Parallel Algorithms and Distributed Systems

A.A. 2013/2014
5 ECTS credits, 3 theory(24 hours) + 2 lab(24 hours)

Lecturer: William Spataro
Department of Mathematics and Computer Science - UNICAL
Email: spataro@unical.it
Web: www.mat.unical.it/spataro
Phone: 0984.494875-3691-6464

MPI – Codes, etc

Beowulf Howto

1. modify `/etc/hosts`
2. Export all home directories, in `/etc/exports`

```
/home   node02(rw)
```

For each new user, update:

```
scp /etc/passwd node02:/etc
scp /etc/shadow node02:/etc
scp /etc/group node02:/etc

echo "Enabling ssh communication..."
cp -r /root/.ssh /home/$1
chown -R $1:$2 /home/$1/.ssh
```

Examples in MPI

Compiling and Execution
Ver 1

```
mpicc –o program_name program_name.c
```

Or, in panic:

```
cc –o program_name program_name.c
  –I/usr/local/mpich-1.2.5/include
  –L/usr/local/mpich-1.2.5/lib
```

To execute on n processes:

```
mpirun –np n program_name
```

Compiling and Execution
Ver 2

```
mpicc –o program_name program_name.c
```

Or, in panic:

```
cc –o program_name program_name.c
  –I/usr/local/mpich2-1.2/include
  –L/usr/local/mpich2-1.2/lib
```

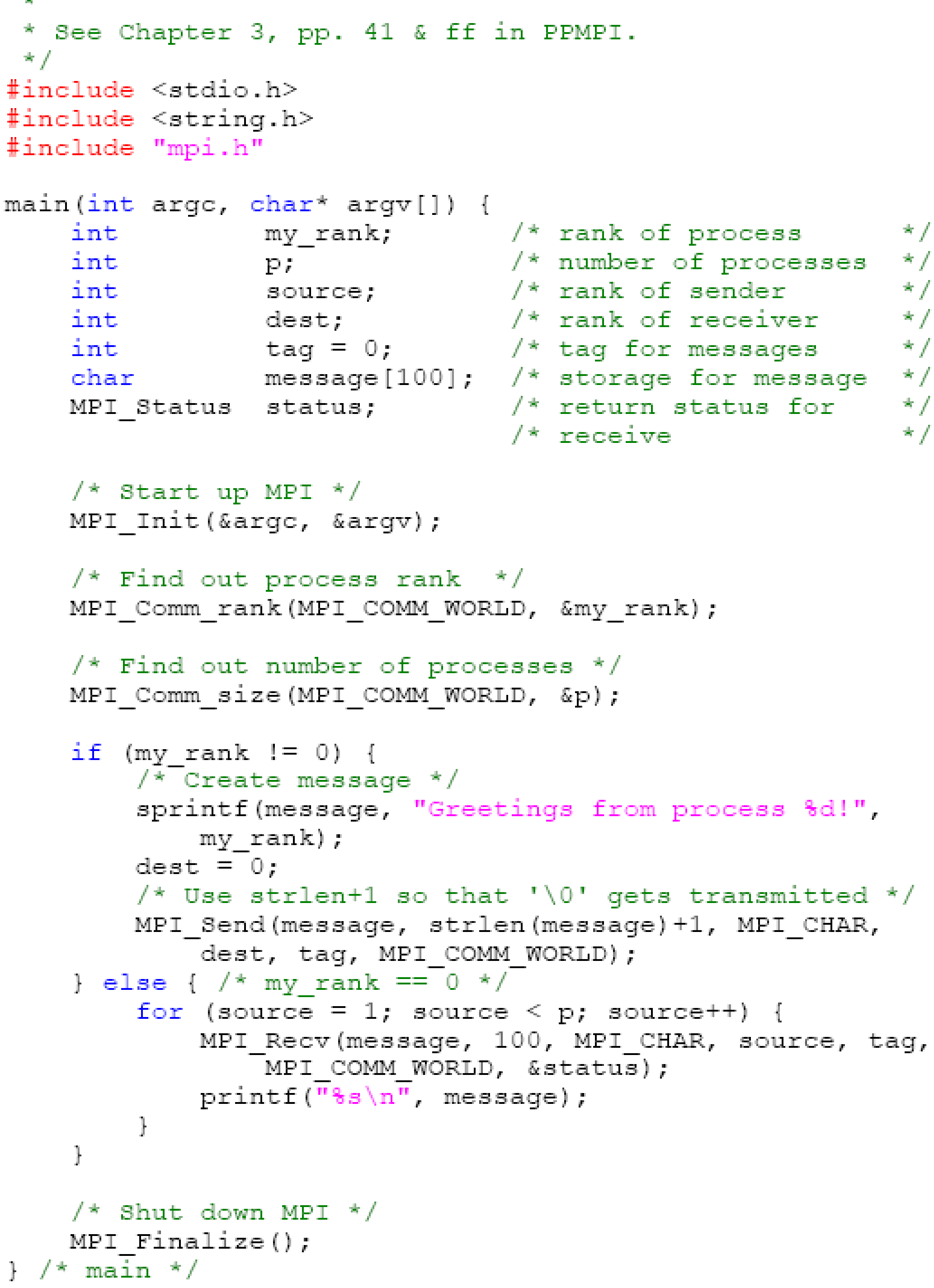
To execute on n processes:

```
  mpd &  // only at the beginning of the session
  mpiexec -n num_proc program_name
  mpdallexit // at the end of the session
```

Debugging tips

- Run the program with one process just like a normal sequential program
- Run the program on 2-4 processes. Check sending of messages (correct recipient, tags, etc.)
- Run the program on 2-4 processors

Hello World!



```
 * See Chapter 3, pp. 41 & ff in PPMPI.
 */
#include <stdio.h>
#include <string.h>
#include "mpi.h"

main(int argc, char* argv[]) {
    int         my_rank;       /* rank of process      */
    int         p;             /* number of processes  */
    int         source;        /* rank of sender       */
    int         dest;          /* rank of receiver     */
    int         tag = 0;       /* tag for messages     */
    char        message[100];  /* storage for message  */
    MPI_Status  status;        /* return status for    */
                               /* receive              */

    /* Start up MPI */
    MPI_Init(&argc, &argv);

    /* Find out process rank  */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!",
            my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR,
            dest, tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    /* Shut down MPI */
    MPI_Finalize();
} /* main */
```

A simple ping

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

int main(argc,argv)
int argc;
char *argv[]; {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

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

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
}
else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d \n",
          rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
MPI_Finalize();
}
```

Vector sum (without Broadcast and Reduce)

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

#define MAXSIZE 10

int main(int argc, char** argv)
{
    int myid, numprocs;
    int data[MAXSIZE], i, x, low, high, myresult, result, result_temp;
    int dest, source;

    MPI_Init(&argc, &argv);
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    result = 0;
    myresult = 0;

    // Inizializzo...(ogni "processo" vedrà la propria porzione inizializzata)
    for (i=0; i<MAXSIZE;i++)
        data[i] = i;
    }

    // Individuo la mia porzione
    x = MAXSIZE/numprocs;
    low = myid * x;
    high = low + x;

    // Calcolo il mio risultato (anche il processo 0 lo fara')
    for (i=low; i<high; i++)
        myresult = myresult + data[i];

     if (myid == 0){
        result = myresult;
        for (source=1; source<numprocs; source++){
            MPI_Recv(&myresult, 1, MPI_INT, source, 0, MPI_COMM_WORLD, &status);
        result = result + myresult;
        }
     }
        else
            MPI_Send(&myresult, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);


    if (myid ==0)
        printf("La somma è %d.\n", result);

    MPI_Finalize();
    exit(0);
}
```

«Global» variables!

Let’s get timings!

- To "take" the execution time of an MPI program we can use the timer function `MPI_Wtime (void)`
- First step: Synchronize all processes via a call to `MPI_Barrier()`

- Get initial time with

```
start=MPI_Wtime();
```

- At the end of the code, call `MPI_Barrier()` to re-synchronize processes
- Take final time with

```
finish=MPI_Wtime();
```

- Let only process 0 print the elapsed time:

```
printf(“Elapsed time is = %f seconds\n”, finish-start);
```

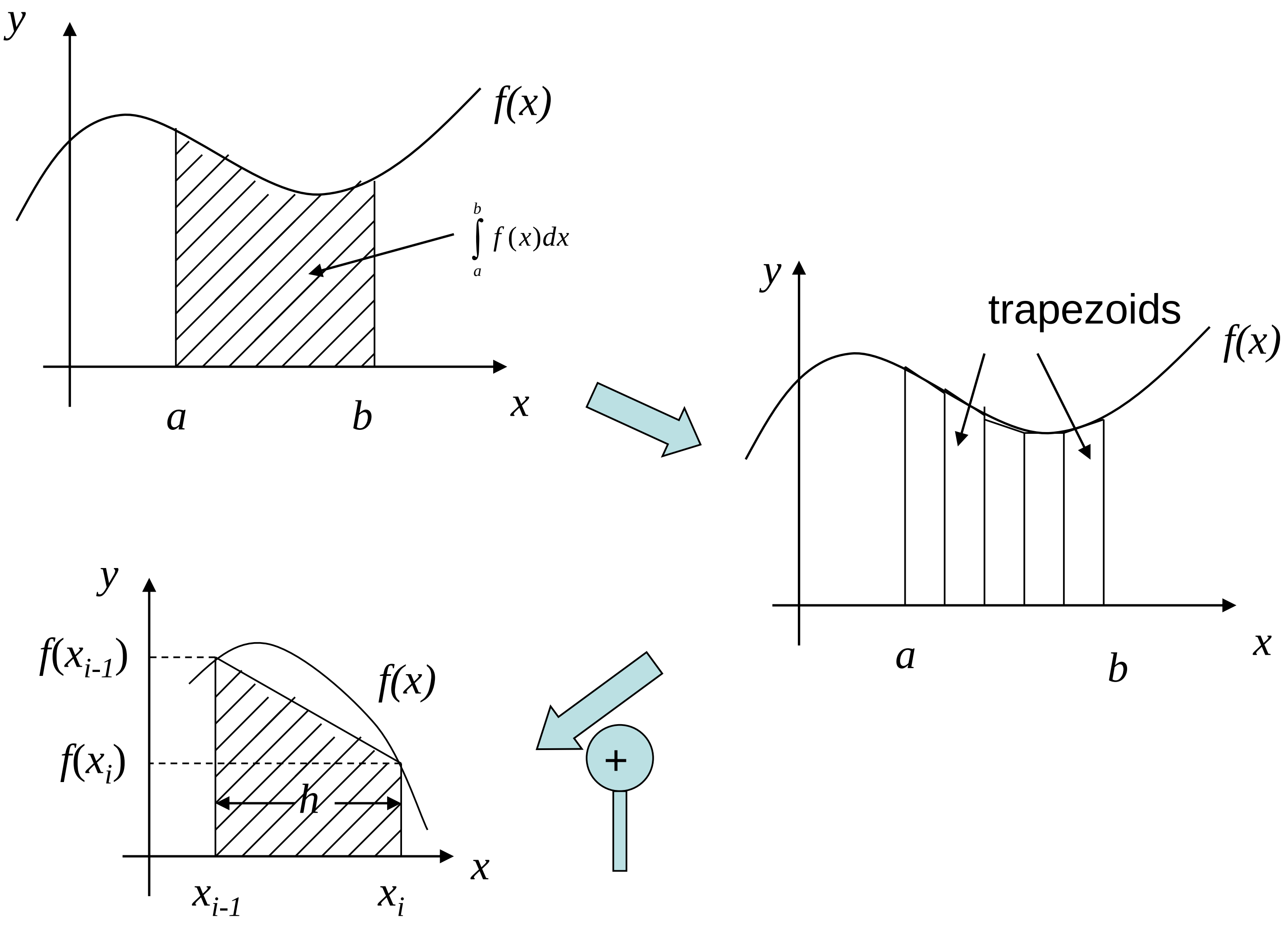
- `MPI_Wtime()` returns the **wall-clock time**, that is includes also system time, etc

Sum of the elements of a vector (dynamic allocation)

Compile with mpiCC!

