



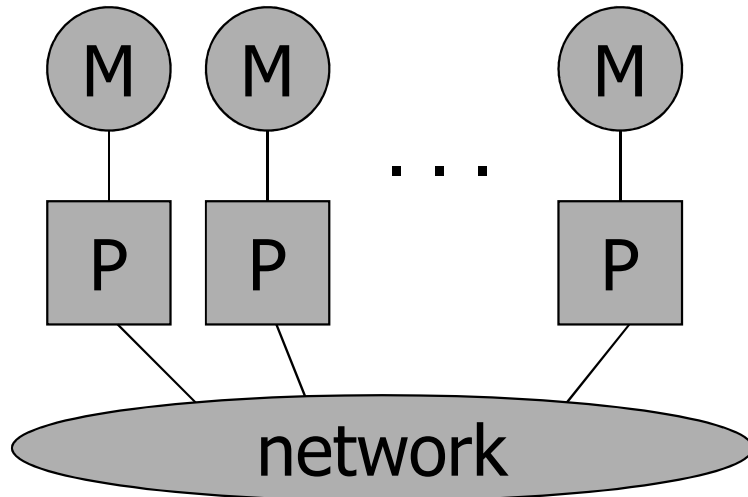
# ***The Message Passing Interface (MPI)***

---

SAN DIEGIO SUPERCOMPUTER  
CENTER

[www.sdsc.edu](http://www.sdsc.edu)

# Message Passing



- Each processor runs a process
- Processes communicate by exchanging messages
- They cannot share memory in the sense that they cannot address the same memory cells

- The above is a programming model and things may look different in the actual implementation (e.g., MPI over Shared Memory)
- Message Passing is popular because it is general:
  - Pretty much any distributed system works by exchanging messages, at some level
  - Distributed- or shared-memory multiprocessors, networks of workstations, uniprocessors
- It is not popular because it is easy (it's not)



