition problem

- **Pure MPI Con:**
  - Difficult to develop and debug
  - High latency, low bandwidth
  - Explicit communication
  - Large granularity
  - Difficult load balancing

- **Pure OpenMP Pro:**
  - Easy to implement parallelism
  - Low latency, high bandwidth
  - Implicit Communication
  - Coarse and fine granularity
  - Dynamic load balancing

- **Pure OpenMP Con:**
  - Only on shared memory machines
  - Scale within one node
  - Possible race condition problem
  - No specific thread order
Why Hybrid?

- Hybrid MPI/OpenMP paradigm is the software trend for clusters of SMP architectures, supercomputers (although GPUs are usually also used here), ... .
- Elegant in concept and architecture: using MPI across nodes and OpenMP within nodes. Good usage of shared memory system resource (memory, latency, and bandwidth).
- Avoids the extra communication overhead with MPI within node.
- OpenMP adds fine granularity and allows increased and/or dynamic load balancing.
- Some problems have two-level parallelism naturally.
- Could have better scalability than both pure MPI and pure OpenMP.
Why Mixed OpenMP/MPI Code is Sometimes Slower?

- OpenMP has less scalability due to implicit parallelism
- MPI communication.
- Thread creation overhead
- Cache coherence
- Natural one level parallelism problems.
- Pure OpenMP code performs worse than pure MPI within node.
- Lack of optimized OpenMP compilers/libraries.
Hybrid Parallelization Strategies

- **From sequential code:** decompose with MPI first, then add OpenMP.
- **Simplest and least error-prone way is:**
  - Use MPI outside parallel region.
  - Allow only master thread to communicate between MPI tasks.
Multi-dimensional array transpose: Pure MPI and Pure OpenMP within One Node

OpenMP vs. MPI (16 CPUs)
64x512x128: 2.76 times faster
16x1024x256: 1.99 times faster
Pure MPI and Hybrid MPI/OpenMP Across Nodes

With 128 CPUs, n_thrds=4 hybrid MPI/OpenMP performs faster than n_thrds=16 hybrid by a factor of 1.59, and faster than pure MPI by a factor of 4.44.
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]) {
    int iam = 0, np = 1;

    #pragma omp parallel default(shared) private(iam, np)
    {
        #if defined (_OPENMP)
            np = omp_get_num_threads();
            iam = omp_get_thread_num();
        #endif
        printf("Hello from thread %d out of %d\n", iam, np);
    }
}
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    int numprocs, rank, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &namelen);
    
    printf("Process %d on %s out of %d\n", rank,
            processor_name, numprocs);
    
    MPI_Finalize();
}
Hybrid MPI + OpenMP: mixhello.c

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

int main(int argc, char *argv[]) {
    int numprocs, rank, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int iam = 0, np = 1;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &namelen);

    #pragma omp parallel default(shared) private(iam, np)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Hello from thread %d out of %d from process %d out of %d on %s\n",
                iam, np, rank, numprocs, processor_name);
    }

    MPI_Finalize();
}
```
Compile and Execution

mpicc -fopenmp mixhello.c

mpiexec -n x ./a.out
Conclusions

• You now know enough to use MPI in many problem solving
• We have not studied all APIs though.
• It is fairly easy to understand the rest of APIs.
• The main rules:
  – Reduce communication
  – Ensure load-balancing
  – Increase concurrency