Parallel Architectures and Programming (PAP)

MPI – Message Passing Interface

Eduard Ayguadé
(eduard@ac.upc.edu)

Computer Architecture Department
Universitat Politècnica de Catalunya

2019/20-Spring

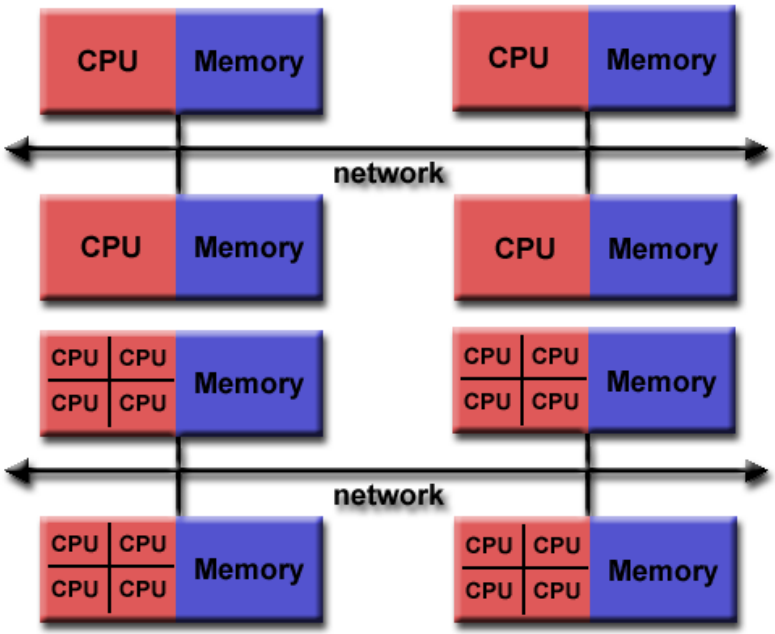
Outline

What is MPI?

- Message Passing Interface
  - Standard application programmer Interface (API) definition
  - Managed by an independent consortium of 40+ members
  - Not an ISO or IEEE standard, just a "de facto" standard
- Multiple implementations
  - OpenMPI
  - MPICH
  - MVAPICH

Architecture model

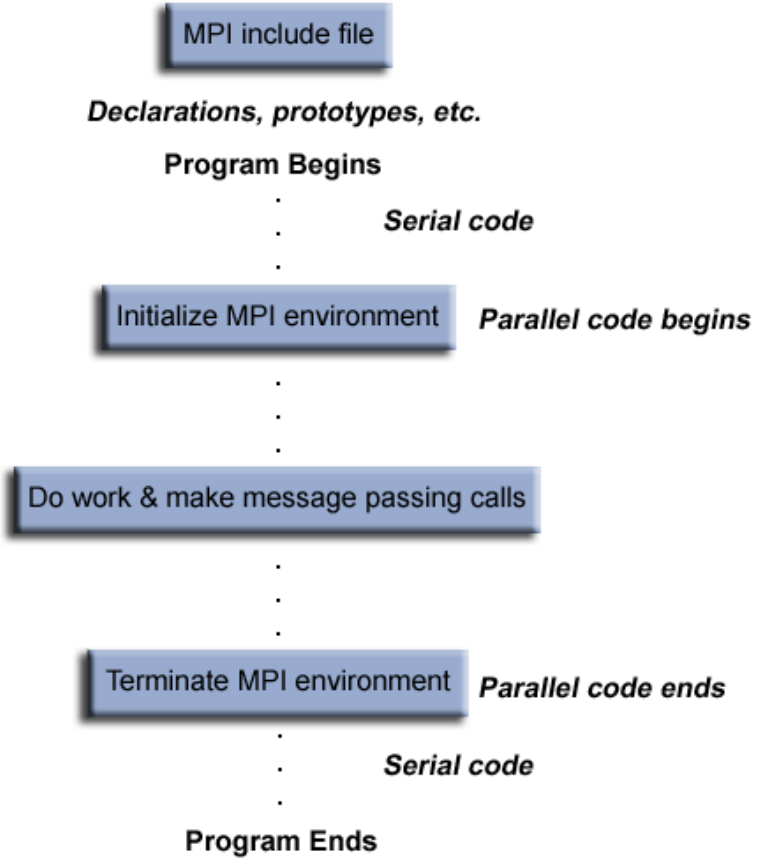
- Distributed memory system
  - Node: CPU(s) + memory + network Interface
  - Multiple nodes: distributed memory, interconnection network



- Multiple processes
  - Shared nothing: every process has a separate logical address space
  - However, it can still run on shared memory systems