# MPI Concepts

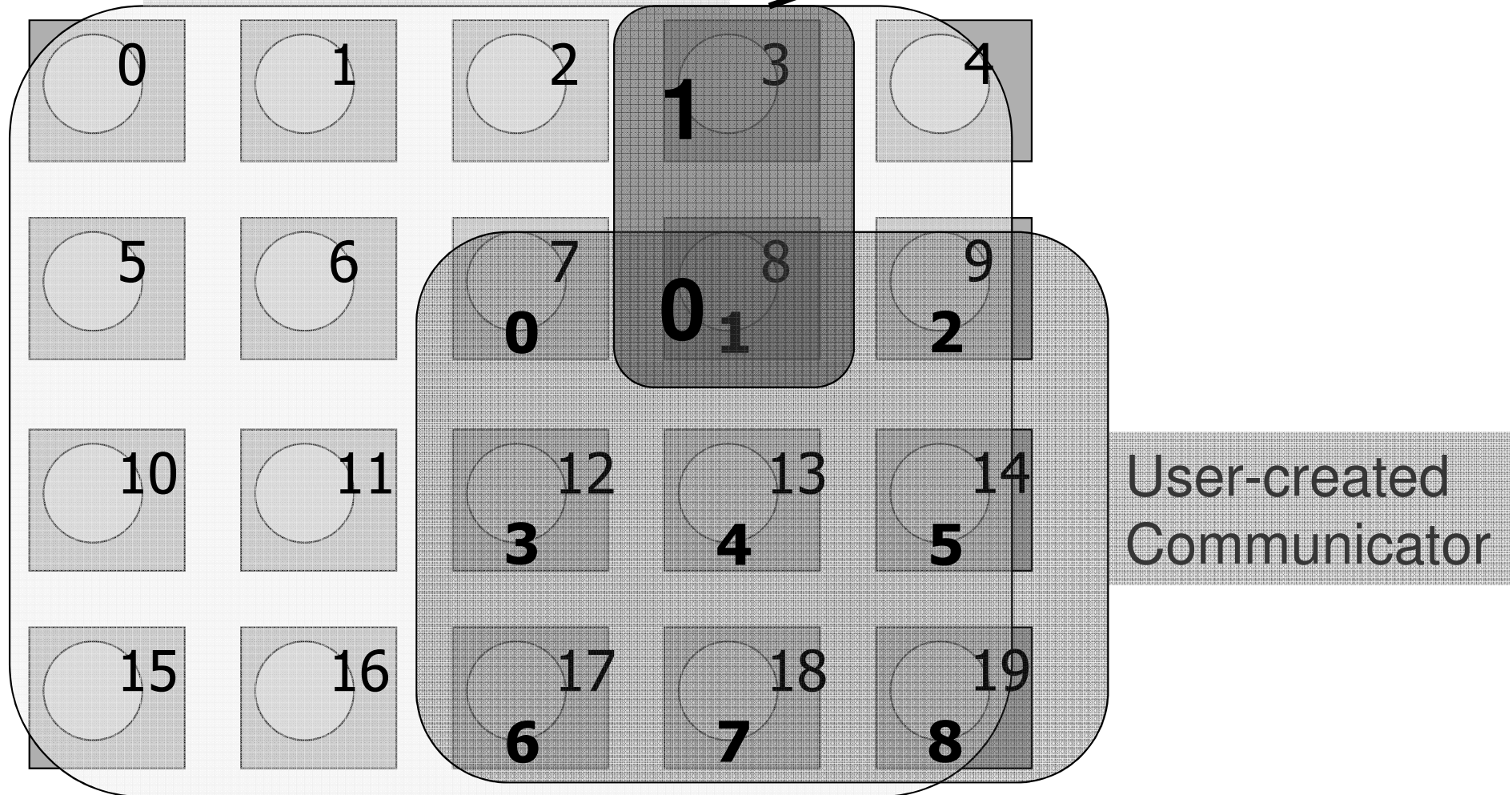
---

- Fixed number of processors
  - When launching the application one must specify the number of processors to use, which remains unchanged throughout execution
- Communicator
  - Abstraction for a group of processes that can communicate
  - A process can belong to multiple communicators
  - Makes it easy to partition/organize the application in multiple layers of communicating processes
  - Default and global communicator: `MPI_COMM_WORLD`
- Process Rank
  - The index of a process within a communicator
  - Typically user maps his/her own virtual topology on top of just linear ranks
    - ring, grid, etc.

# MPI Communicators

User-created  
Communicator

MPI\_COMM\_WORLD



# A First MPI Program

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
    int my_rank, n;
    char hostname[128];
    MPI_init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n);
    gethostname(hostname, 128);
    if (my_rank == 0) { /* master */
        printf("I am the master: %s\n", hostname);
    } else { /* worker */
        printf("I am a worker: %s (rank=%d/%d)\n",
            hostname, my_rank, n-1);
    }
    MPI_Finalize();
    exit(0);
}
```

Has to be called first, and once

Has to be called last, and once

# Compiling/Running it

- Link with `libmpi.a`
- Run with `mpirun`
  - `% mpirun -np 4 my_program <args>`
    - requests 4 processors for running `my_program` with command-line arguments
    - see the `mpirun` man page for more information
    - in particular the `-machinefile` option that is used to run on a network of workstations

- Some systems just run all programs as MPI programs and no explicit call to `mpirun` is actually needed

- Previous example program:

```
% mpirun -np 3 -machinefile hosts my_program
```

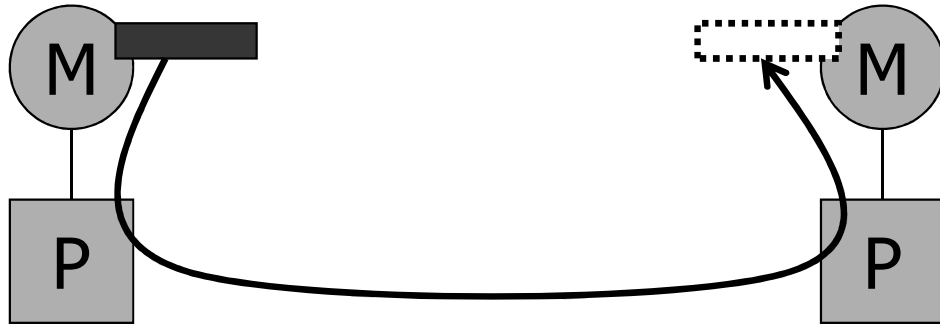
```
I am the master: somehost1
```