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

#define MAXSIZE 100000

int main(int argc, char** argv)
{
    int myid, numprocs;
    int x, low, high, result_temp, i;
    int dest, source;
    int *data, *local_data;
    double myresult, result;
    double start, end;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    result = 0;
    myresult = 0;
    // Inizializzo...(ogni "processo" vedrà la propria porzione inizializzata)
    if (myid==0){
        data = new int[MAXSIZE];
        for (i=0; i<MAXSIZE;i++)
            data[i] = i;
    }
    start = MPI_Wtime();
    // Individuo la mia porzione
    x = MAXSIZE/numprocs;
    local_data = new int[x];
    MPI_Scatter(data, x, MPI_INT, local_data, x, MPI_INT, 0, MPI_COMM_WORLD);

    // Calcolo il mio risultato (anche il processo 0 lo fara')
    for (i=0; i<x; i++)
        myresult = myresult + local_data[i];
     if (myid == 0){
        result = myresult;
        for (source=1; source<numprocs; source++){
            MPI_Recv(&myresult, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, &status);
        result = result + myresult;
        }
     }
        else
            MPI_Send(&myresult, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

    end = MPI_Wtime();
    if (myid ==0){
        printf("La somma è %e.\n", result);
        printf("Calcolato in tempo %f millisecs\n", 1000*(end - start));
   }
    MPI_Finalize();
}
```

Example: Numerical Integration



Example: Numerical Integration

- Trapezoid rule
- Each rectangle has base $h=(b-a)/n$
- The trapezium has left-most base $[a, a+h]$, the next $[a+h, a+2h]$, the next $[a+2h, a+3h]$, etc
- Let's denote $x_i=a+ih$, $i=0,...,n$
- So, the left side of each trapezoid is $f(x_{i-1})$, while the right is $f(x_i)$

Example: Numerical Integration

- The area of the i-th trapezoid is $\frac{1}{2} h[f(x_{i-1})+f(x_i)]$

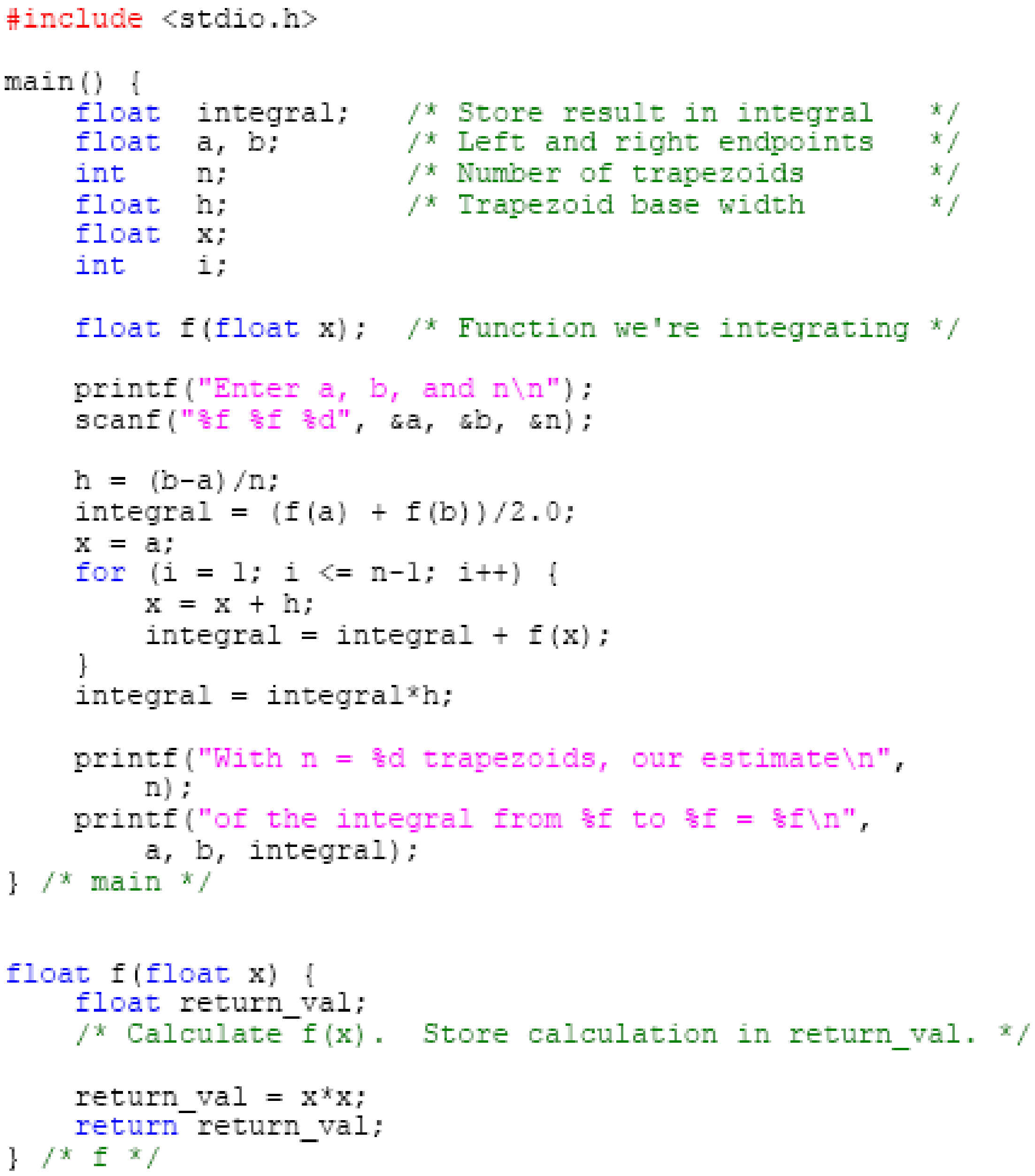
- The approximation of the entire area will be the sum of the area of the trapezoids:

$$\tfrac{1}{2} h[f(x_0)+f(x_1)]+\tfrac{1}{2} h\,[f(x_1)+f(x_2)]+\ldots+\tfrac{1}{2} h\,[f(x_{n-1})+f(x_n)]=$$

$$=h/2[f(x_o)+2f(x_1)+2f(x_2)+\ldots+f(x_n)]=$$

$$=[f(x_o)/2+f(x_n)/2+f(x_1)+f(x_2)+\ldots+f(x_{n-1})]h$$

Sequential version



```c
#include <stdio.h>

main() {
    float  integral;   /* Store result in integral   */
    float  a, b;       /* Left and right endpoints   */
    int    n;          /* Number of trapezoids       */
    float  h;          /* Trapezoid base width       */
    float  x;
    int    i;

    float f(float x);  /* Function we're integrating */

    printf("Enter a, b, and n\n");
    scanf("%f %f %d", &a, &b, &n);

    h = (b-a)/n;
    integral = (f(a) + f(b))/2.0;
    x = a;
    for (i = 1; i <= n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;

    printf("With n = %d trapezoids, our estimate\n",
        n);
    printf("of the integral from %f to %f = %f\n",
        a, b, integral);
} /* main */


float f(float x) {
    float return_val;
    /* Calculate f(x).  Store calculation in return_val. */

    return_val = x*x;
    return return_val;
} /* f */
```

Parallelization

- A possible method is to assign a portion of the interval [a, b] for each process (data parallelism!)

- How does each process know the subinterval and how many trapezoids to use?

- **Natural solution**: the first process computes the first n/p trapezoids, the second the second n/p trapezoids, etc. (where p is the number of processes)

Parallelization

- Therefore, each process needs to know
    - The number of processes, p
    - Their rank
    - The entire integration interval [a, b]
    - The number of subintervals, n

- The first two information are provided by `MPI_Comm_size` and `MPI_Comm_rank`, the latter two should be provided by the user.

- **Last observation**: how are the partial sums added together for each process?

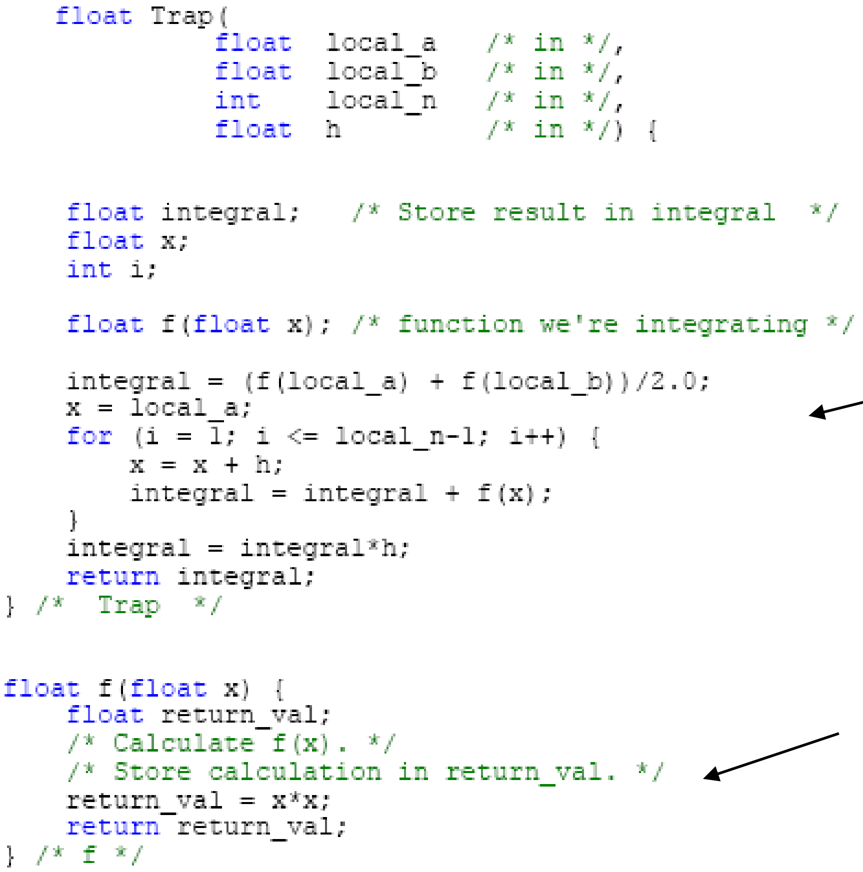
- **Possible Solution**: Each process sends its partial sum to process 0, and this performs the sum.

```c
#include <stdio.h>

/* We'll be using MPI routines, definitions, etc. */
#include "mpi.h"


main(int argc, char** argv) {
    int         my_rank;   /* My process rank           */
    int         p;         /* The number of processes   */
    float       a = 0.0;   /* Left endpoint             */
    float       b = 1.0;   /* Right endpoint            */
    int         n = 1024;  /* Number of trapezoids      */
    float       h;         /* Trapezoid base length     */
    float       local_a;   /* Left endpoint my process  */
    float       local_b;   /* Right endpoint my process */
    int         local_n;   /* Number of trapezoids for  */
                           /* my calculation            */
    float       integral;  /* Integral over my interval */
    float       total;     /* Total integral            */
    int         source;    /* Process sending integral  */
    int         dest = 0;  /* All messages go to 0      */
    int         tag = 0;
    MPI_Status  status;

    float Trap(float local_a, float local_b, int local_n,
              float h);    /* Calculate local integral  */

    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    if (my_rank == 0) {
        total = integral;
        for (source = 1; source < p; source++) {
            MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                MPI_COMM_WORLD, &status);
            total = total + integral;
        }
    } else {
        MPI_Send(&integral, 1, MPI_FLOAT, dest,
            tag, MPI_COMM_WORLD);
    }

    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",
            n);
        printf("of the integral from %f to %f = %f\n",
            a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /*  main  */
```

NB!

Each process (including 0) calculates the sum of the areas of the "local“ trapezoids

Process 0 receives all (it has already its area!)



```
   float Trap(
              float  local_a   /* in */,
              float  local_b   /* in */,
              int    local_n   /* in */,
              float  h         /* in */) {

    float integral;   /* Store result in integral  */
    float x;
    int i;

    float f(float x); /* function we're integrating */

    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
} /*  Trap  */


float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
} /* f */
```

Area computation of local trapezoids

Ex: $f(x)= x^2$

`I/O`

- The function $f(x)$ and the variables `a`, `b` and `n` well are "hardwired“
- $f(x)$ can be defined as a pointer function (or callback function) (home exercise)
- Although not a standard procedure, it is advisable that a process takes care of the I / O (for instance, process 0 sends the initial data, a, b and n to processes)

I/O

If for each process I execute:

```
scanf("%f %f %d", &a, &b, &n);
```

What happens?

If for each process I execute:

```
printf("%f %f %d", a, b, n);
```

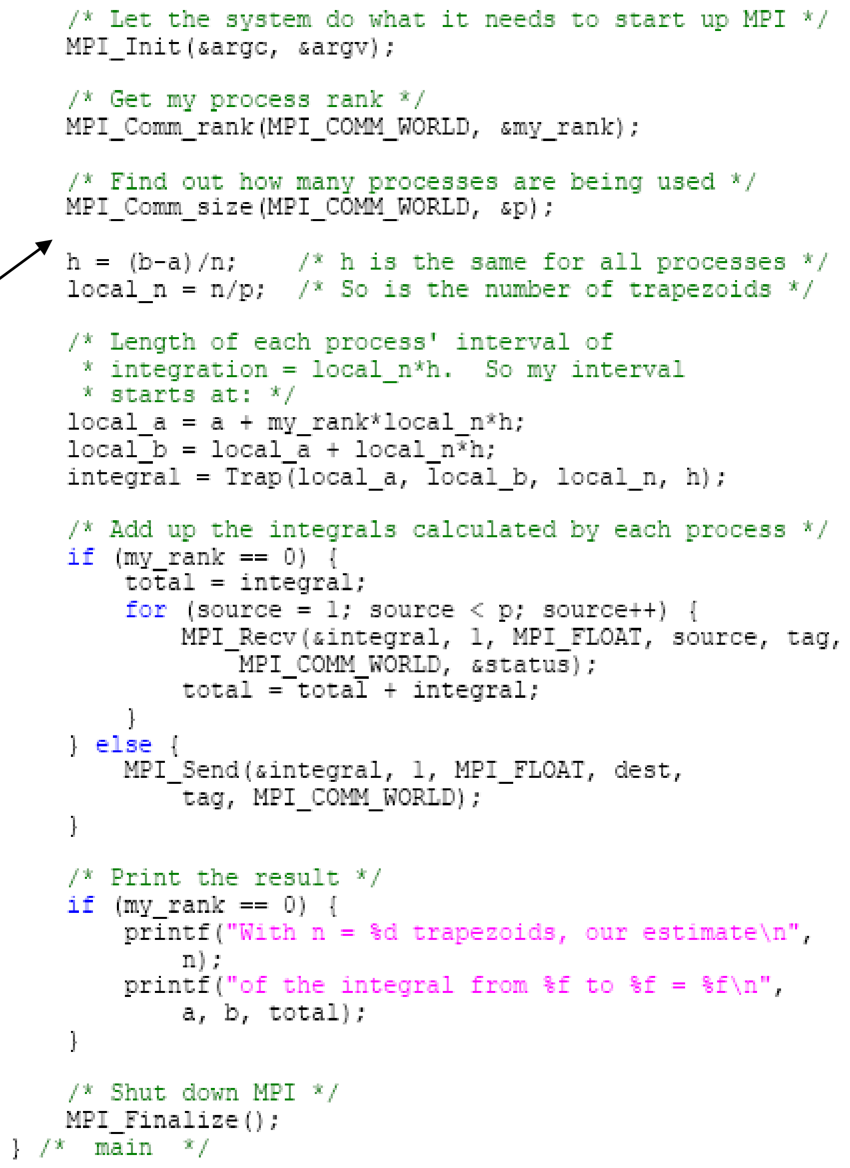
What happens?