Outline

General MPI program structure



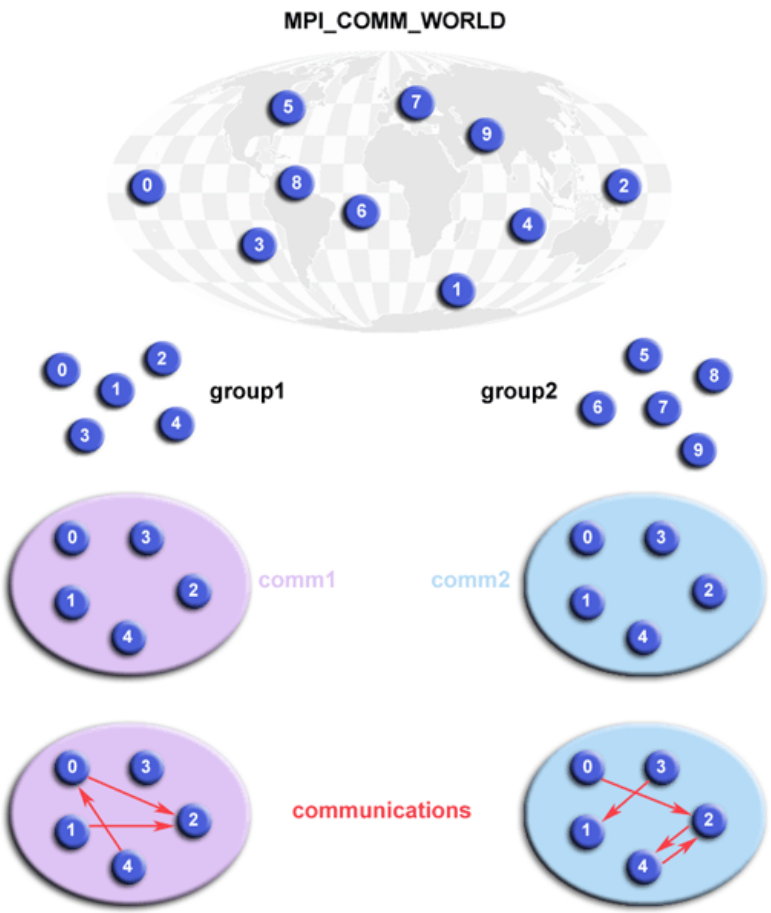
Initialize and terminate

- MPI_Init
  - Initializes the MPI execution environment. This function must be called before any other MPI functions and must be called only once in an MPI program
  - `MPI_Init (&argc, &argv)`
- MPI_Finalize
  - Terminates the MPI execution environment. This function should be the last MPI routine called, no other MPI routines may be called after it
  - `MPI_Finalize ()`

MPI communicators

- Collection (group) of processes working together and communicating
  - Default MPI communicator: `MPI_COMM_WORLD` which includes all the processes in the MPI program
  - MPI allows you to dynamically organize processes into groups with an associated communicator: they can be created and destroyed during program execution
- Most MPI routines require you to specify a communicator as an argument
  - Messages are always exchanged within the context of a communicator
- In this course we do not cover MPI routines related to groups and communicators, only `MPI_COMM_WORLD`

MPI communicators (cont.)



"How many" and "who am I"

- MPI_Comm_size
  - Returns the total number of MPI processes in the specified communicator
  - `MPI_Comm_size (comm, &size)`
- MPI_Comm_rank
  - Returns the rank of the calling MPI process within the specified communicator (unique integer between 0 and number of tasks - 1 within the communicator)
  - `MPI_Comm_rank (comm, &rank)`

Example: Hello World!

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

int main (int argc, char *argv[]) {
int   numtasks, taskid;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);

printf ("Hello world! Task %d/%d\n", taskid, numtasks);

MPI_Finalize();
}
```

A couple more calls ...

- MPI_Abort
  - Terminates all MPI processes associated with the communicator.
  - `MPI_Abort (comm, errorcode)`
- MPI_Get_processor_name
  - Returns the processor name and length of the name.
  - `MPI_Get_processor_name (&name, &resultlength)`
  - The buffer for "name" should be `MPI_MAX_PROCESSOR_NAME` characters in size, since MPI will write up to this many characters into name.

Example: Hello World! with initialization check

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

int main (int argc, char *argv[]) {
int   numtasks, taskid, len;
char hostname[MPI_MAX_PROCESSOR_NAME];

int rc = MPI_Init(&argc, &argv);
if (rc != MPI_SUCCESS) {
     printf ("Error starting MPI program\n");
     MPI_Abort(MPI_COMM_WORLD, rc);
     }
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Get_processor_name(hostname, &len);

printf ("Hello #world! Task %d/%d\n on %s", taskid, numtasks, hostname);

MPI_Finalize();
}
```

Outline

Point–to–point communication

- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks
  - One task is performing a send operation and the other task is performing a matching receive operation
- The simplest point–to–point communication routines are **blocking**
  - A send will only "return" after it is safe to modify the application buffer (your send data) for reuse[1]
  - A receive only "returns" after the data has arrived and is ready for use by the program
- Non-blocking covered later in this chapter

[1] For a few slides assume the sender needs to know the matching receive ...

Blocking message passing routines

- `MPI_Send (&buf, count, datatype, dest, tag, comm)`
  - Sends a message with `count` consecutive elements (of type `datatype`) from application buffer starting at address `&buf`.
- `MPI_Recv (&buf, count, datatype, source, tag, comm, &status)`
  - Receives a message with `count` elements (of type `datatype`) and stores in application buffer starting at address `&buf`.
- Message envelope
  - Information used to distinguish messages and selectively receive them: `<source/dest, tag, comm>`
  - `MPI_ANY_SOURCE` and `MPI_ANY_TAG` may be used to receive a message from any source and/or with any tag. The actual source and tag are returned in `status.MPI_SOURCE` and `status.MPI_TAG`.