```
I am a worker: somehost2 (rank=2/2)
```

```
I am a worker: somehost3 (rank=1/2)
```

(stdout/stderr redirected o the process calling `mpirun`)

# Point-to-Point Communication



- Data to be communicated is described by three things:
  - address
  - data type of the message
  - length of the message
- Involved processes are described by two things
  - communicator
  - rank
- Message is identified by a “tag” (integer) that can be chosen by the user



# Point-to-Point Communication

---

- Two modes of communication:
  - Synchronous: Communication does not complete until the message has been received
  - Asynchronous: Completes as soon as the message is “on its way”, and hopefully it gets to destination
- MPI provides four versions
  - synchronous, buffered, standard, ready





# Synchronous/Buffered sending in MPI

- Synchronous with MPI\_Ssend
  - The send completes only once the receive has succeeded
    - copy data to the network, wait for an ack
    - The sender has to wait for a receive to be posted
    - No buffering of data
- Buffered with MPI\_Bsend
  - The send completes once the message has been buffered internally by MPI
    - Buffering incurs an extra memory copy
    - Do not require a matching receive to be posted
    - May cause buffer overflow if many bsend and no matching receives have been posted yet



# Standard/Ready Send

---

- Standard with MPI\_Send
  - Up to MPI to decide whether to do synchronous or buffered, for performance reasons
  - The rationale is that a correct MPI program should not rely on buffering to ensure correct semantics
- Ready with MPI\_Rsend
  - May be started *only* if the matching receive has been posted
  - Can be done efficiently on some systems as no handshaking is required

# Example: Sending and Receiving

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
    int i, my_rank, nprocs, x[4];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (my_rank == 0) { /* master */
        x[0]=42; x[1]=43; x[2]=44; x[3]=45;
        MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
        for (i=1; i<nprocs; i++)
            MPI_Send(x, 4, MPI_INT, i, 0, MPI_COMM_WORLD);
    } else { /* worker */
        MPI_Status status;
        MPI_Recv(x, 4, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
    exit(0);
}
```

destination and source

user-defined tag

Max number of elements to receive

Can be examined via calls like `MPI_Get_count()`, etc.

# Non-blocking Communication

- `MPI_Issend`, `MPI_IbSEND`, `MPI_Isend`, `MPI_IrSEND`, `MPI_Irecv`

```
MPI_Request request;
```

```
MPI_Isend(&x, 1, MPI_INT, dest, tag, communicator, &request);
```

```
MPI_Irecv(&x, 1, MPI_INT, src, tag, communicator, &request);
```

- Functions to check on completion: `MPI_Wait`, `MPI_Test`, `MPI_Waitany`, `MPI_Testany`, `MPI_Waitall`, `MPI_Testall`, `MPI_Waitsome`, `MPI_Testsome`.

```
MPI_Status status;
```

```
MPI_Wait(&request, &status) /* block */
```

```
MPI_Test(&request, &status) /* doesn't block */
```



# Collective Communication

---

- Operations that allow more than 2 processes to communicate simultaneously
  - barrier
  - broadcast
  - reduce
- All these can be built using point-to-point communications, but typical MPI implementations have optimized them, and it's a good idea to use them
- In all of these, all processes place the same call (in good SPMD fashion), although depending on the process, some arguments may not be used



# Barrier

---

- Synchronization of the calling processes
  - the call blocks until all of the processes have placed the call
- No data is exchanged

...

```
MPI_Barrier (MPI_COMM_WORLD)
```

...

# Broadcast

- One-to-many communication
- Note that multicast can be implemented via the use of communicators (i.e., to create processor groups)

...

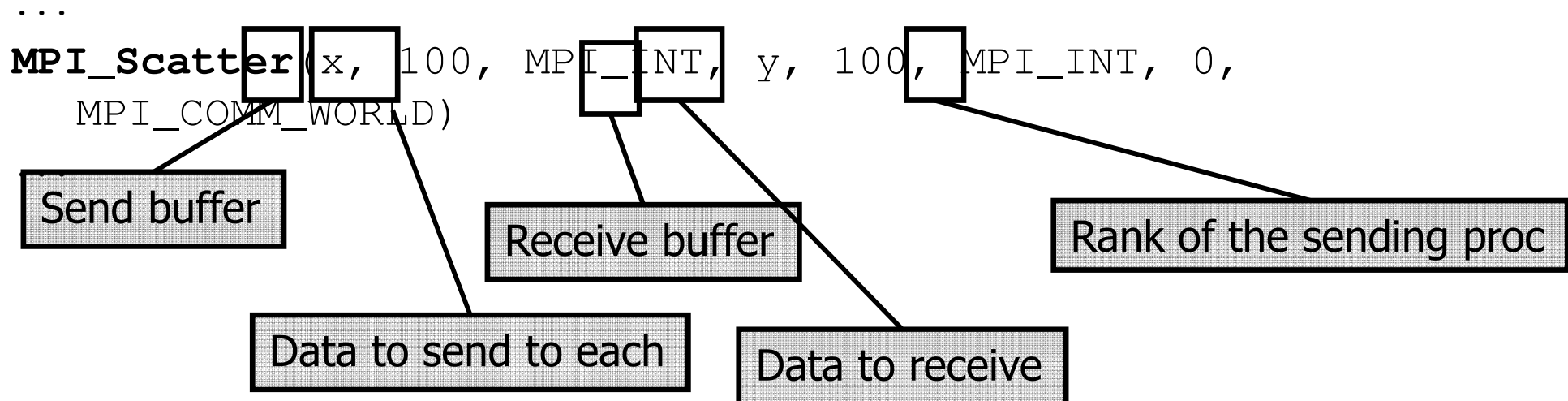
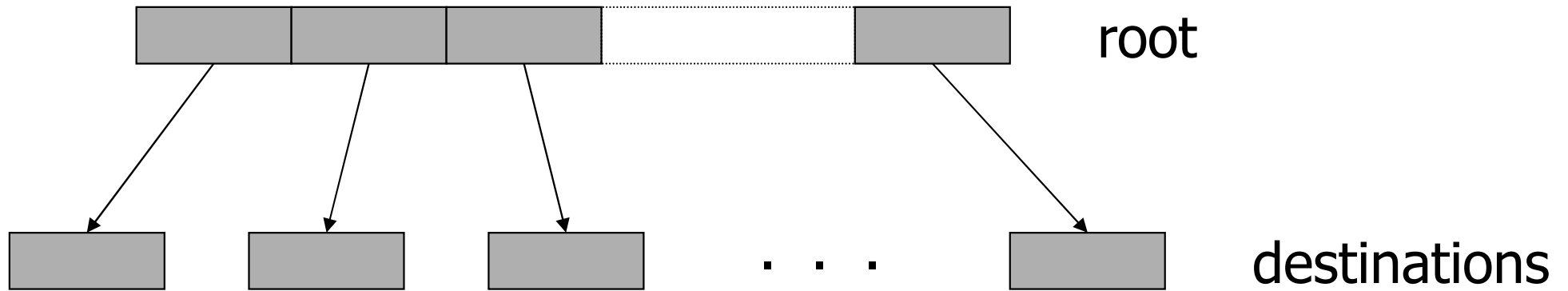
```
MPI_Bcast (x, 4, MPI_INT, 0,  
MPI_COMM_WORLD)
```