```
void Get_data(
         float*  a_ptr    /* out */,
         float*  b_ptr    /* out */,
         int*    n_ptr    /* out */,
         int     my_rank  /* in  */,
         int     p        /* in  */) {

    int source = 0;    /* All local variables used by */
    int dest;          /* MPI_Send and MPI_Recv       */
    int tag;
    MPI_Status status;

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
        for (dest = 1; dest < p; dest++){
            tag = 0;
            MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag,
                MPI_COMM_WORLD);
            tag = 1;
            MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag,
                MPI_COMM_WORLD);
            tag = 2;
            MPI_Send(n_ptr, 1, MPI_INT, dest, tag,
                MPI_COMM_WORLD);
        }
    } else {
        tag = 0;
        MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag,
            MPI_COMM_WORLD, &status);
        tag = 1;
        MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag,
            MPI_COMM_WORLD, &status);
        tag = 2;
        MPI_Recv(n_ptr, 1, MPI_INT, source, tag,
                MPI_COMM_WORLD, &status);
    }
} /* Get_data */
```

Note different tags!!!

Note different tags!!!

Call to `Get_data`



```
    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    if (my_rank == 0) {
        total = integral;
        for (source = 1; source < p; source++) {
            MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                MPI_COMM_WORLD, &status);
            total = total + integral;
        }
    } else {
        MPI_Send(&integral, 1, MPI_FLOAT, dest,
            tag, MPI_COMM_WORLD);
    }

    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",
            n);
        printf("of the integral from %f to %f = %f\n",
            a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /*  main  */
```

Home work ☺

- Vector maximum
- Search of an element in a vector
- Summation of two matrixes
- From Pacheco, Programming Assignment 3.7.1
- From Pacheco, Exercise 4.6.2, Programming Assignment 4.7.1, 4.7.2

Hard Homework ☹

- Matrix – Matrix Product (AxB=C)
- Advice:

```
for (each column x of B){
    Compute parallel dot product matrix-vettor Ax
}
```

Scalar Product

Let

$x=(x_0, x_1, ..., x_{n\text{-}1})^{\mathrm{T}}$

$y=(y_0, y_1, ..., y_{n\text{-}1})^{\mathrm{T}}$

$$x \oplus y = x_0y_0 + x_1y_1 + ... + x_{n\text{-}1}y_{n\text{-}1}$$

Serial

```c
#include <stdio.h>

#define MAX_ORDER 100

main() {
    float  x[MAX_ORDER];
    float  y[MAX_ORDER];
    int    n;
    float  dot;

    void Read_vector(char* prompt, float v[], int n);
    float Serial_dot(float x[], float y[], int n);

    printf("Enter the order of the vectors\n");
    scanf("%d", &n);
    Read_vector("the first vector", x, n);
    Read_vector("the second vector", y, n);
    dot = Serial_dot(x, y, n);
    printf("The dot product is %f\n", dot);
}  /* main */
```

```c
/************************************
void Read_vector(
         char*   prompt   /* in  */,
         float   v[]      /* out */,
         int     n        /* in  */) {
    int i;

    printf("Enter %s\n", prompt);
    for (i = 0; i < n; i++)
        scanf("%f", &v[i]);
}  /* Read_vector */


/************************************
float Serial_dot(
          float  x[]  /* in */,
          float  y[]  /* in */,
          int    n    /* in */) {

    int    i;
    float  sum = 0.0;

    for (i = 0; i < n; i++)
        sum = sum + x[i]*y[i];
    return sum;
} /* Serial_dot */
```

Parallel – Block Mapping

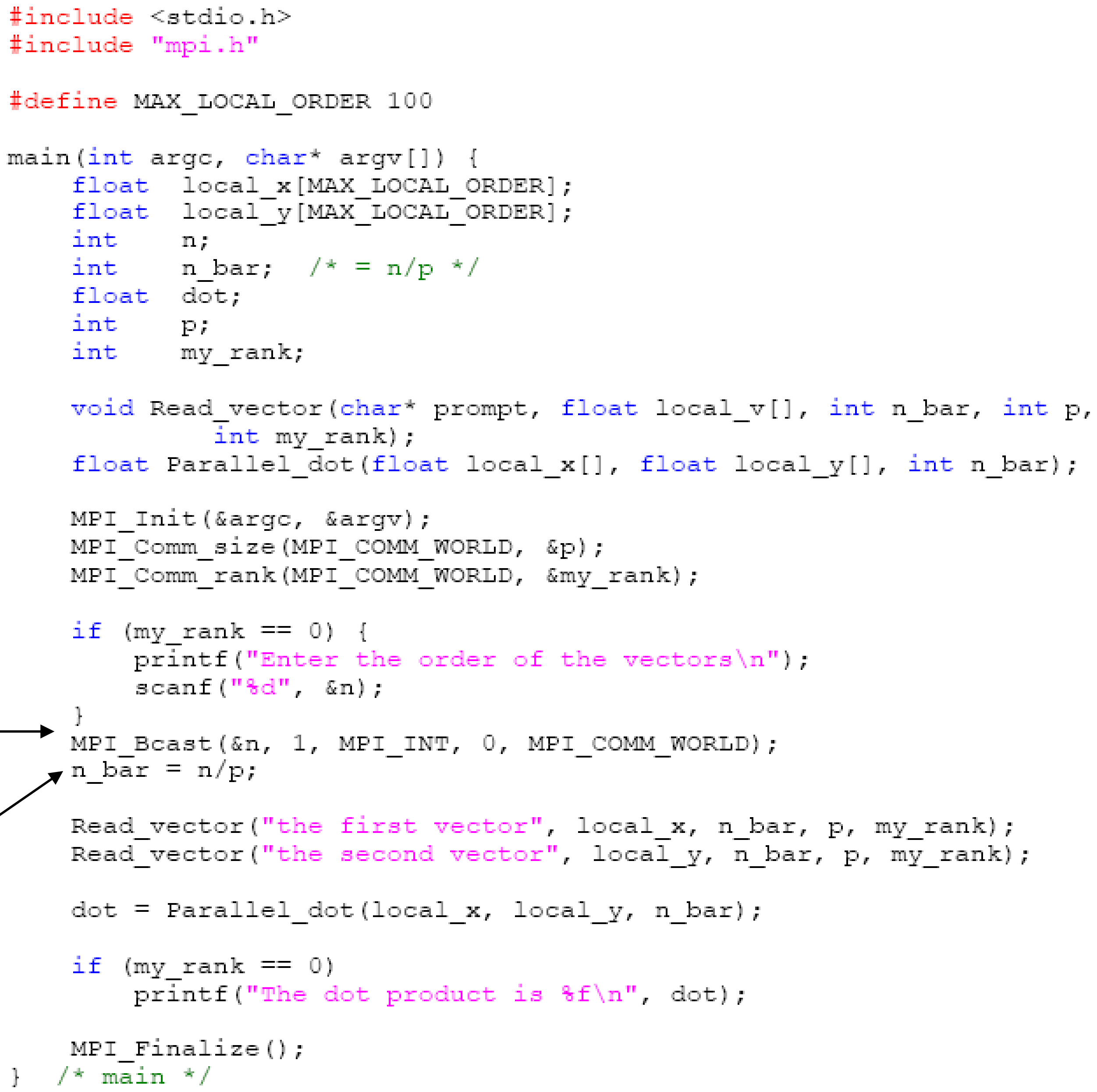
Process	*Components*
0	$x_0, x_1, \ldots, x_{\check{n}-1}$
1	$x_{\check{n}}, x_{\check{n}+1}, \ldots, x_{2\check{n}-1}$
.	.
.	.
k	$x_{k\check{n}}, x_{k\check{n}+1}, \ldots, x_{(k+1)\check{n}-1}$
.	.
.	.
p-1	$x_{(p-1)\check{n}}, x_{(p-1)\check{n}+1}, \ldots, x_{n-1}$

Parallel – Block Mapping

- This allocation "technique" is different than that used, for example, for the "sum of the elements of a vector", where each process **<u>sees</u>** the entire data structure

- In this case, although each process **allocates the entire data structure**, it receives only the portion of data that interests it

Parallel



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

#define MAX_LOCAL_ORDER 100

main(int argc, char* argv[]) {
    float  local_x[MAX_LOCAL_ORDER];
    float  local_y[MAX_LOCAL_ORDER];
    int    n;
    int    n_bar;  /* = n/p */
    float  dot;
    int    p;
    int    my_rank;

    void Read_vector(char* prompt, float local_v[], int n_bar, int p,
             int my_rank);
    float Parallel_dot(float local_x[], float local_y[], int n_bar);

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0) {
        printf("Enter the order of the vectors\n");
        scanf("%d", &n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    n_bar = n/p;

    Read_vector("the first vector", local_x, n_bar, p, my_rank);
    Read_vector("the second vector", local_y, n_bar, p, my_rank);

    dot = Parallel_dot(local_x, local_y, n_bar);

    if (my_rank == 0)
        printf("The dot product is %f\n", dot);

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

“broadcast” **n** to all!

Vector portion **ň**

Broadcast more efficent than multiple sends!

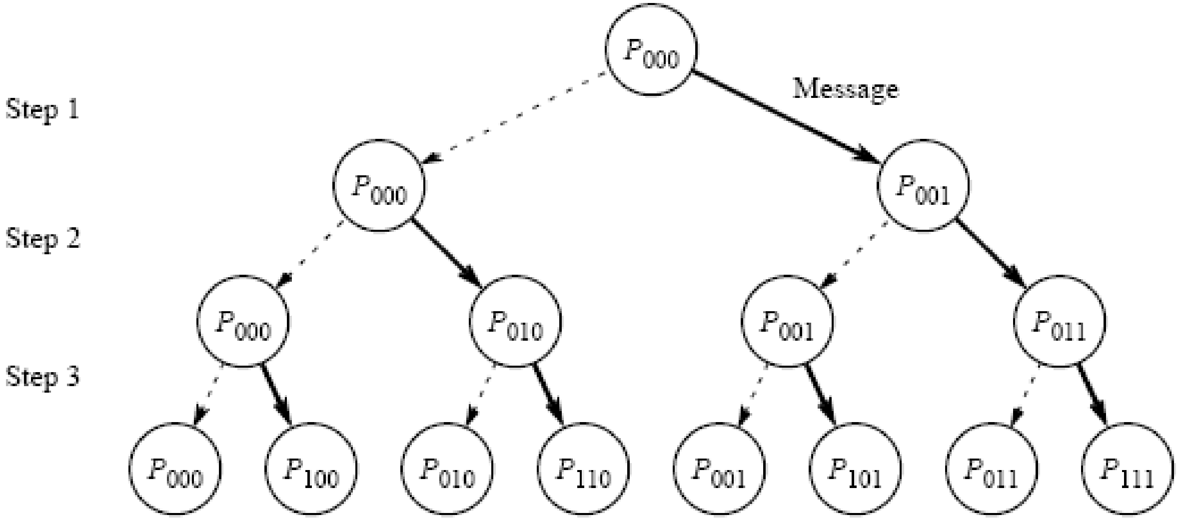
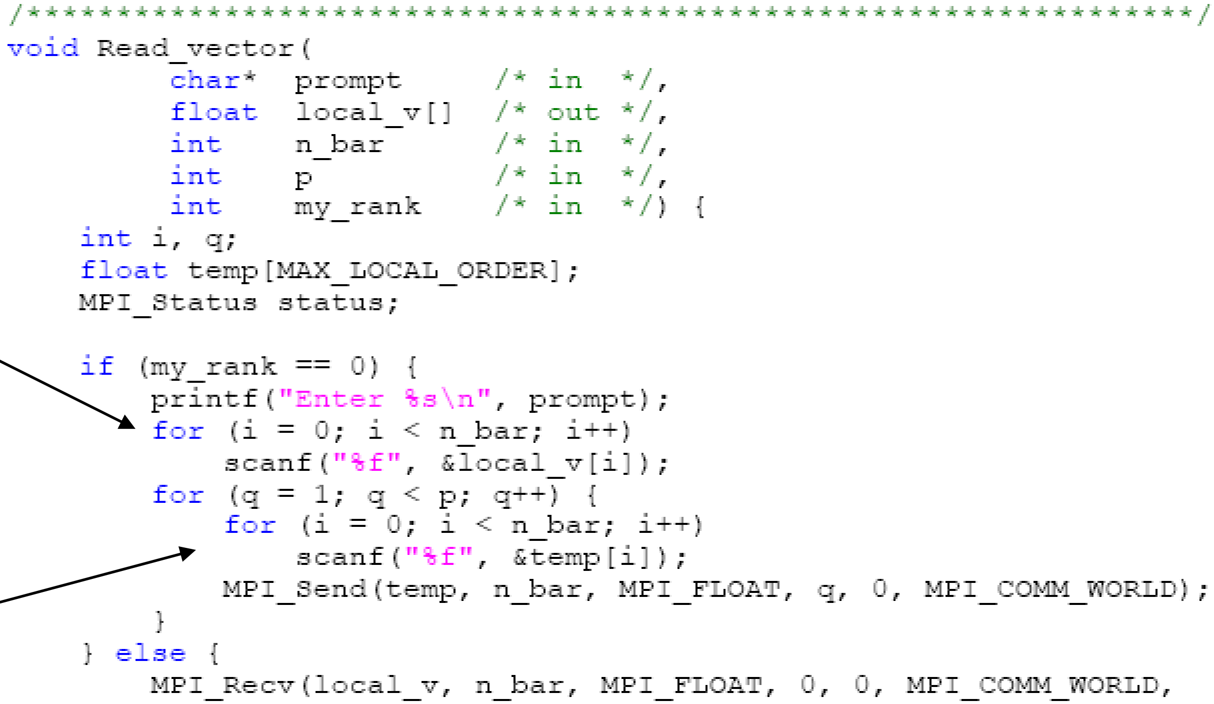


Figure 2.21 Broadcast as a tree construction.

OSS: A Reduce in practice does the reverse path, so both have $O(\log n)$ cost

Block mapping