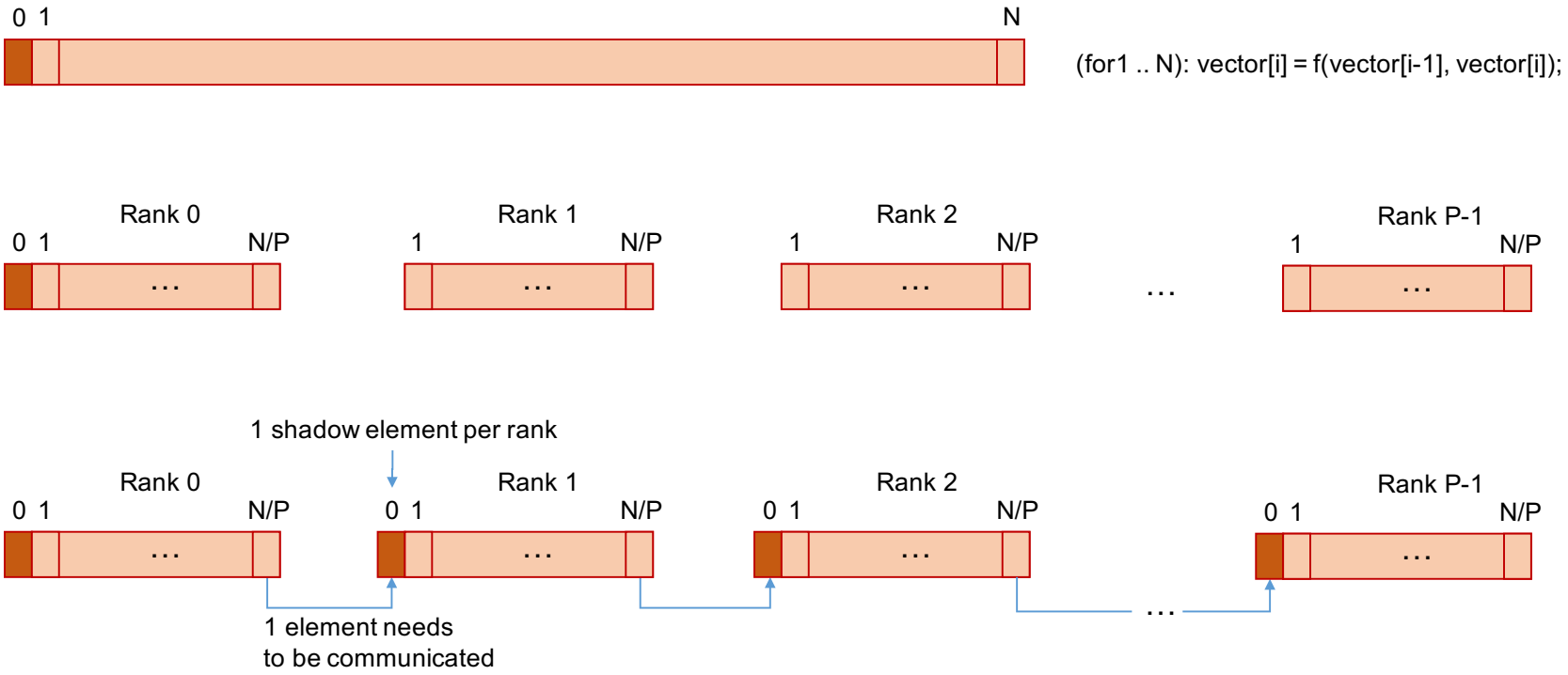
MPI data types

MPI datatype	C equivalent
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	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_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

Example: PI computation

```
#include <mpi.h>
#define NUMSTEPS 100000000

void main (int argc, char *argv[]) {
   int i, rank, procs, num_steps = NUMSTEPS;
   double x, pi, step, sum = 0.0 ;

   MPI_Init(&argc, &argv) ;
   MPI_Comm_Rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_Size(MPI_COMM_WORLD, &procs);

   step = 1.0/(double) num_steps ;
   for (i = rank; i < num_steps; i += procs){
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x*x);
   }
   pi = sum * step;

   if (rank == 0)
       for (i = 1; i < procs; i++) {
           MPI_Recv(&x, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
           pi += x;
           }
   else MPI_Send(&pi, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

   MPI_Finalize() ;
}
```

Example: distributed vector

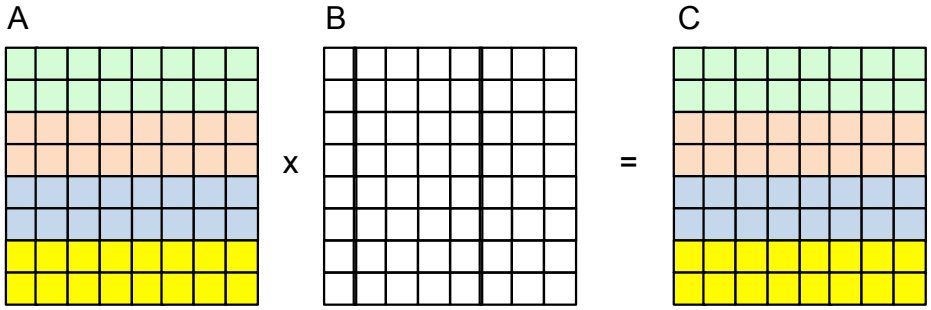


Example: distributed vector

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define N 1024
#define P 4

int main (int argc, char *argv[]) {
int   procs, rank;
double vector[N/P+1];

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &procs);
if (procs != P) {
        printf("Error: number of processors should be %d\n", P);
        MPI_Abort(MPI_COMM_WORLD, 0);
        }
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

random_init(vector, ((rank == 0) ? 0 : 1), N/P);
if (rank < P-1) MPI_Send(&buf[N/P], 1, MPI_DOUBLE, (rank+1), INIT, MPI_COMM_WORLD);
if (rank > 0) MPI_Recv(&buf[0], 1, MPI_DOUBLE, (rank-1), INIT, MPI_COMM_WORLD,
                           MPI_STATUS_IGNORE);
compute(vector, 1, N/P);

MPI_Finalize();
}
```

Example: matrix multiply in MPI

```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE])
{
   for (int i=0; i<MATSIZE; i++)
      for (int j=0; j<MATSIZE; j++)
         for (int k=0; k<MATSIZE; k++)
            C[i][j] += A[i][k]*B[k][j];
}
```



A and C are initially distributed by rows ($MATSIZE/nproc$ rows per process) and B is replicated. Rank 0 will initially and finally store all matrices.

Example: matrix multiply in MPI (cont.)

```
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpiRank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpiSize);
    ...
    n = MATSIZE;
    n_local  = getRowCount(n, mpiRank, mpiSize);
    n_sq     = n * n;
    n_sq2    = n * n_local;
    ...
    A = (double *) malloc(sizeof(double) * (mpiRank ? n_sq2 : n_sq));
    B = (double *) malloc(sizeof(double) *                    n_sq );
    C = (double *) malloc(sizeof(double) * (mpiRank ? n_sq2 : n_sq));
    ...
```

where

```
int getRowCount(int rowsTotal, int mpiRank, int mpiSize) {
    /* Adjust slack of rows in case rowsTotal is not exactly divisible */
    return (rowsTotal / mpiSize) + (rowsTotal % mpiSize > mpiRank);
}
```

Example: matrix multiply in MPI (cont.)

```
...
/* Initialize A and B using some functions */
if (!mpiRank) {
    ReadfromDisk(A, n_sq, 0); /* 0: from beginning; otherwise: from last element read */
    ReadfromDisk(B, n_sq, 0); /* 0: from beginning; otherwise: from last element read */
}

/* Send A by splitting it in row-wise parts */
if (!mpiRank) {
    currentRow = n_sq2;
    for (i=1; i<mpiSize; i++) {
        sizeToBeSent = n * getRowCount(n, i, mpiSize);
        MPI_Send(A + currentRow, sizeToBeSent, MPI_DOUBLE, i, TAG_INIT, MPI_COMM_WORLD);
        currentRow += sizeToBeSent;
    }
}
else { /* Receive parts of A */
    MPI_Recv(A, n_sq2, MPI_DOUBLE, 0, TAG_INIT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
...
```

Example: matrix multiply in MPI (cont.)