...

Rank of the root

# Scatter

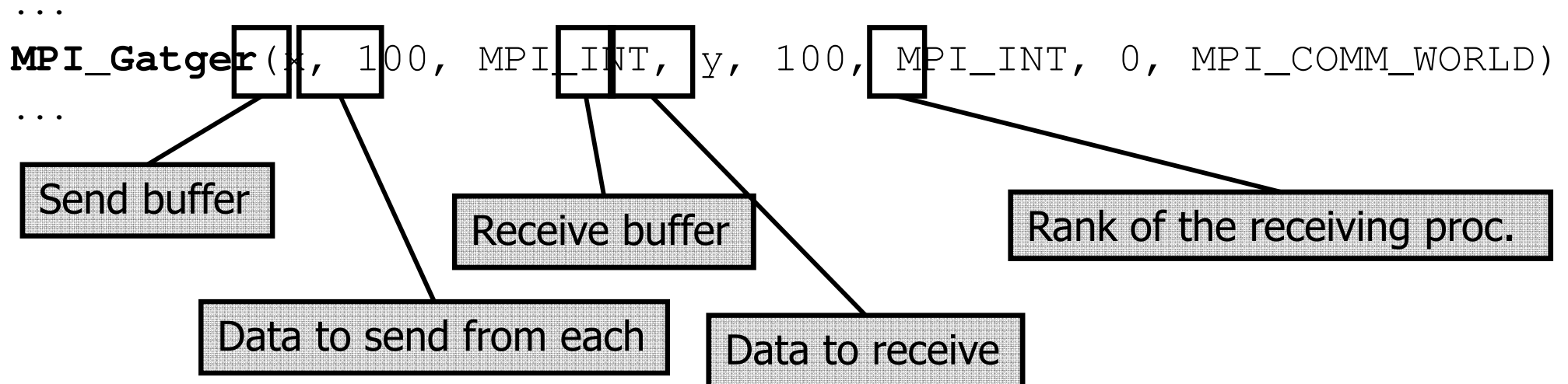
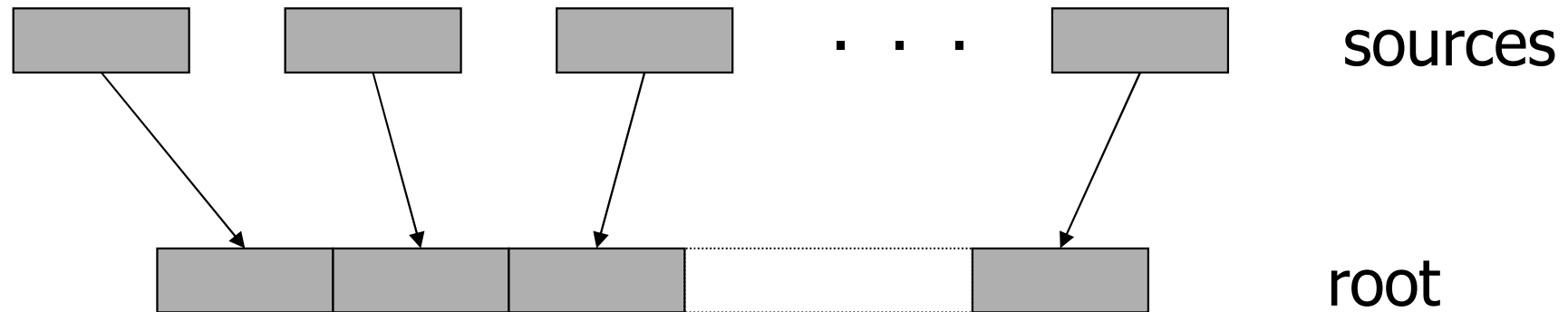
- One-to-many communication
- Not sending the same message to all