```
/*****************************************************************/
void Read_vector(
         char*  prompt     /* in  */,
         float  local_v[]  /* out */,
         int    n_bar      /* in  */,
         int    p          /* in  */,
         int    my_rank    /* in  */) {
    int i, q;
    float temp[MAX_LOCAL_ORDER];
    MPI_Status status;

    if (my_rank == 0) {
        printf("Enter %s\n", prompt);
        for (i = 0; i < n_bar; i++)
            scanf("%f", &local_v[i]);
        for (q = 1; q < p; q++) {
            for (i = 0; i < n_bar; i++)
                scanf("%f", &temp[i]);
            MPI_Send(temp, n_bar, MPI_FLOAT, q, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
```

Read portion for process 0

Reads ň elements and sends them (Block mapping) one Process at a time

```
            &status);
    }
}  /* Read_vector */
```

Only the first `n_bar` of `local_v` are what each process needs!

NB Use Scatter (homework)

```
/*****************************************************
float Serial_dot(
          float  x[]   /* in */,
          float  y[]   /* in */,
          int    n     /* in */) {

    int    i;
    float  sum = 0.0;

    for (i = 0; i < n; i++)
        sum = sum + x[i]*y[i];
    return sum;
} /* Serial_dot */
```

Each process has allocated all the vector, but only the first **ň** elements are its

reduce…. Do the sum of all `local_dot` and place in `dot`

```
/*****************************************************
float Parallel_dot(
          float  local_x[]   /* in */,
          float  local_y[]   /* in */,
          int  n_bar         /* in */) {

    float  local_dot;
    float  dot = 0.0;
    float  Serial_dot(float x[], float y[], int m);

    local_dot = Serial_dot(local_x, local_y, n_bar);
    MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
        MPI_SUM, 0, MPI_COMM_WORLD);
    return dot;
} /* Parallel_dot */
```

Matrix-Vettor Product

Let $A=(a_{ij})$ be a m x n matrix
Let $x=(x_0, x_1, ..., x_{n-1})^T$

The **product** $\boldsymbol{y = Ax}$, is formed by all scalar products of each row of A with x

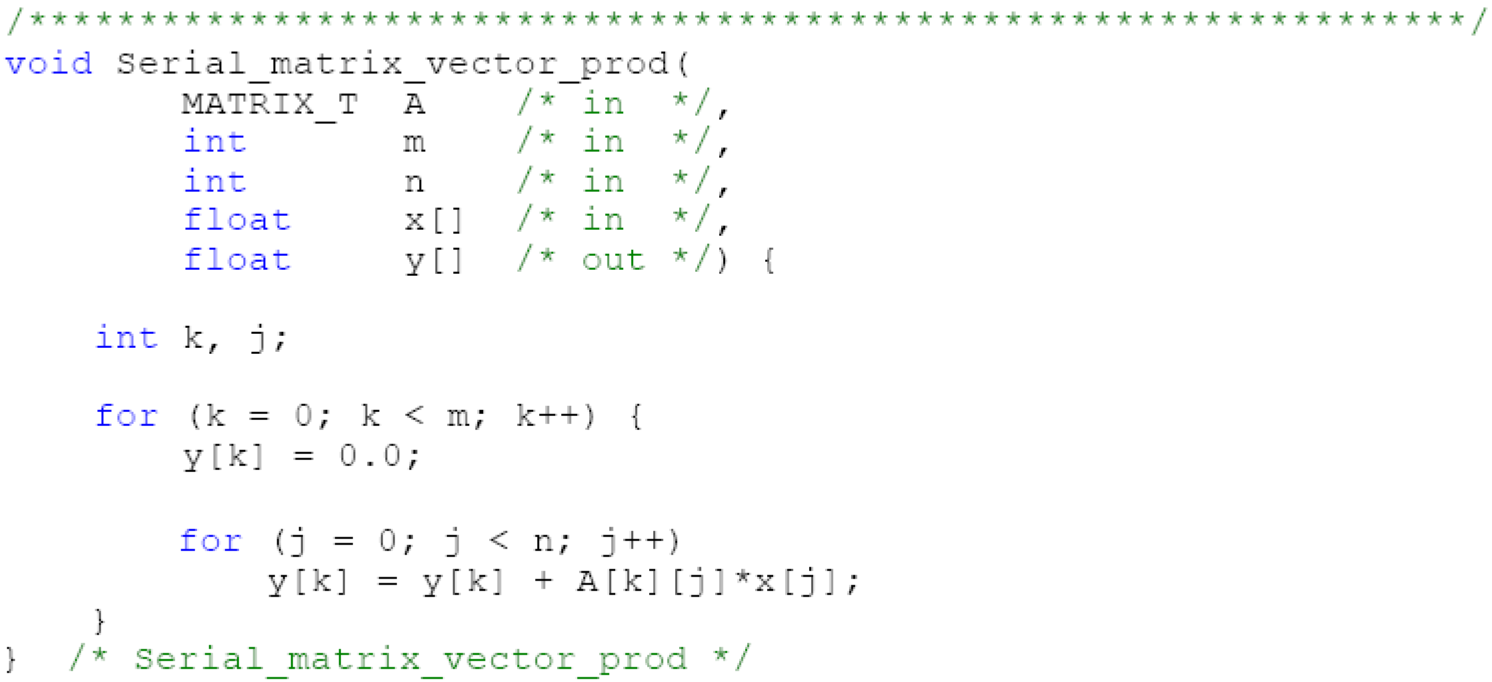
Thus, the vector y will be given by:

$$y=(y_0, y_1, ..., y_{m-1})^T$$

with:

$$y_k=a_{k0}x_0+a_{k1}x_1+...+a_{k,n-1}x_{n-1}$$

Serial



```
/*****************************************************************/
void Serial_matrix_vector_prod(
        MATRIX_T  A     /* in  */,
        int       m     /* in  */,
        int       n     /* in  */,
        float     x[]   /* in  */,
        float     y[]   /* out */) {

    int k, j;

    for (k = 0; k < m; k++) {
        y[k] = 0.0;