```
...
/* Replicate complete B in each process */
if (!mpiRank) {
    for (i=1; i<mpiSize; i++) {
        MPI_Send(B, n_sq, MPI_DOUBLE, i, TAG_INIT, MPI_COMM_WORLD);
    }
}
else { /* Receive B in each other process  */
    MPI_Recv(B, n_sq, MPI_DOUBLE, 0, TAG_INIT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

/* Let each process initialize C to zero */
for (i=0; i<n_sq2; i++)
    C[i] = 0.0;

/* And finally ... let each process perform its own multiplications */
matrixMultiply(A, B, C, n, n_local);

...
```

Example: matrix multiply in MPI (cont.)

```
    ...
    /* Receive partial results from each slave */
    if (!mpiRank) {
        currentRow = n_sq2;
        for (i=1; i<mpiSize; i++) {
            sizeToBeSent = n * getRowCount(n, i, mpiSize);
            MPI_Recv(C + currentRow, sizeToBeSent, MPI_DOUBLE, i, TAG_RESULT, MPI_COMM_WORLD,
                                                   MPI_STATUS_IGNORE);
            currentRow += sizeToBeSent;
        }
    }
    else /* Send partial results to master */
        MPI_Send(C, n_sq2, MPI_DOUBLE, 0, TAG_RESULT, MPI_COMM_WORLD);

    MPI_Finalize();
    ...
```

```
int matrixMultiply(double *a, double *b, double *c, int n, int n_local) {
    for (int i=0; i<n_local; i++)
        for (int j=0; j<n; j++)
            for (int k=0; k<n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
    return 0;
}
```

Blocking message passing routines (cont.)

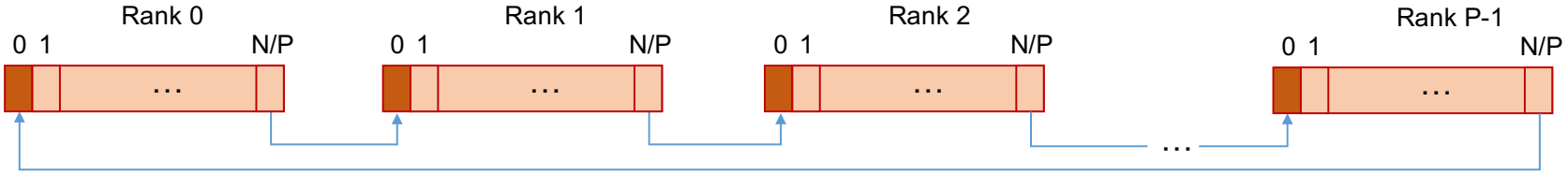
- MPI_Probe
  - Performs a blocking test for a message.
  - `MPI_Probe (source, tag, comm, &status)`
  - The "wildcards" `MPI_ANY_SOURCE` and `MPI_ANY_TAG` may be used to test for a message from any source or with any tag. The actual source and tag will be returned in the status structure as `status.MPI_SOURCE` and `status.MPI_TAG`.
- Example:

```
MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
if (status.MPI_TAG == data)
    {
    MPI_Recv(u, N, MPI_DOUBLE, 0, data, MPI_COMM_WORLD, &status);
    ... // do computation
    }
else {
    MPI_Recv(NULL, 0, MPI_DOUBLE, 0, control, MPI_COMM_WORLD, &status);
    finish = 1;
}
```

Blocking communication and deadlock

- Can deadlock occur due to the blocking nature of blocking communications?
- For example, in the following communication pattern:

```
MPI_Send(&buf[N/P], 1, MPI_DOUBLE, (rank+1)%P, INIT, MPI_COMM_WORLD);
MPI_Recv(&buf[0], 1, MPI_DOUBLE, (rank-1)%P, INIT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



Deadlock: solution 1

Use rank in order to break the cycle. Each process does:

```
if (rank%2) {
   MPI_Recv (&buf[0], 1, MPI_DOUBLE, (rank-1)%P, INIT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   MPI_Send(&buf[N/P], 1, MPI_DOUBLE, (rank+1)%P, INIT, MPI_COMM_WORLD);
}
else {
   MPI_Send(&buf[N/P], 1, MPI_DOUBLE, (rank+1)%P, INIT, MPI_COMM_WORLD);
   MPI_Recv (&buf[0], 1, MPI_DOUBLE, (rank-1)%P, INIT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Deadlock: solution 2

Use MPI_Sendrecv, specially designed for that. Each process does:

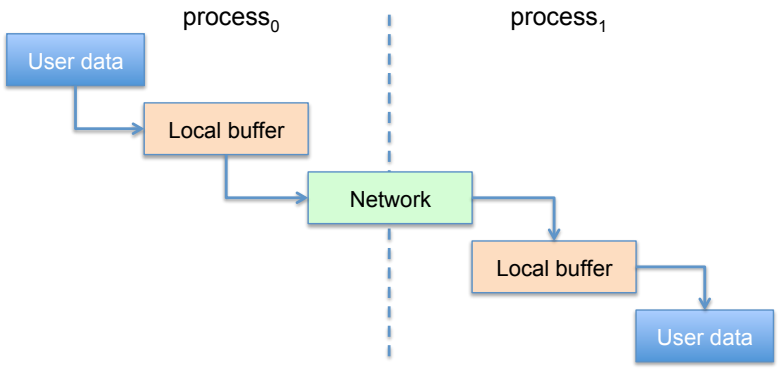
```
MPI_Sendrecv (&buf[N/P], 1, MPI_DOUBLE, (rank+1)%P, INIT,
              &buf[0], 1, MPI_DOUBLE, (rank-1)%P, INIT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Syntax:

```
MPI_Sendrecv(&sendbuf, sendcount, sendtype, dest, sendtag,
             &recvbuf, recvcount, recvtype, source, recvtag,
             comm, &status)
```

Buffering

- Rarely send operations are perfectly synchronized with their matching receives
- The MPI implementation needs to provide some sort of buffering (at sender and/or receiver) to handle this "out of sync" behavior



- On the sender side, once data is in buffer, `MPI_Send` returns
  - What if message to be sent is larger than space available in local buffers?