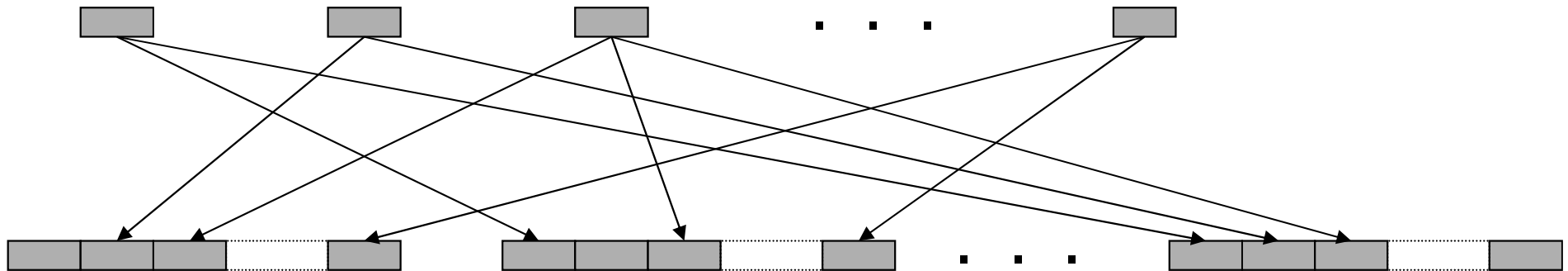
# Gather

- Many-to-one communication
- Not sending the same message to the root



# Gather-to-all

- Many-to-many communication
- Each process sends the same message to all
- Different Processes send different messages



```
...  
MPI_Allgather(x, 100, MPI_INT, y, 100, MPI_INT, MPI_COMM_WORLD)  
...
```

Send buffer

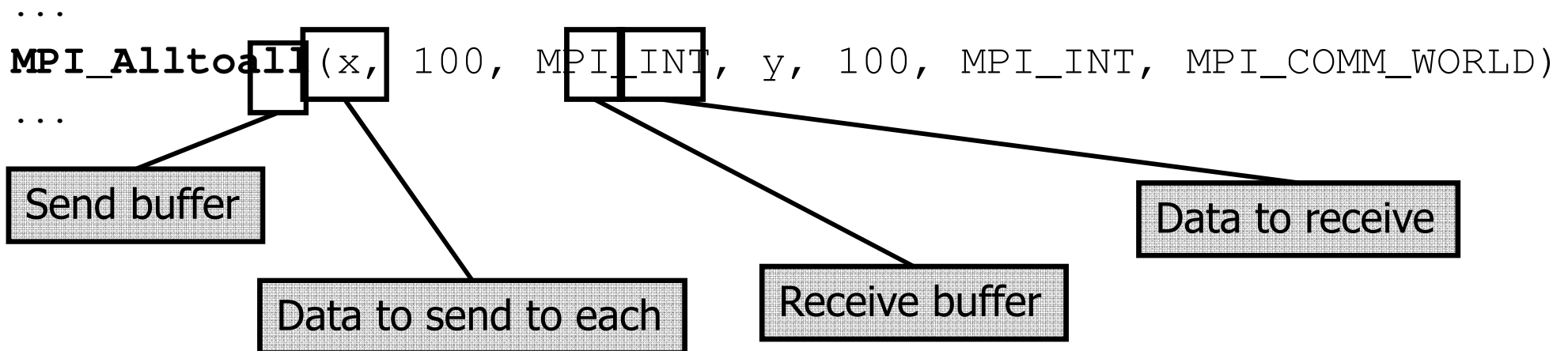
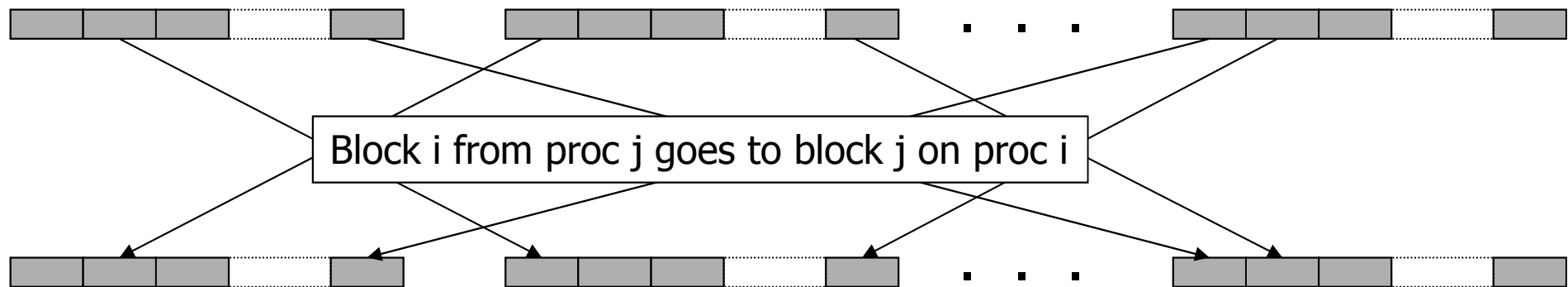
Data to send to each