        for (j = 0; j < n; j++)
            y[k] = y[k] + A[k][j]*x[j];
    }
}  /* Serial_matrix_vector_prod */
```

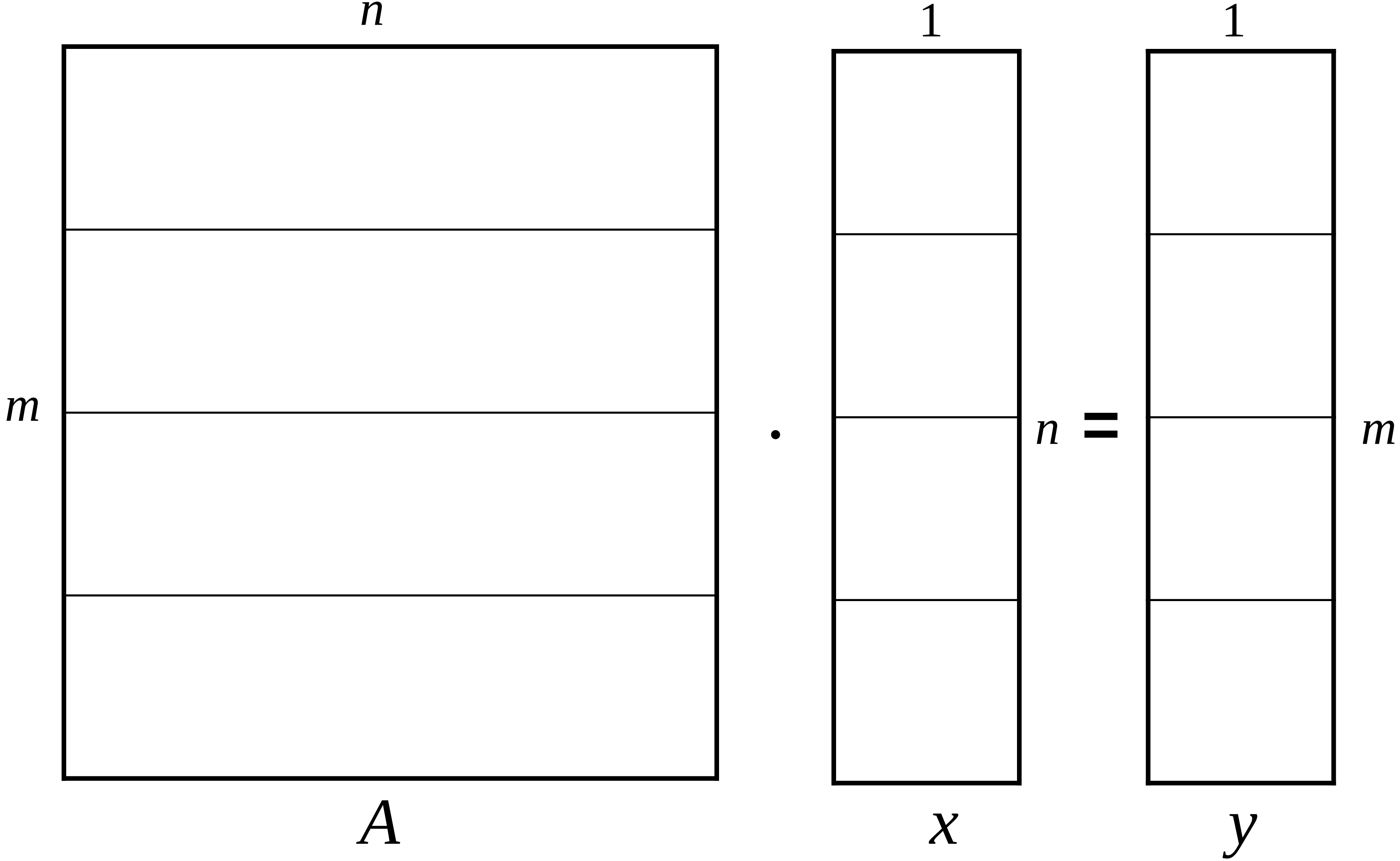
Data Distribution

- Block-row (panel) distribution

Process	Elements			
0	a_{00}	a_{01}	a_{02}	a_{03}
	a_{10}	a_{11}	a_{12}	a_{13}
1	a_{20}	a_{21}	a_{22}	a_{23}
	a_{30}	a_{31}	a_{32}	a_{33}
2	a_{40}	a_{41}	a_{42}	a_{43}
	a_{50}	a_{51}	a_{52}	a_{53}
3	a_{60}	a_{61}	a_{62}	a_{63}
	a_{70}	a_{71}	a_{72}	a_{73}

Mapping (4 processes)

$(m,n) \times (n,1) = (m,1)$



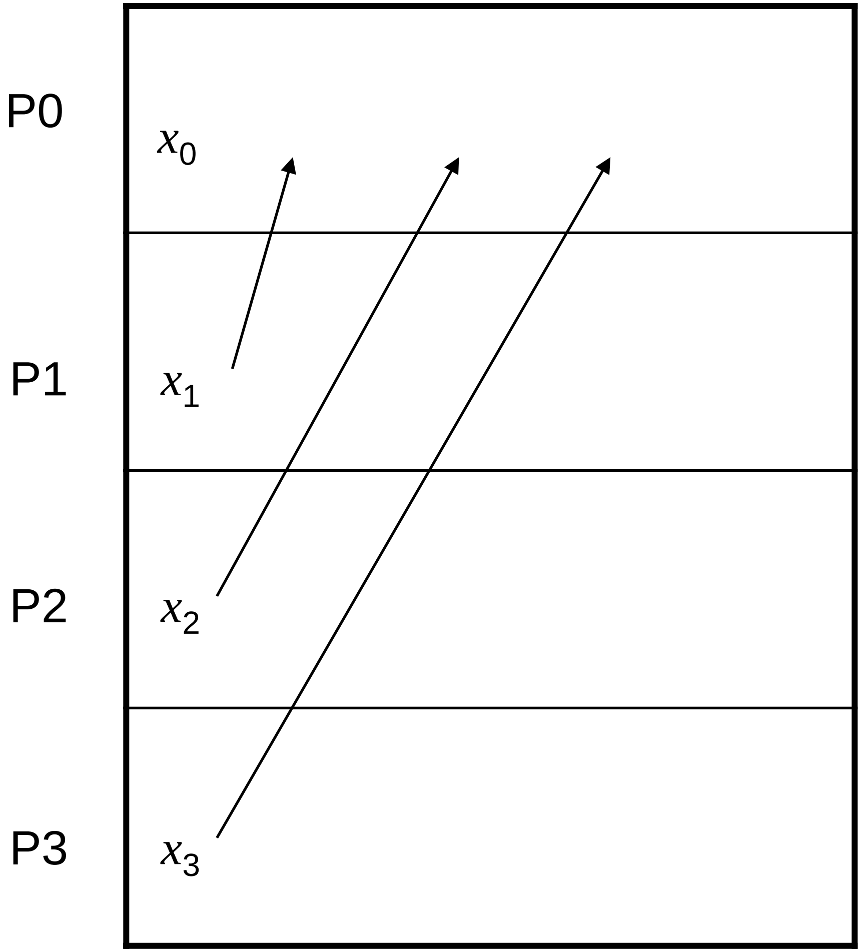
Gather or scatter?

- In order to form the scalar product of each row of A with x, we must make a **gather** of x on each process, or a **scatter** of each row of A on the processes
- For example, if m = n = p = 4, then a_{00}, a_{01}, a_{02} , a_{03} and x_0 are assigned to process 0, x_1 to process, x_2 to process 2, etc..

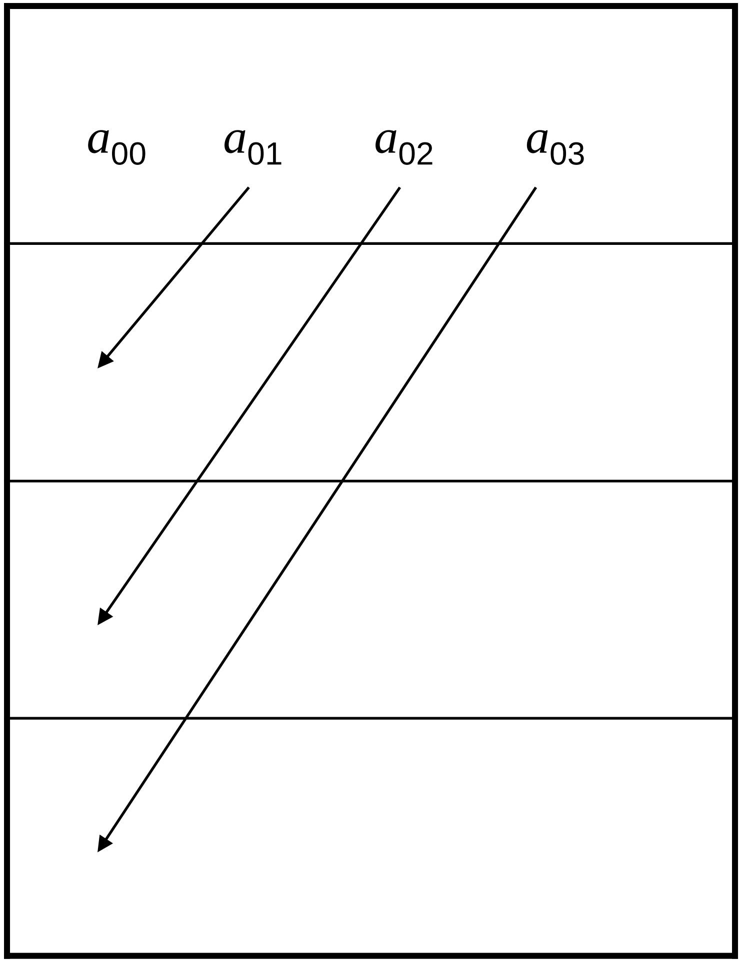
- In this way, to form the scalar product of the first row of A with x, we can
  - send x_1, x_2 and x_3 to the process 0, **<u>or</u>**
  - we can send a_{01} to the process 1, a_{02} and to process 2 and a_{01} to process 3.

- The first step is a **<u>gather</u>**, the second a **<u>scatter</u>**!
- We will use gather in the example below ... (scatter for the reading stage!)

$m=n=p=4$



Gather

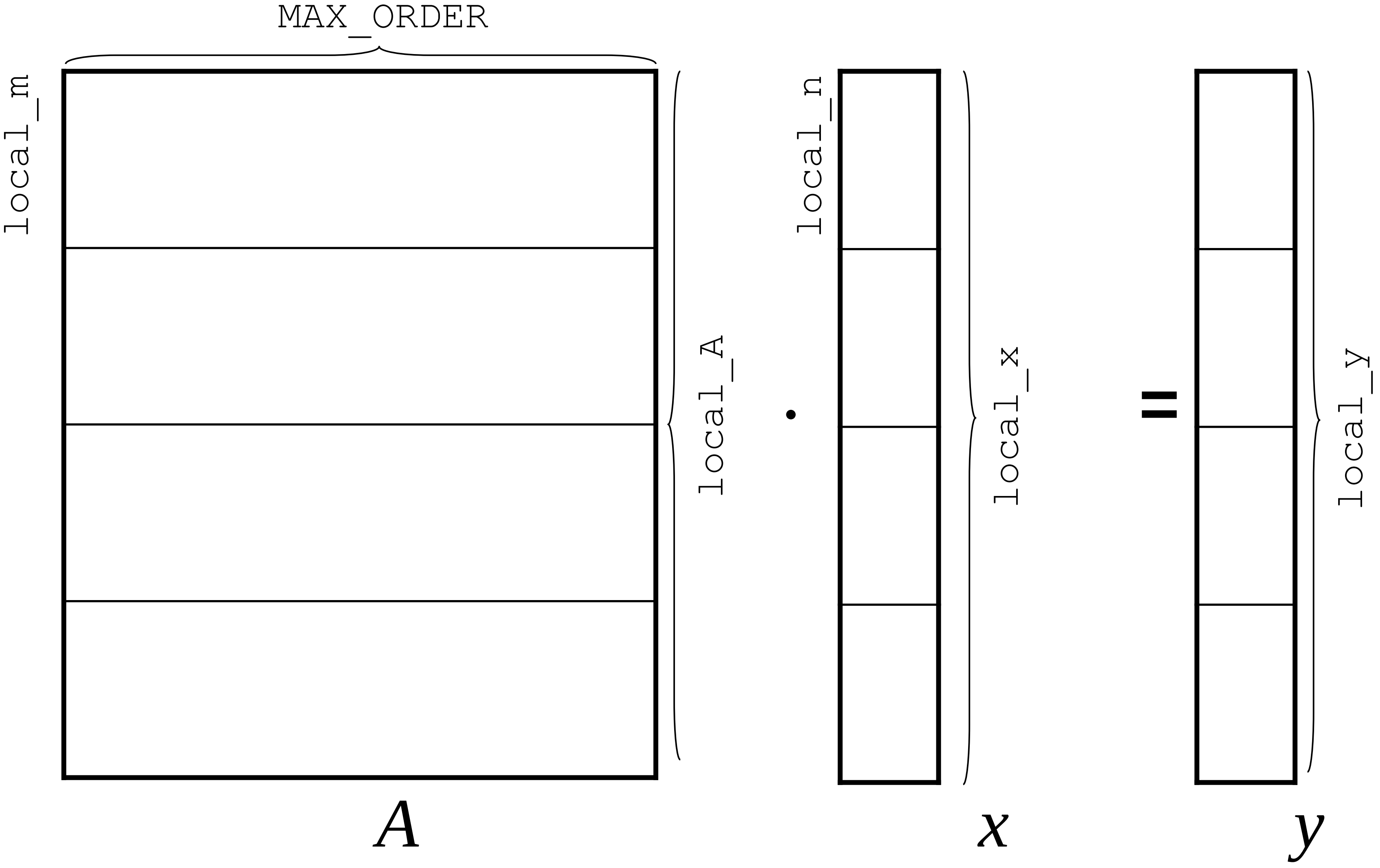


Scatter

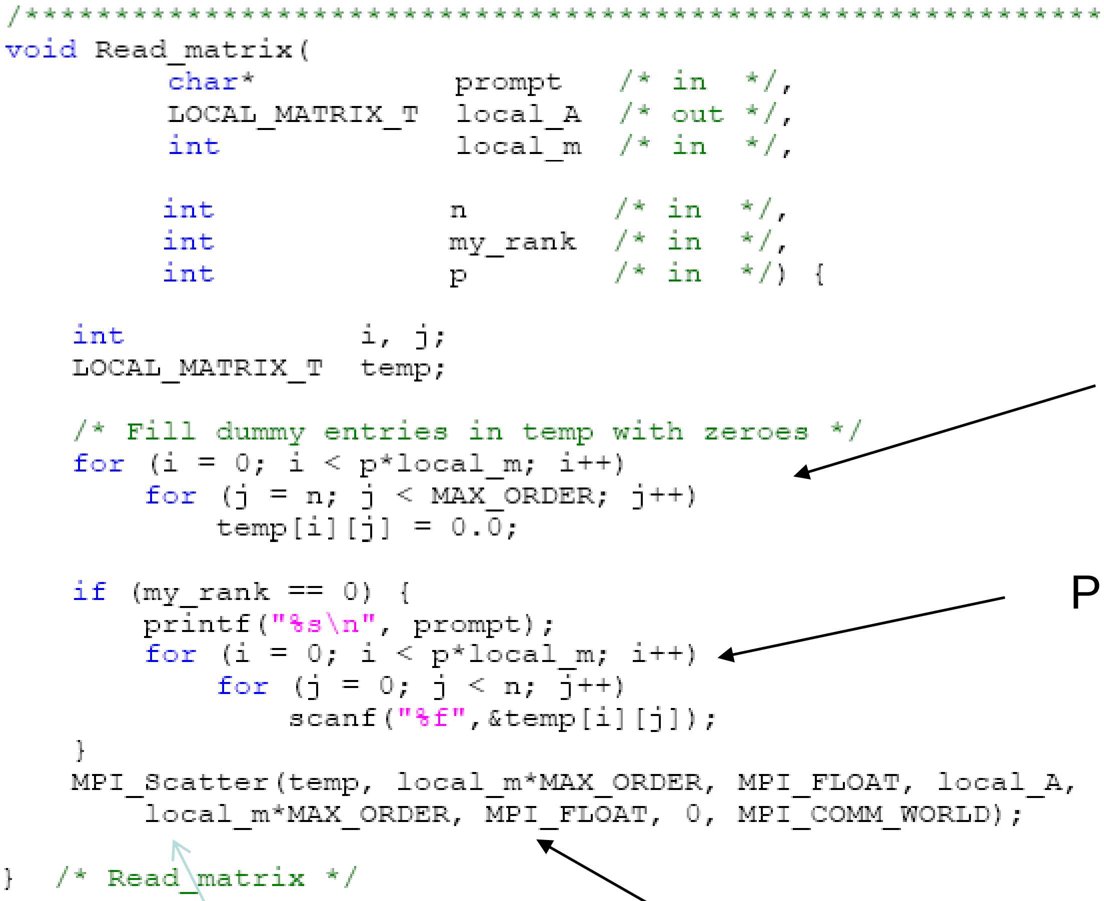
main

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

#define MAX_ORDER 100

typedef float LOCAL_MATRIX_T[MAX_ORDER][MAX_ORDER];

main(int argc, char* argv[]) {
    int              my_rank;
    int              p;
    LOCAL_MATRIX_T   local_A;
    float            global_x[MAX_ORDER];
    float            local_x[MAX_ORDER];
    float            local_y[MAX_ORDER];
    int              m, n;
    int              local_m, local_n;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0) {
        printf("Enter the order of the matrix (m x n)\n");
        scanf("%d %d", &m, &n);
    }
    MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    local_m = m/p;
    local_n = n/p;

    Read_matrix("Enter the matrix", local_A, local_m, n, my_rank, p);
    Print_matrix("We read", local_A, local_m, n, my_rank, p);

    Read_vector("Enter the vector", local_x, local_n, my_rank, p);
    Print_vector("We read", local_x, local_n, my_rank, p);

    Parallel_matrix_vector_prod(local_A, m, n, local_x, global_x,
        local_y, local_m, local_n);
    Print_vector("The product is", local_y, local_m, my_rank, p);

    MPI_Finalize();

}  /* main */
```

Mapping



Read and data allocation



```
/*********************************************************************/
void Read_matrix(
         char*              prompt   /* in  */,
         LOCAL_MATRIX_T     local_A  /* out */,
         int                local_m  /* in  */,

         int                n        /* in  */,
         int                my_rank  /* in  */,
         int                p        /* in  */) {

    int               i, j;
    LOCAL_MATRIX_T    temp;

    /* Fill dummy entries in temp with zeroes */
    for (i = 0; i < p*local_m; i++)
        for (j = n; j < MAX_ORDER; j++)
            temp[i][j] = 0.0;

    if (my_rank == 0) {
        printf("%s\n", prompt);
        for (i = 0; i < p*local_m; i++)
            for (j = 0; j < n; j++)
                scanf("%f",&temp[i][j]);
    }
    MPI_Scatter(temp, local_m*MAX_ORDER, MPI_FLOAT, local_A,
        local_m*MAX_ORDER, MPI_FLOAT, 0, MPI_COMM_WORLD);

}  /* Read_matrix */
```

Sets to zero the surplus elements of the matrix

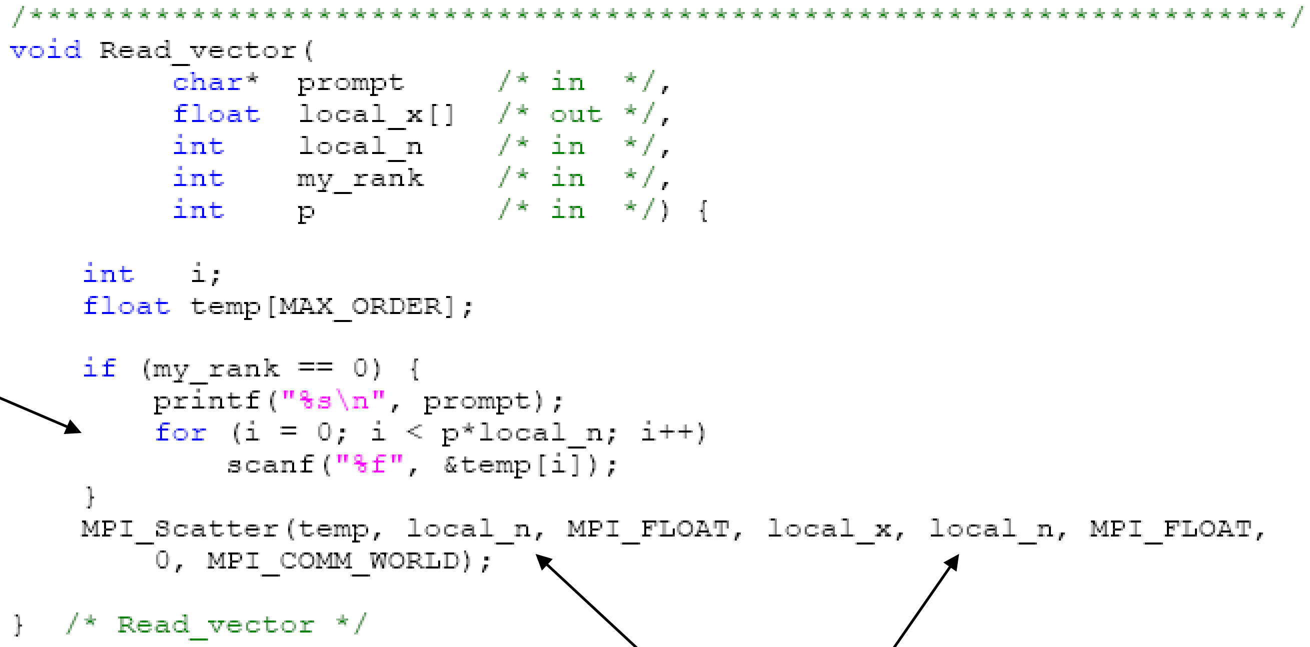
Process 0: reads all the matrix

CAREFUL! Scatter ok for **static** allocated matrix/vectors!