MPI message delivery

Message = {envelope + data}

- Eager - An asynchronous protocol that allows a send operation to complete without acknowledgement from a matching receive
  - Single exchange of {envelope+data}
  - It is the responsibility of the receiving process to buffer the message upon its arrival if the receive operation has not been posted

MPI message delivery (cont.)

Message = {envelope + data}

- Rendez–vous - A synchronous protocol which requires some type of "handshaking" between the sender and the receiver processes:
  - Sender sends {envelope} to destination process
  - Envelope received and buffered in destination
  - When buffer (for unexpected) is available, destination sends ack of readiness to sender
  - Sender receives ack and sends {data}

MPI message delivery (cont.)

- Eager
  - Cons: Not scalable (significant buffering may be required to provide space for "potential" messages from an arbitrary number of senders)
  - Pros: Reduces synchronization delays
- Rendez–vous
  - Cons: Inherent synchronization delays due to necessary handshaking between sender and receiver
  - Pros: Scalable, only required to buffer envelopes

MPI message delivery (cont.)

protocol	cost
Eager (expected)	lat + (msg + env)/bw
Eager (unexpected)	lat + (msg + env)/bw + copy*msg
Rendezvous (any)	2*(lat + env/bw) + (lat + (msg + env)/bw)

- MPI implementations can use a combination of protocols for the same MPI routine. For example, a standard send might use eager protocol for small messages (`MP_EAGER_LIMIT`), and rendez–vous protocol for larger messages

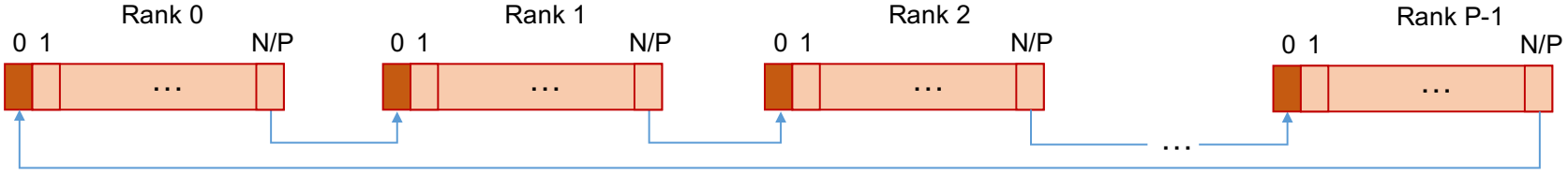
Non–blocking message passing routines

- MPI_Isend
  - Identifies an area in memory to serve as a send buffer. The program should not modify that area until subsequent calls to `MPI_Wait` or `MPI_Test` indicate that the non-blocking send has completed
  - `MPI_Isend (&buf, count, datatype, dest, tag, comm, &request)`
- MPI_Irecv
  - Identifies an area in memory to serve as a receive buffer. The program must use calls to `MPI_Wait` or `MPI_Test` to determine when the requested message is available in that area
  - `MPI_Irecv (&buf, count, datatype, source, tag, comm, &request)`

Non–blocking message passing routines (cont.)

- MPI_Wait
  - Blocks until a specified non-blocking send or receive operation has completed (handle `request`)
  - `MPI_Wait (&request, &status)`
- MPI_Test
  - Checks the status of a specified non-blocking send or receive operation: `flag=1` if the operation has completed, 0 otherwise
  - `MPI_Test (&request, &flag, &status)`
- For multiple non-blocking operations, the programmer can specify any, all or some completions: `MPI_Waitany`, `MPI_Waitall`, `MPI_Waitsome`, `MPI_Testany`, `MPI_Testall` and `MPI_Testsome`

Deadlock: solution 3

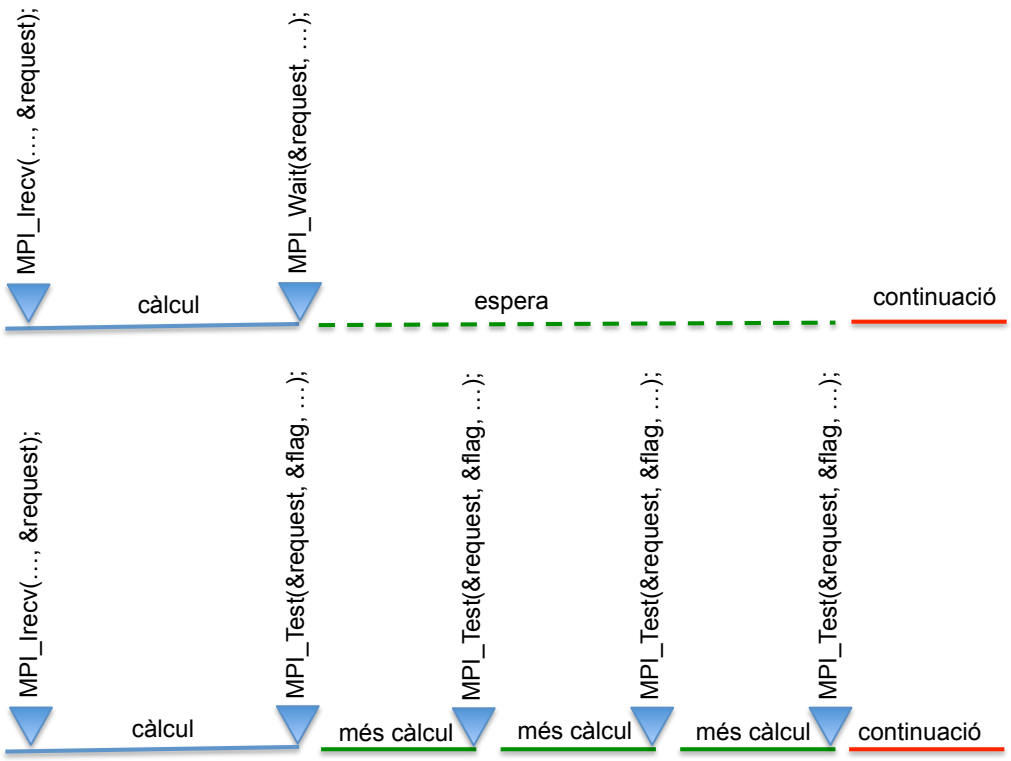


- Non-blocking operations return (immediately) "request handles" that can be tested and waited on

```
MPI_Irecv(&buf[0], 1, MPI_DOUBLE, (rank-1)%P, INIT, MPI_COMM_WORLD, &request);
MPI_Send(&buf[N/P], 1, MPI_DOUBLE, (rank+1)%P, INIT, MPI_COMM_WORLD);
MPI_Wait (&request, &status);
```