Receive buffer

Data to receive

# All-to-all

- Many-to-many communication
- Each process sends a different message to each other process





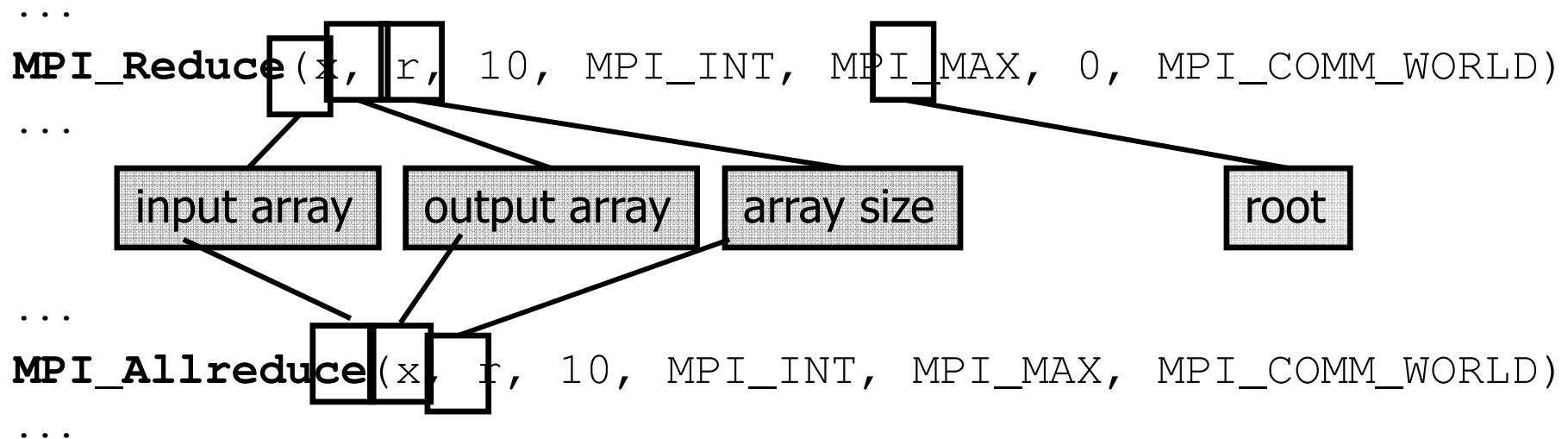
# Reduction Operations

---

- Used to compute a result from data that is distributed among processors
  - often what a user wants to do anyway
  - so why not provide the functionality as a single API call rather than having people keep re-implementing the same things
- Predefined operations:
  - `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, etc.
- Possibility to have user-defined operations