Process 0: scatter all matrix, but each process will receive aonly `local_A`
(in `C` le matrici sono row-wise)

Read and data allocation

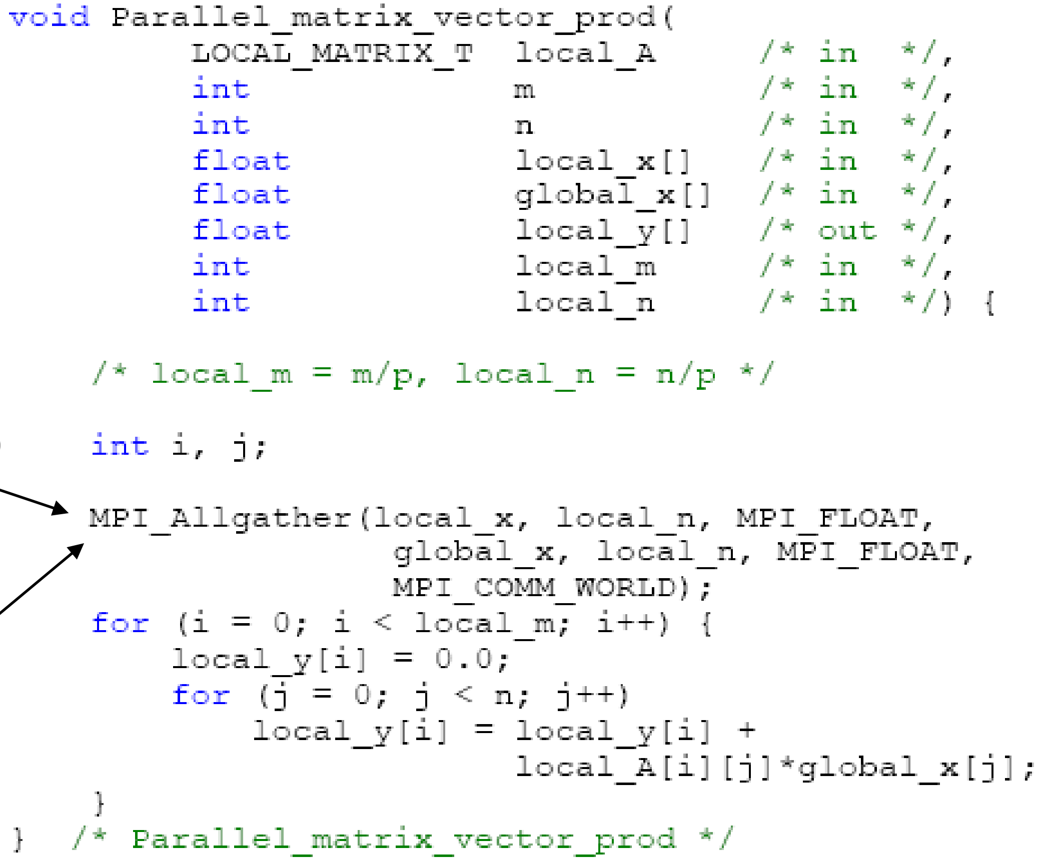
Reads all the vector and scatters it!



```
/********************************************************************/
void Read_vector(
         char*  prompt     /* in  */,
         float  local_x[]  /* out */,
         int    local_n    /* in  */,
         int    my_rank    /* in  */,
         int    p          /* in  */) {

    int   i;
    float temp[MAX_ORDER];

    if (my_rank == 0) {
        printf("%s\n", prompt);
        for (i = 0; i < p*local_n; i++)
            scanf("%f", &temp[i]);
    }
    MPI_Scatter(temp, local_n, MPI_FLOAT, local_x, local_n, MPI_FLOAT,
        0, MPI_COMM_WORLD);

}  /* Read_vector */
```

NB Like in all Scatters, it should be the same!

AllGather



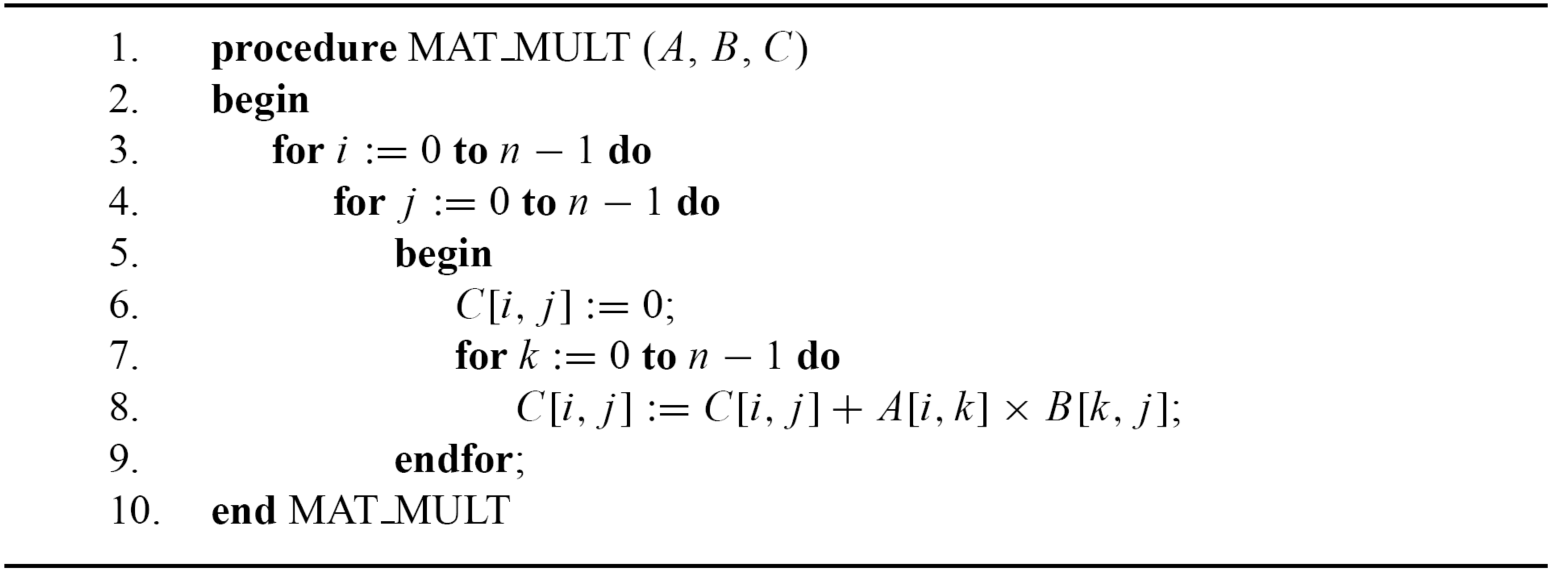
```
void Parallel_matrix_vector_prod(
         LOCAL_MATRIX_T  local_A      /* in  */,
         int             m            /* in  */,
         int             n            /* in  */,
         float           local_x[]    /* in  */,
         float           global_x[]   /* in  */,
         float           local_y[]    /* out */,
         int             local_m      /* in  */,
         int             local_n      /* in  */) {

    /* local_m = m/p, local_n = n/p */

    int i, j;

    MPI_Allgather(local_x, local_n, MPI_FLOAT,
                  global_x, local_n, MPI_FLOAT,
                  MPI_COMM_WORLD);
    for (i = 0; i < local_m; i++) {
        local_y[i] = 0.0;
        for (j = 0; j < n; j++)
            local_y[i] = local_y[i] +
                         local_A[i][j]*global_x[j];
    }
}  /* Parallel_matrix_vector_prod */
```

“We collect the pieces of x on each process (in `global_x`)

WE could of used also `MPI_Gather` with a `for` on all processes

Matrix – Matrix Product : Serial Algorithm

- O(n^3) Cost



```
procedure MAT_MULT ($A$, $B$, $C$)
begin
    for $i$ := 0 to $n - 1$ do
        for $j$ := 0 to $n - 1$ do
            begin
                $C[i, j] := 0$;
                for $k$ := 0 to $n - 1$ do
                    $C[i, j] := C[i, j] + A[i, k] \times B[k, j]$;
            endfor;
end MAT_MULT
```

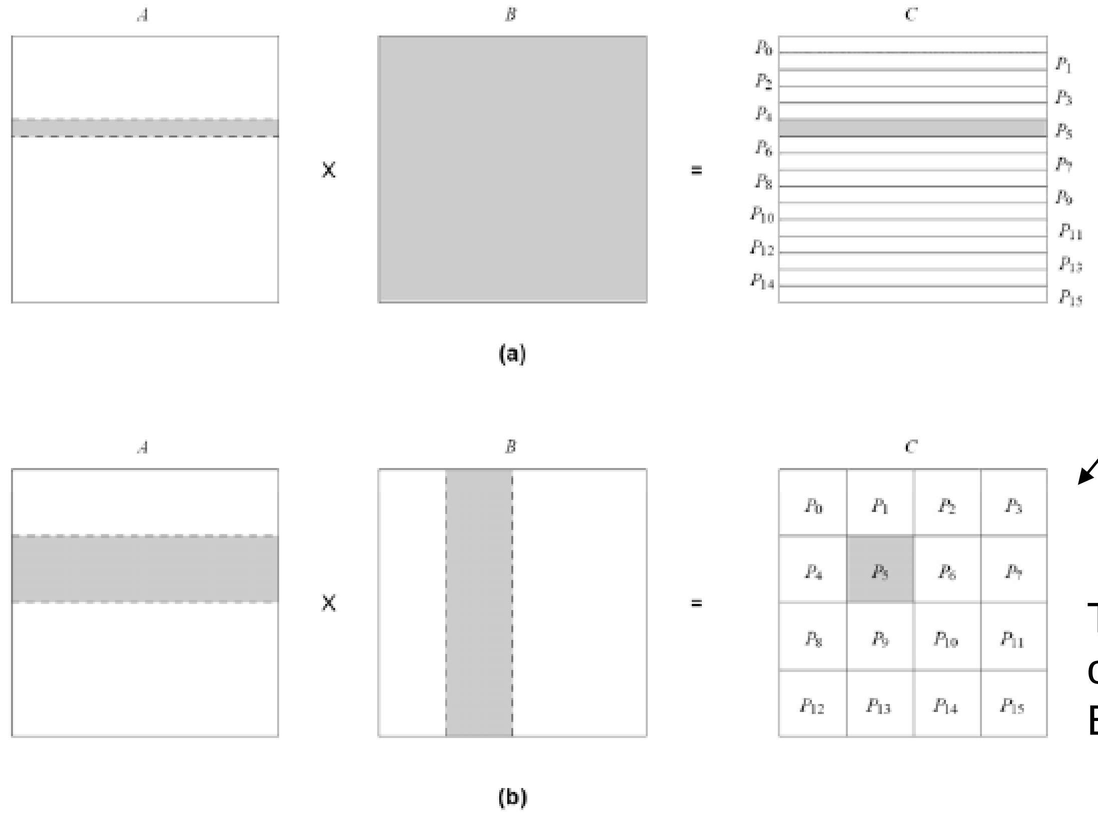
Algorithm 8.2 The conventional serial algorithm for multiplication of two $n \times n$ matrices.

Matrix – Matrix Product : Possible Allocations

C “rules” the allocation

Case a

Case b



(a)

(b)

The portions in gray
of A and B that are required
By the gray portion of C

Figure 3.26 Data sharing needed for matrix multiplication with (a) one-dimensional and (b) two-dimensional partitioning of the output matrix. Shaded portions of the input matrices A and B are required by the process that computes the shaded portion of the output matrix C.

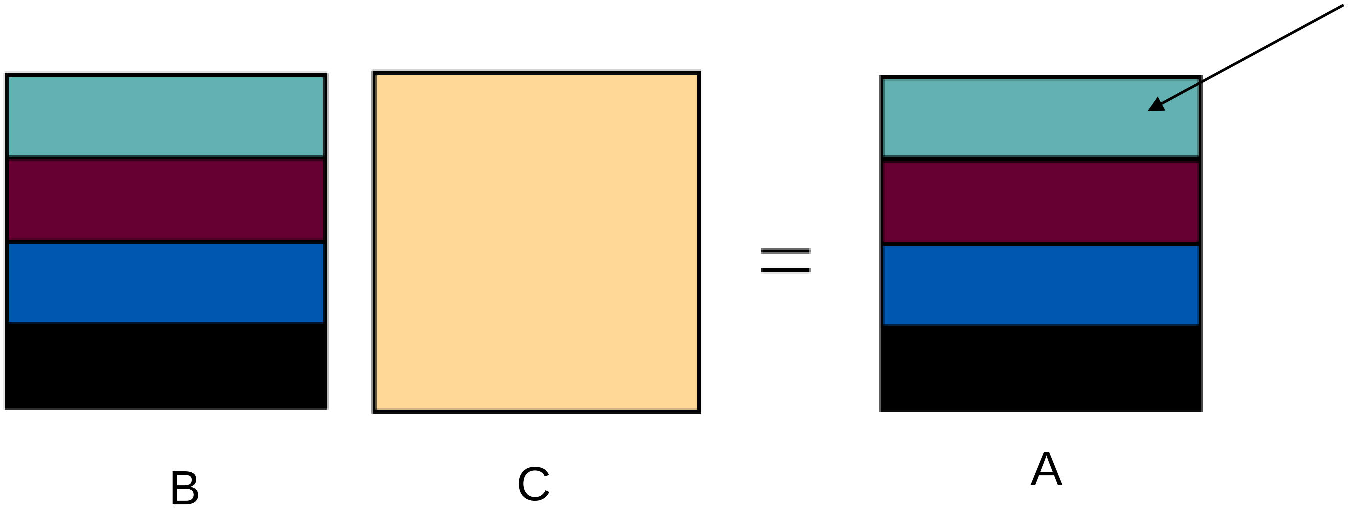
Matrix – Matrix Product

- For simplicity, we consider matrices of the same order (n, n)
- In case (a) we have a decomposition into one-dimensional blocks ; in (b) a bi-dimensional block decomposition
- Each process in the case (a) will have n / p rows, while in case (b), each process will have a block of dim

$$n/\sqrt{p} \times n/\sqrt{p}$$

- In (a) we can use up to n processes, in (b) up to n^2 (thus increasing the degree of parallelism)
- The "counter" of case (a) is that each process requires the corresponding n / p rows of A and of the whole B matrix, while in (b) each process requires $n/\sqrt{p}$ rows of A and $n/\sqrt{p}$ columns of B

Algorithm case (a)



To form this
row, we need
all C!
That's why we
broadcast all C

- **Distribute the rows of B to all** (Scatter)
- **Broadcast** all C (unfortunately !)
- Form the product of C with rows of B for each process.