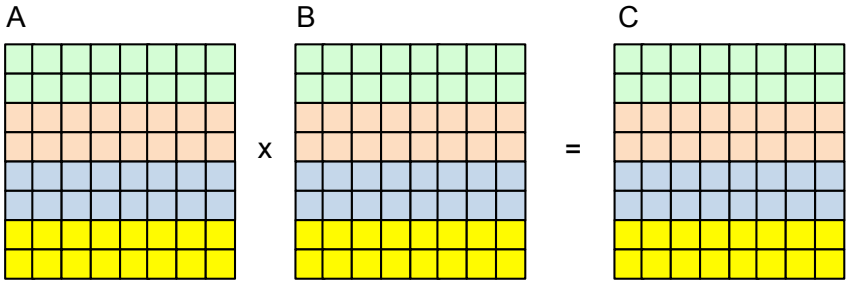
Communication/computation overlap

Non-blocking communications can also be used to overlap computation with communication

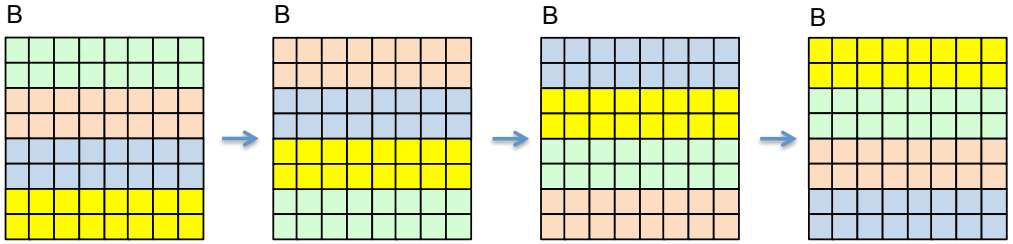


Example: matrix multiply in MPI revisited

- What if B is also distributed by rows?



We need to circulate B among processors in order to do the complete product



Example: matrix multiply in MPI revisited (cont.)

```
...
A = (double *) malloc(sizeof(double) * (mpiRank ? n_sq2 : n_sq));
B = (double *) malloc(sizeof(double) * (mpiRank ? n_sq2 : n_sq));
C = (double *) malloc(sizeof(double) * (mpiRank ? n_sq2 : n_sq));
...
/* Send A and B by splitting it in row-wise parts */
if (!mpiRank) {
    currentRow = n_sq2;
    for (i=1; i<mpiSize; i++) {
        sizeToBeSent = n * getRowCount(n, i, mpiSize);
        MPI_Send(A + currentRow, sizeToBeSent, MPI_DOUBLE, i, TAG_INIT, MPI_COMM_WORLD);
        MPI_Send(B + currentRow, sizeToBeSent, MPI_DOUBLE, i, TAG_INIT, MPI_COMM_WORLD);
        currentRow += sizeToBeSent;
    }
}
else { /* Receive parts of A and B */
    MPI_Recv(A, n_sq2, MPI_DOUBLE, 0, TAG_INIT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(B, n_sq2, MPI_DOUBLE, 0, TAG_INIT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
...
matrixMultiply(A, B, C, n, n_local, mpiRank, mpiSize);
...
```

Example: matrix multiply in MPI revisited (cont.)

```
/* Warning: This solution assumes that n%mpiSize=0 */

void matrixMultiply(double *a, double *b, double *c, int n, int n_local,
                    int mpiRank, int mpiSize) {
    int n_sq2 = n_local*n;
    double *b2 = (double *) malloc(sizeof(double) * n_sq2);
    double *aux;

    int k_ini = mpiRank * n_local;
    for (int p=0; p<mpiSize; p++) {
        for (int i=0; i<n_local; i++)
            for (int j=0; j<n; j++)
                for (int k=0; k<n_local; k++)
                    c[i*n + j] += a[i*n + k+k_ini] * b[k*n + j];

        // Exchange B rows
        int dest = mpiRank?(mpiRank-1):mpiSize-1;
        int src = (mpiRank+1)%mpiSize;
        MPI_Sendrecv (b, n_sq2, MPI_DOUBLE, dest, 0,
                      b2, n_sq2, MPI_DOUBLE, src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        aux = b; b = b2; b2 = aux;
        k_ini = (k_ini + n_local)%n;
    }
}
```

Example: matrix multiply in MPI revisited (cont.)

```
void matrixMultiply(double *a, double *b, double *c, int n, int n_local,
                    int mpiRank, int mpiSize) {
    MPI_Request req[2];
    int n_sq2 = n_local*n;
    double *b2 = (double *) malloc(sizeof(double) * n_sq2);
    double *aux;

    int k_ini = mpiRank * n_local;
    for (int p=0; p<mpiSize; p++) {
            // Exchange B rows
        int dest=mpiRank?(mpiRank-1):mpiSize-1;
        int src=(mpiRank+1)%mpiSize;
        MPI_Isend(b, n_sq2, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD, &req[0]);
        MPI_Irecv(b2, n_sq2, MPI_DOUBLE, src, 0, MPI_COMM_WORLD, &req[1]);

        for (int i=0; i<n_local; i++)
           for (int j=0; j<n; j++)
                for (int k=0; k<n_local; k++)
                    c[i*n + j] += a[i*n + k+k_ini] * b[k*n + j];

        MPI_Wait(&req[0], MPI_STATUS_IGNORE);
        MPI_Wait(&req[1], MPI_STATUS_IGNORE);

        aux = b; b = b2; b2 = aux;
        k_ini = (k_ini + n_local)%n;
    }
}
```