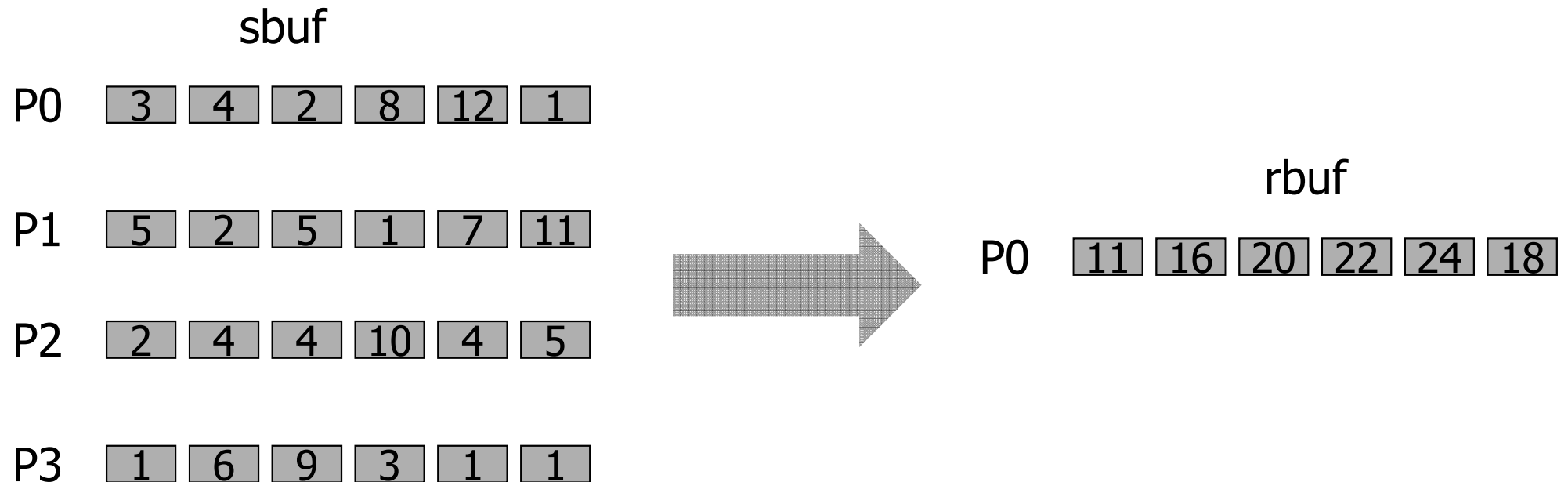
# MPI\_Reduce, MPI\_Allreduce

- MPI\_Reduce: result is sent out to the root
  - the operation is applied element-wise for each element of the input arrays on each processor
- MPI\_Allreduce: result is sent out to everyone



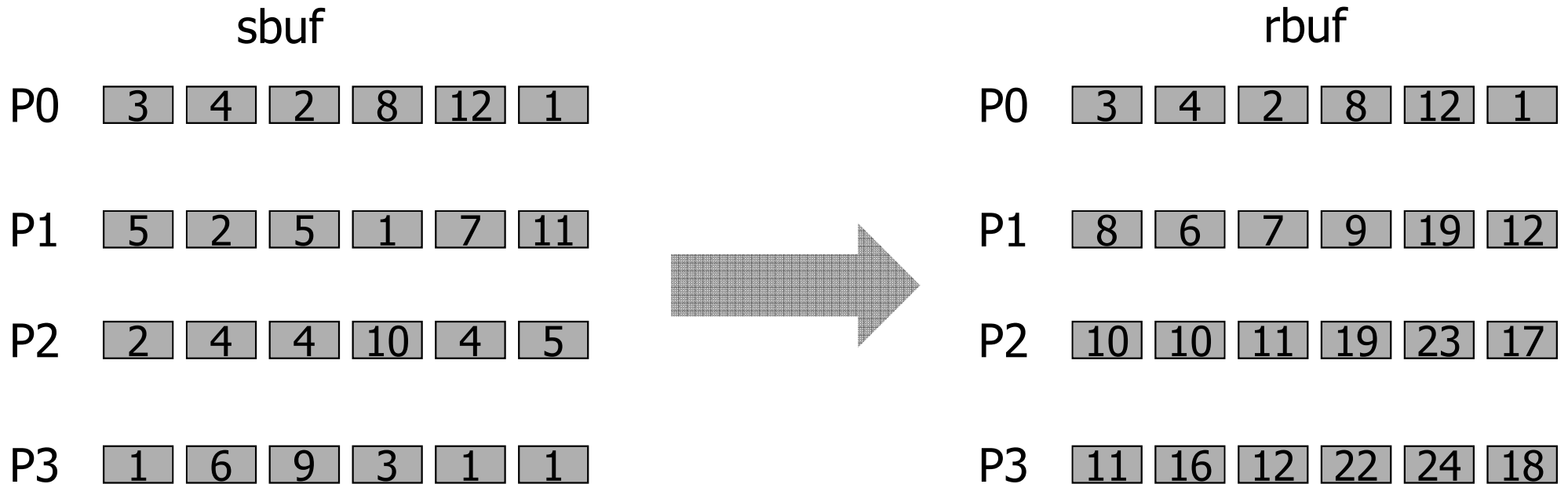
# MPI\_Reduce example

**MPI\_Reduce** (sbuf, rbuf, 6, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD)



# MPI\_Scan: Prefix reduction

- process  $i$  receives data reduced on process 0 to  $i$ .



```
MPI_Scan(sbuf, rbuf, 6, MPI_INT, MPI_SUM, MPI_COMM_WORLD)
```