These will be the corresponding rows of A
Returns the rows of A to a process using a **gather**
- This algorithm is different from that suggested by Pacheco, but similar to that of the LLNLt utorial (master / slave)

Algorithm case (a)

The algorithm applies only to these values! Generalize it!

A=BxC

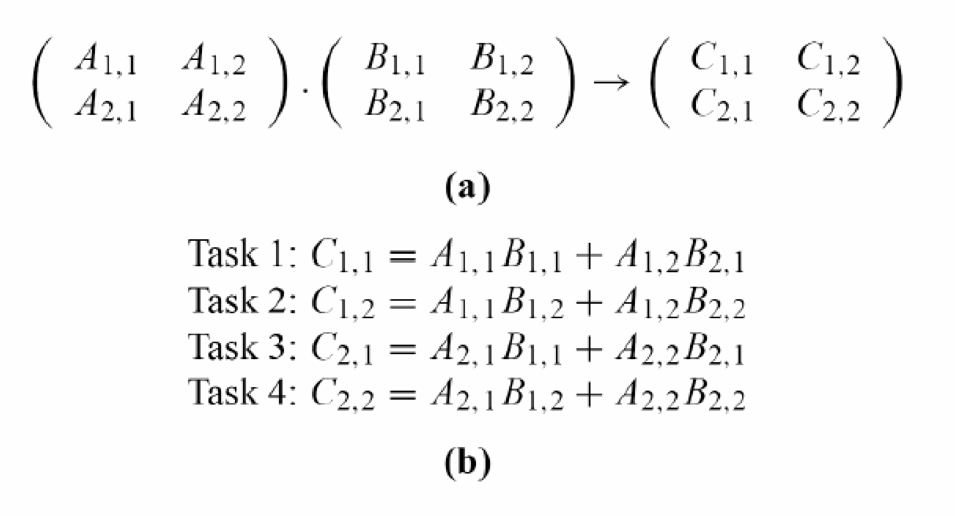
The algorithm applies to only 4 processors!
Generalize it**!**

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

#define NCOLS 4
#define NROWS 4

int main(int argc, char **argv) {

  int i,j,k,l;

  int ierr, rank, size, root;

  float A[NROWS][NCOLS];
  float Apart[NCOLS];
  float Bpart[NCOLS];
  float C[NROWS][NCOLS];

  float B[NCOLS][NCOLS];

  root = 0;
 ....
 /* Scatter matrix B by rows. */
  ierr=MPI_Scatter(B,NCOLS,MPI_FLOAT,Bpart,NCOLS,MPI_FLOAT,root,MPI_COMM_WORLD);

  /* Broadcast C */

  ierr=MPI_Bcast(C,NROWS*NCOLS,MPI_FLOAT,root,MPI_COMM_WORLD);

  /* Do the vector-scalar multiplication. */

  for(j=0;j<NCOLS;j++){
    Apart[j] = 0.0;
    for(k=0; k<NROWS; k++)
      Apart[j] += Bpart[k]*C[k][j];
  }
 ....
  /* Gather matrix A. */

  ierr=MPI_Gather(Apart,NCOLS,MPI_FLOAT,A,NCOLS,MPI_FLOAT, root,MPI_COMM_WORLD);

  /* Report results */

  if (rank == 0) {

    printf("\nThis is the result of the parallel computation:\n\n");

    for(j=0;j<NROWS;j++) {
      for(k=0;k<NCOLS;k++) {
        printf("A[%d][%d]=%g\n",j,k,A[j][k]);
      }
    }
```

Remember what a gather does!
A is constructed from many Apart pieces!
That is, we collect the various rows!

Algorithm case (b)

- Suppose, for example, the partitioning of data as in Fig.
- The 4 submatrixes $C_{i,j}$ (of dimension $n/2$ x $n/2$), can be computed independently



$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

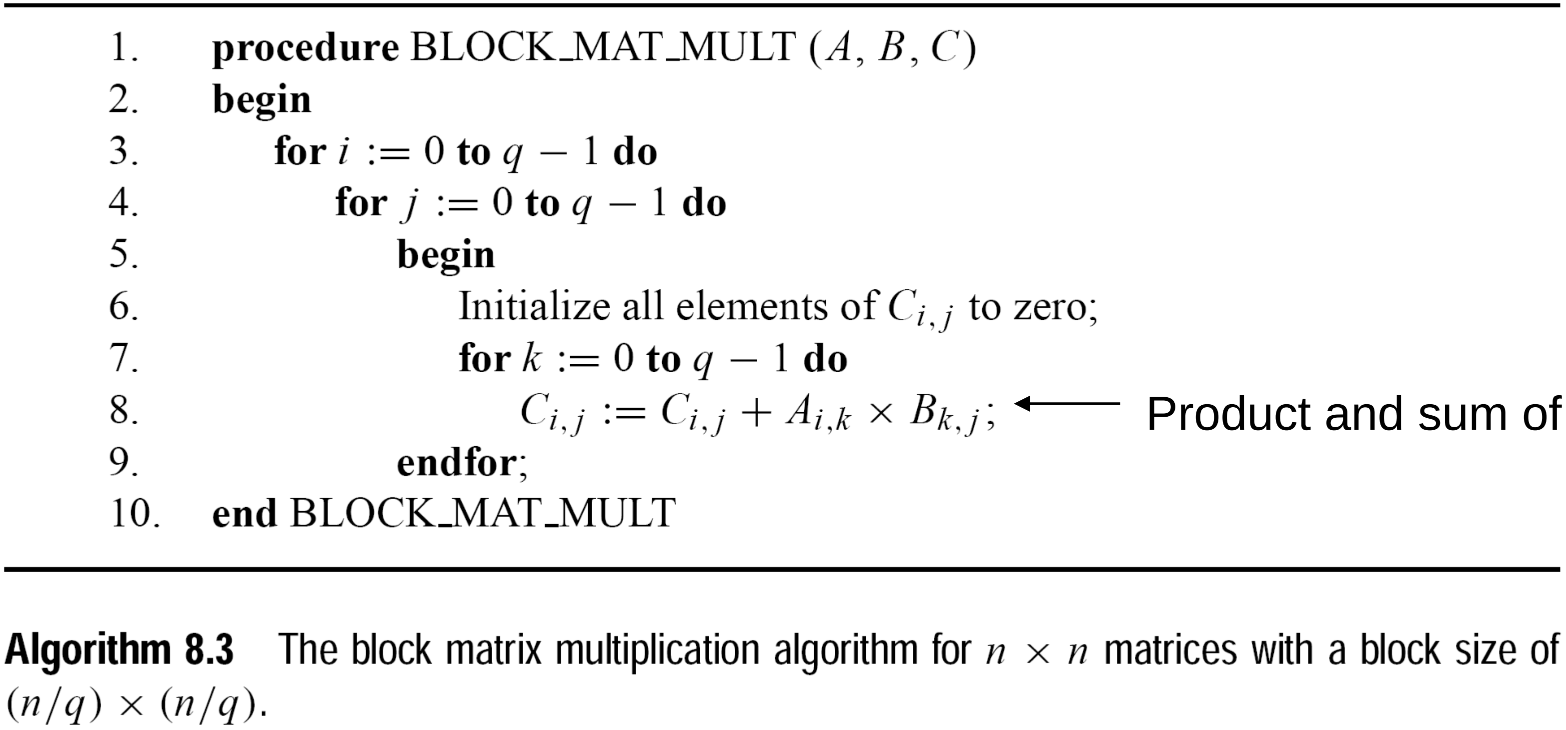
(a)

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

(b)

OBS: other partitionings are also possible!

Block Algorithm (Serial) – Case (b)



```
procedure BLOCK_MAT_MULT ($A$, $B$, $C$)
begin
    for $i$ := 0 to $q - 1$ do
        for $j$ := 0 to $q - 1$ do
            begin
                Initialize all elements of $C_{i,j}$ to zero;
                for $k$ := 0 to $q - 1$ do
                    $C_{i,j} := C_{i,j} + A_{i,k} \times B_{k,j}$;
            endfor;
end BLOCK_MAT_MULT
```

Product and sum of matrixes

Algorithm 8.3 The block matrix multiplication algorithm for $n \times n$ matrices with a block size of $(n/q) \times (n/q)$.

Parallel Algorithm (case b)

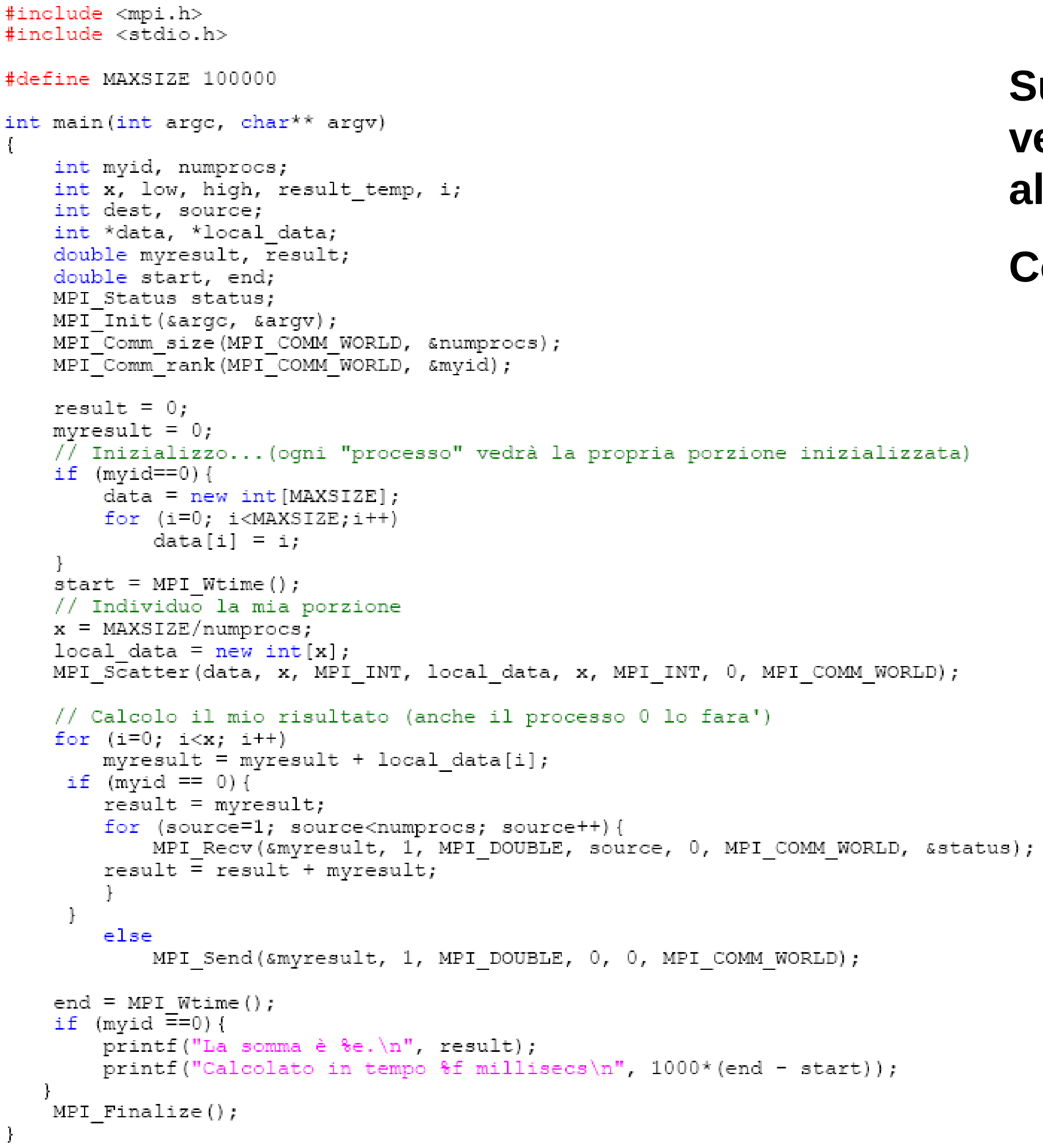
- Consider two matrixes ($n \times n$) A e B partitioned in p blocks $A_{i,j}$ and $B_{i,j}$ ($0 \le i, j < \sqrt{p}$) di dimension n
- Initially process $P_{i,j}$ stores $A_{i,j}$ and $B_{i,j}$ and computes the block $C_{i,j}$ of the resulting matrix $(n/\sqrt{p}) \times (n/\sqrt{p})$
- The computation of the submatrix $C_{i,j}$ requires all submatrixes $A_{i,k}$ and $B_{k,j}$ for $0 \le k < \sqrt{p}$
- Execute All-to-all broadcast (that is `MPI_Allgather`) of A blocks along the rows and of B along columns
- Execute the multiplication of local submatrixes
- **Obs**: The cost of this algorithm is identical to the serial version (n^3): q^3 matrix products are carried out, each of $(n/q) \times (n/q)$ matrixes and $(n/q)^3$ additions and multiplications

Homework ☺ - again?

- Pi computation with Montecarlo
- Vector Maximum
- Search of element in a vector
- Sum of two matrixes

Sum of elements of a vector

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

#define MAXSIZE 10

int main(int argc, char** argv)
{
    int myid, numprocs;
    int data[MAXSIZE], i, x, low, high, myresult, result;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    result = 0;
    myresult = 0;

    if (myid == 0)
        {
        // Inizializzo...
        for (i=0; i<MAXSIZE;i++)
            data[i] = i;
        }

    // Invio il vettore
    MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD);

    // NB Tutti i processi (compresi 0) calcolano...
    x = MAXSIZE/numprocs;
    low = myid * x;
    high = low + x;
    for (i=low; i<high; i++)
        myresult = myresult + data[i];

    printf("Il processo %d ha calcolato %d\n", myid, myresult);

    MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid ==0)
        printf("La somma è %d.\n", result);

    MPI_Finalize();
    exit(0);
}
```