Message ordering

- MPI guarantees sequential consistency within one sender
  - Sender sends two messages to the same destination
  - Both match the same receive
  - The receive operation will receive Message 1 before Message 2
- MPI guarantees sequential consistency within one receiver
  - Receiver posts two receives, both looking for the same message
  - Receive 1 will receive the message before Receive 2
- Order rules do not apply if there are multiple tasks participating in the communication operations

Sender-receiver synchronization

- When sender and receiver process serve communications?
- Polling Mode
  - The user MPI task will be stopped by the system to check for and service communication events at regular (implementation defined) intervals. If a communication event occurs while the user task is busy doing other work, it must wait
- Interrupt Mode
  - The user MPI task will be interrupted by the system for communication events when they occur
  - Usually higher overhead but more responsive

Outline

MPI collectives

- Collective communication routines involve all processes within the scope of a communicator
- Types of collective operations
  - Synchronization - processes wait until all members of the group have reached the synchronization point
  - Data Movement - broadcast, scatter/gather, all to all
  - Collective Computation (reductions) - one member of the group collects data from other members and performs an operation (min, max, add, multiply, etc.) on that data

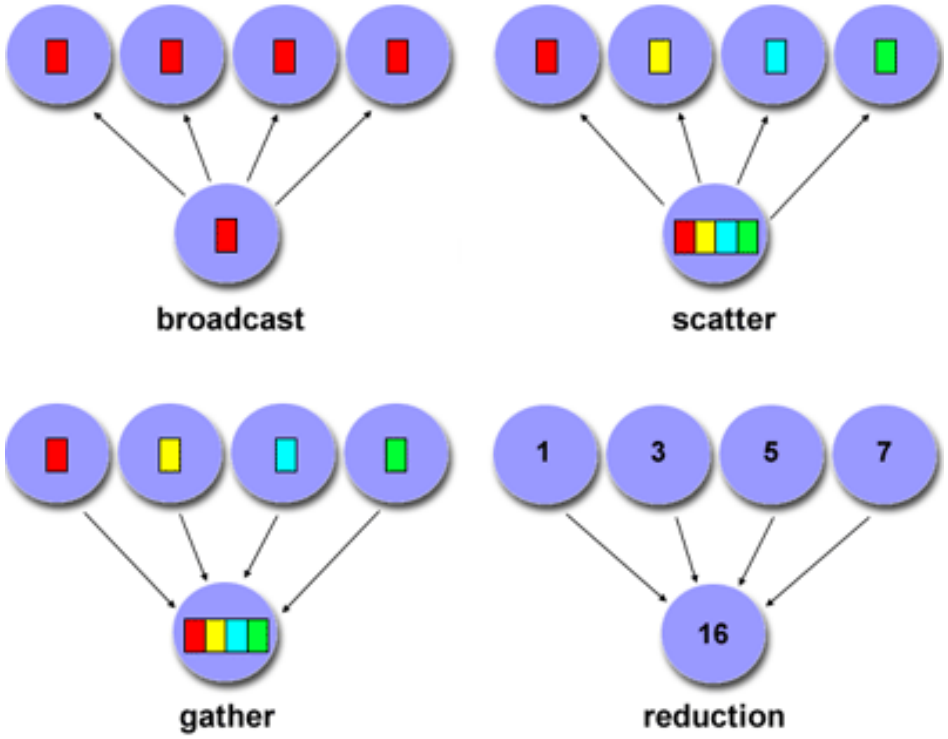
MPI barrier

- Creates a barrier synchronization in a group
  - Each task, when reaching the `MPI_Barrier` call, blocks until all tasks in the group reach the same `MPI_Barrier` call
  - Then all tasks are free to proceed
  - `MPI_Barrier (comm)`

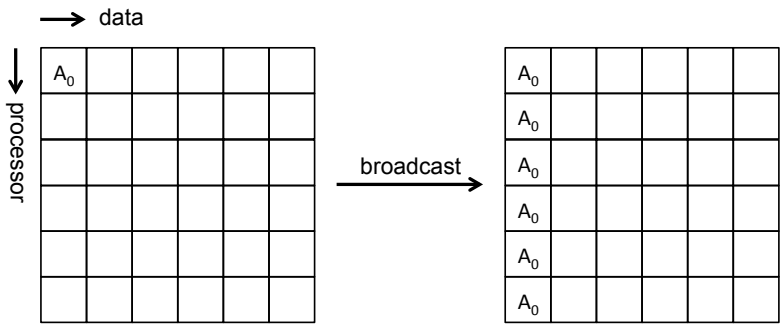
Hello World! with timed useful work afterwards

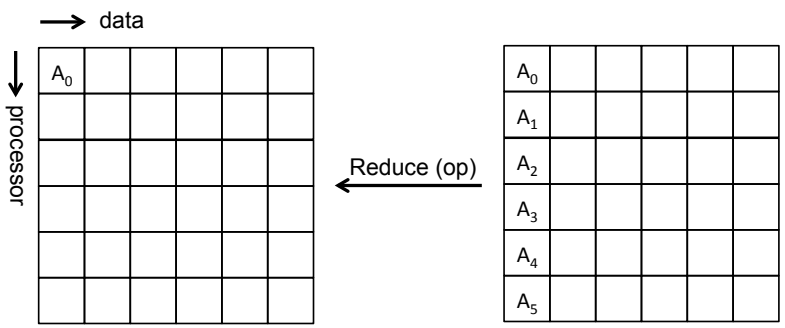
```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
    int   numtasks, taskid;
    double start_time,end_time;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);

    printf ("Hello world! task %d/%d\n", taskid, numtasks);

    // Initialization code here

    MPI_Barrier(MPI_COMM_WORLD); // ensure all tasks start at the same time

    start_time = MPI_Wtime();
    // Do useful work to time here
    end_time = MPI_Wtime();
    printf("Wallclock time elapsed: %.2lf seconds\n",end_time-start_time);

    MPI_Finalize();
    return(0);
}
```