# User-defined reduce operations

```
MPI_Op_create(MPI_User_function  
*function,  
               int commute, MPI_Op *op)
```

- pointer to a function with a specific prototype
- commute (0 or 1) allows for optimization if true

```
typedef void MPI_User_function(void *a,  
void *b, int *len, MPI_Datatype  
*datatype);
```

- $b[i] = a[i] \text{ op } b[i]$ , for  $i=0, \dots, \text{len}-1$



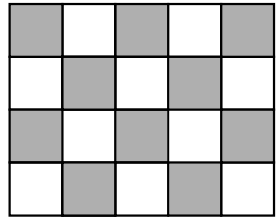
# MPI\_Op\_create example

```
void myfunc(void *a, void *b, int *len, MPI_Datatype
    *dtype) {
    int i;
    for (i = 0; i < *len; ++i) ((int*)b)[i] =
        ((int*)b)[i] + ((int*)a)[i];
}

int main(int argc, char *argv[]) {
    int myrank, nprocs, sendbuf, recvbuf;
    MPI_Op myop;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Op_create(myfunc, 1, &myop);
    sendbuf = 2*myrank+1; // odd
    numbers
    MPI_Reduce(&sendbuf, &recvbuf, 1, MPI_INT, myop, 0,
    MPI_COMM_WORLD);
    if(myrank == 0) printf("%d^2 = %d\n", nprocs,
```

# More Advanced Messages

- Regularly strided data



Blocks/Elements of a matrix

- Data structure

```
struct {  
    int a;  
    double b;  
}
```

- A set of variables

```
int a; double b; int x[12];
```

# Derived Data Types

- A data type is defined by a “type map”
  - set of <type, displacement> pairs
- Created at runtime in two phases
  - Construct the data type from existing types
  - Commit the data type before it can be used
- Simplest constructor: contiguous type

```
int MPI_Type_contiguous(int count,  
                        MPI_Datatype oldtype,  
                        MPI_Datatype *newtype)
```

# MPI\_Type\_contiguous example

```
int buffer[100];
```

```
MPI_Datatype chvec;
```

```
MPI_Type_contiguous(20, MPI_CHAR,  
    &chvec);
```

```
MPI_Type_commit(&chvec);
```

```
...
```

```
MPI_Send(buffer, 1, chvec, 1, 44, MPI_COMM_WOR  
LD);
```

```
MPI_Type_free(&chvec);
```

# MPI\_Type\_indexed()

```
int MPI_Type_indexed(int count,  
int *array_of_blocklengths,  
int *array_of_displacements,  
MPI_Datatype oldtype,  
MPI_Datatype *newtype)
```



# MPI\_Type\_indexed example

```
int vector[4][3] = { 11, 12, 13, 21, 22, 23, 31, 32, 33,
41, 42, 43 };
int wvector[4][3] = { 0 };
int blocklengths[2] = {2, 2};
int displacements[2] = {4, 10}; int rank;
MPI_Datatype mytype;
MPI_Status mystatus;
MPI_Init(&argc, &argv);
MPI_Type_indexed(4, blocklengths, displacements, MPI_INT,
&mytype);
MPI_Type_commit(&mytype);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) MPI_Send(vector, 1, mytype, 1, 0,
MPI_COMM_WORLD);
else {
    MPI_Recv(wvector, 1, mytype, 0, 0, MPI_COMM_WORLD,
&mystatus);
    for (i = 0; i < 4; i++) { printf("\n");
        for (j=0; j < 3; j++) printf("%02d ",
vector[i][j]);
    }
```

# MPI\_Type\_struct()

```
int MPI_Type_struct(int count,  
    int *array_of_blocklengths,  
    MPI_Aint *array_of_displacements,  
    MPI_Datatype *array_of_types,  
    MPI_Datatype *newtype)
```

MPI_INT		MPI_DOUBLE		My_weird_type
---------	--	------------	--	---------------

# MPI\_Type\_vector example

- Sending the 5th column of a 2-D matrix:

```
double results[IMAX][JMAX];  
MPI_Datatype newtype;  
MPI_Type_vector (IMAX, 1, JMAX, MPI_DOUBLE, &newtype);  
MPI_Type_Commit (&newtype);  
MPI_Send(&(results[0][5]), 1, newtype, dest, tag, comm);
```