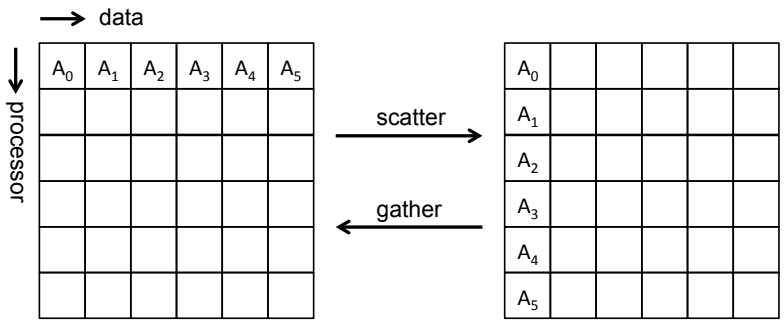
Collective communications

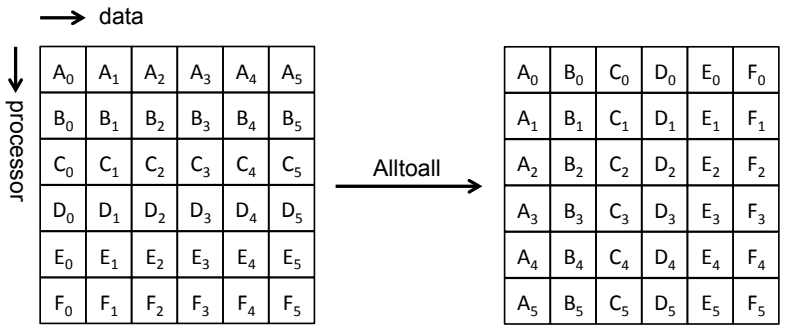


Collective communications









Common collective communications

- MPI_Bcast
  - Broadcasts (sends) a message from the process with rank "root" to all other processes in the group
  - `MPI_Bcast (&buffer, count, datatype, root, comm)`
- MPI_Scatter
  - Distributes distinct messages from a single source task to each task in the group
  - `MPI_Scatter (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)`
- MPI_Gather
  - Gathers distinct messages from each task in the group to a single destination task
  - `MPI_Gather (&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype, root, comm)`

Using collectives in matrix multiply

```
...
/* Replicate complete B in each process */
if (!mpiRank) {
    for (i=1; i<mpiSize; i++) {
        MPI_Send(B, n_sq, MPI_DOUBLE, i, TAG_INIT, MPI_COMM_WORLD);
    }
}
else { /* Receive B in each other process  */
    MPI_Recv(B, n_sq, MPI_DOUBLE, 0, TAG_INIT, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
...
```

Using a single collective:

```
...
/* Replicate complete B in each process */
MPI_Bcast(B, n_sq, MPI_DOUBLE, 0, MPI_COMM_WORLD);
...
```

Using collectives in matrix multiply (cont.)

```
...
/* Send A by splitting it in row-wise parts */
if (!mpiRank) {
    currentRow = n_sq2;
    for (i=1; i<mpiSize; i++) {
        sizeToBeSent = n * getRowCount(n, i, mpiSize);
        MPI_Send(A + currentRow, sizeToBeSent, MPI_DOUBLE, i, TAG_INIT,
                 MPI_COMM_WORLD);
        currentRow += sizeToBeSent;
    }
}
else { /* Receive parts of A */
    MPI_Recv(A, n_sq2, MPI_DOUBLE, 0, TAG_INIT, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
...
```

Using a single collective, assuming same number of rows per processor:

```
...
sizeToBeSent = n * n/mpiSize;
MPI_Scatter(A,sizeToBeSent,MPI_DOUBLE,A,sizeToBeSent,MPI_DOUBLE,0,MPI_COMM_WORLD)
...
```

Using collectives in matrix multiply (cont.)

```
...
/* Receive partial results from each slave */
if (!mpiRank) {
    currentRow = n_sq2;
    for (i=1; i<mpiSize; i++) {
        sizeToBeSent = n * getRowCount(n, i, mpiSize);
        MPI_Recv(C + currentRow, sizeToBeSent, MPI_DOUBLE, i, TAG_RESULT,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        currentRow += sizeToBeSent;
    }
}
else /* Send partial results to master */
    MPI_Send(C, n_sq2, MPI_DOUBLE, 0, TAG_RESULT, MPI_COMM_WORLD);
...
```

Using a single collective, assuming same number of rows per processor:

```
...
sizeToBeSent = n * n/mpiSize;
MPI_Gather(C,sizeToBeSent,MPI_DOUBLE,C,sizeToBeSent,MPI_DOUBLE,0,MPI_COMM_WORLD)
...
```

Common collective communications (cont.)

- MPI_Reduce
  - Applies a reduction operation on all tasks in the group and places the result in one task
  - `MPI_Reduce (&sendbuf, &recvbuf, count, datatype, op, root, comm)`
- Predefined operations
  - `MPI_SUM, MPI_PROD`
  - `MPI_MAX, MPI_MIN`
  - `MPI_MAXLOC, MPI_MINLOC`
  - `MPI_LAND, MPI_LOR, MPI_LXOR`
  - `MPI_BAND, MPI_BOR, MPI_BXOR`

Example: PI computation – broadcast and reduce

```
#include <mpi.h>
#define NUMSTEPS 100000000

void main (int argc, char *argv[]) {
   int i, rank, procs, num_steps = NUMSTEPS;
   double x, pi, step, sum = 0.0 ;

   MPI_Init(&argc, &argv) ;
   MPI_Comm_Rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_Size(MPI_COMM_WORLD, &procs);

   step = 1.0/(double) num_steps ;
   for (i = rank; i < num_steps; i += procs){
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x*x);
   }

   sum *= step;
   MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   MPI_Finalize() ;
}
```

Outline

To be presented by some of you! Interface and simple examples of use (Pi, MxM, ...)

Parallel Architectures and Programming (PAP)

MPI – Message Passing Interface

Eduard Ayguadé
(eduard@ac.upc.edu)

Computer Architecture Department
Universitat Politècnica de Catalunya

2019/20-Spring