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

#define MAXSIZE 100000

int main(int argc, char** argv)
{
    int myid, numprocs;
    int x, low, high, result_temp, i;
    int dest, source;
    int *data, *local_data;
    double myresult, result;
    double start, end;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    result = 0;
    myresult = 0;
    // Inizializzo...(ogni "processo" vedrà la propria porzione inizializzata)
    if (myid==0){
        data = new int[MAXSIZE];
        for (i=0; i<MAXSIZE;i++)
            data[i] = i;
    }
    start = MPI_Wtime();
    // Individuo la mia porzione
    x = MAXSIZE/numprocs;
    local_data = new int[x];
    MPI_Scatter(data, x, MPI_INT, local_data, x, MPI_INT, 0, MPI_COMM_WORLD);

    // Calcolo il mio risultato (anche il processo 0 lo fara')
    for (i=0; i<x; i++)
        myresult = myresult + local_data[i];
     if (myid == 0){
        result = myresult;
        for (source=1; source<numprocs; source++){
            MPI_Recv(&myresult, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, &status);
        result = result + myresult;
        }
     }
        else
            MPI_Send(&myresult, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

    end = MPI_Wtime();
    if (myid ==0){
        printf("La somma è %e.\n", result);
        printf("Calcolato in tempo %f millisecs\n", 1000*(end - start));
   }
    MPI_Finalize();
}
```

Sum of elements o a vector (dynamic allocation)

Compile with mpiCC !

Matrix Scatter

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];

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

if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
             MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
         recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

NB Scatter called by all processes!

Output

```
rank= 0  Results: 1.000000 2.000000 3.000000 4.000000
rank= 1  Results: 5.000000 6.000000 7.000000 8.000000
rank= 2  Results: 9.000000 10.000000 11.000000 12.000000
rank= 3  Results: 13.000000 14.000000 15.000000 16.000000
```

`Get_data2` **(with Broadcast)**

```c
/********************************************************************/
/* Function Get_data2
 * Reads in the user input a, b, and n.
 * Input parameters:
 *     1.  int my_rank:  rank of current process.
 *     2.  int p:  number of processes.
 * Output parameters:
 *     1.  float* a_ptr:  pointer to left endpoint a.
 *     2.  float* b_ptr:  pointer to right endpoint b.
 *     3.  int* n_ptr:  pointer to number of trapezoids.
 * Algorithm:
 *     1.  Process 0 prompts user for input and
 *         reads in the values.
 *     2.  Process 0 sends input values to other
 *         processes using three calls to MPI_Bcast.
 */
void Get_data2(
         float*  a_ptr    /* out */,
         float*  b_ptr    /* out */,
         int*    n_ptr    /* out */,
         int     my_rank  /* in  */) {

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }
    MPI_Bcast(a_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
} /* Get_data2 */
```

```
#include <stdio.h>

/* We'll be using MPI routines, definitions, etc. */
#include "mpi.h"

main(int argc, char** argv) {

    ....


    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    MPI_Comm_size(MPI_COMM_WORLD, &p);

    Get_data2(&a, &b, &n, my_rank);

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
        MPI_SUM, 0, MPI_COMM_WORLD);

    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",
            n);
        printf("of the integral from %f to %f = %f\n",
            a, b, total);
    }

    /* Shut down MPI */
    MPI_Finalize();
} /*  main  */
```

Get_data2!

Numerical integration (final)

```
    ...
    MPI_Barrier(MPI_COMM_WORLD);
    start = MPI_Wtime();

    h = (b-a)/n;     /* h is the same for all processes */
    local_n = n/p;   /* So is the number of trapezoidals */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;

    /* Call the serial trapezoidal function */
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
        MPI_SUM, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
    finish = MPI_Wtime();
     /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",
            n);
        printf("of the integral from %f to %f = %f\n",
            a, b, total);
        printf("Elapsed time in seconds","%e",
         (finish - start) - overhead);

    }

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

?

Wow!

```c
 */
#include <stdio.h>
#include "mpi.h"
#include "cio.h"

main(int argc, char* argv[]) {
    ...
    MPI_Comm  io_comm;
    ...

    float Trap(float local_a, float local_b, int local_n,
              float h);    /* Calculate local integral  */

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_dup(MPI_COMM_WORLD, &io_comm);
    Cache_io_rank(MPI_COMM_WORLD, io_comm);

    Cscanf(io_comm,"Enter a, b, and n","%f %f %d", &a, &b, &n);

    /* Estimate overhead */
    overhead = 0.0;
    for (i = 0; i < 100; i++) {
        MPI_Barrier(MPI_COMM_WORLD);
        start = MPI_Wtime();
        MPI_Barrier(MPI_COMM_WORLD);
        finish = MPI_Wtime();
        overhead = overhead + (finish - start);
    }
    overhead = overhead/100.0;
```

Barrier estimation time (average on 100 times)

```c
    MPI_Barrier(MPI_COMM_WORLD);
    start = MPI_Wtime();

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoidals */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;

    /* Call the serial trapezoidal function */
    integral = Trap(local_a, local_b, local_n, h);

    /* Add up the integrals calculated by each process */
    MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
        MPI_SUM, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
    finish = MPI_Wtime();
    Cprintf(io_comm,"Our estimate is","%f",total);
    Cprintf(io_comm,"Elapsed time in seconds","%e",
        (finish - start) - overhead);

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

Calcolo di π:

un esempio

- Calcoliamo π tramite integrazione numerica
- Usando le seguenti routine MPI:

MPI_BARRIER, MPI_BCAST, MPI_REDUCE

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

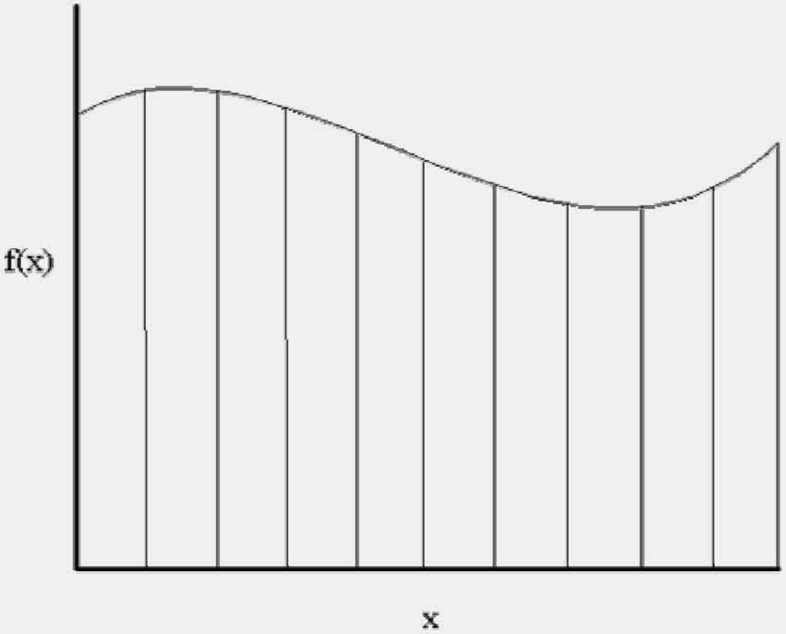
Pseudocodice seriale

```
f(x) = 1/(1+x^2)
h = 1/N, sum = 0.0
do i = 1, N
        x = h*(i - 0.5)
        sum = sum + f(x)
enddo
pi = h * sum
```

Esempio:

N = 10, h=0.1

x={.05, .15, .25, .35, .45,
.55, 65, .75, .85, .95}



Pseudocodice parallelo

P(0) legge N e lo spedisce con broadcast a tutti i processori

```
$f(x) = 1/(1+x^2)$
h = 1/N, sum = 0.0
do i = my_rank+1, N, nproc
    x = h*(i - 0.5)
    sum = sum + f(x)
enddo
mypi = h * sum
```

P(0) colleziona la variabile mypi da ogni processore e la riduce al valore pi

Esempio:

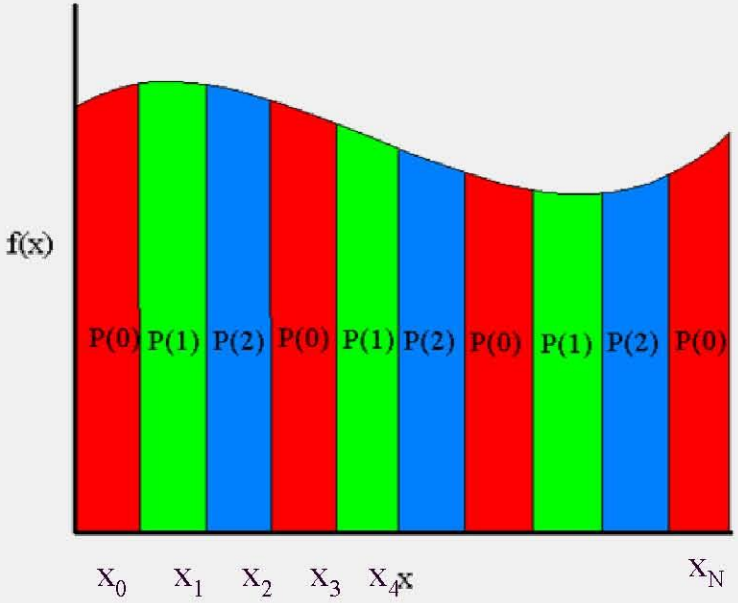
N = 10, h=0.1

Procrs: {P(0),P(1),P(2)}

P(0) -> {.05, .35, .65, .95}

P(1) -> {.15, .45, .75}

P(2) -> {.25, .55, .85}

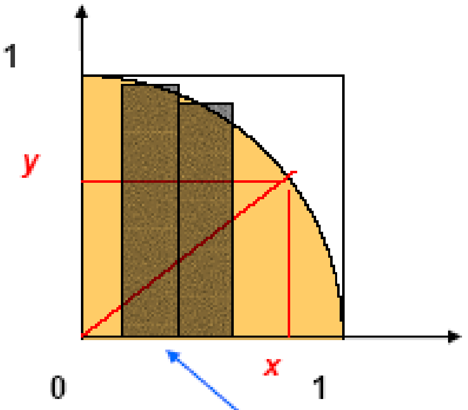


Calcolo di π:

il programma

```
int n; /* Numero di rettangoli */
int nproc, myrank;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_Size(MPI_COMM_WORLD,&nproc);
if (my_rank == 0)  read_from_keyboard(&n);

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
h   = 1.0 / (double) n;
       sum = 0.0;
       for (i = my_rank + 1; i <= n; i += nproc) {
         x = h * ((double)i - 0.5);
         sum += 4.0 / (1.0 + x*x);
       }
       mypi = h * sum;

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
     MPI_COMM_WORLD);
```

Calcolo PI greco



E' noto che l'area del cerchio è $r^2 \pi$, per cui l'area del semicerchio con r=1 è:

$\pi/4$

Curva cerchio (teorema di Pitagora):

$x^2 + y^2 = 1$

$y = \sqrt{(1-x^2)}$

L'area del semicerchio corrisponde al calcolo del seguente integrale:

$$\int_0^1 \sqrt{1-X^2}$$

Possiamo calcolarlo numericamente. Maggiore è il numero di intervalli in cui suddividiamo [0..1], maggiore è la precisione del calcolo dell'integrale

Equivalentemente possiamo calcolare

$$\int_0^1 \frac{1}{1+x^2}$$

ARCTAN(1) = $\pi/4$

ARCTAN(0) = 0

Esempio: PI greco in C (1)

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

int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  if (myid == 0) {
      printf("Enter the number of intervals: (0 quits) ");
      scanf("%d",&n);
  }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Esempio: PI greco in C (2)

```
  if (n != 0) {
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double) i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is %.16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
  return 0;
}
```

Altre routine...

- Molte routine di broadcast hanno un corrispettivo che permette di maneggiare vettori anzichè scalari
  - MPI_Gatherv(), MPI_Scatterv(), MPI_Allgatherv(), MPI_Alltoallv()
- MPI_Reduce_scatter(): funzionalità equivalente a reduce seguita da scatter
- I dettagli riguardanti queste ed altre routine derivate si possono ottenere dal manuale dell'MPI

Derived Datatypes

Send a sub-vector from process 0 to 1

```
/*  spedizione di un sotto-vettore dal processo 0 al processo 1
 *
 *
 *
 * Note: Dovrebbe eseguito su due processi.
 *
 */
#include <stdio.h>
#include "mpi.h"

main(int argc, char* argv[]) {
    float vector[100];
    MPI_Status status;
    int p;
    int my_rank;
    int i;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /*  Inizializzazione vettore e spedizione */
    if (my_rank == 0) {
        for (i = 0; i < 50; i++)
            vector[i] = 0.0;
        for (i = 50; i < 100; i++)
            vector[i] = 1.0;
        MPI_Send(vector+50, 50, MPI_FLOAT, 1, 0,
            MPI_COMM_WORLD);
    } else { /* my_rank == 1 */
        MPI_Recv(vector+50, 50, MPI_FLOAT, 0, 0,
            MPI_COMM_WORLD, &status);
        for (i = 50; i < 100; i++)
            printf("%3.1f ",vector[i]);
        printf("\n");
    }

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

Possible since elements of A vector in C are contiguous!

```
/* spedizone della terza colonna di una matrice dal processo 0 al processo 1
 *     process 1
 *
 * Note: Dovrebbe eseguito su due processi.
 *
 */
#include <stdio.h>
#include "mpi.h"

main(int argc, char* argv[]) {
    int p;
    int my_rank;
    float A[10][10];
    MPI_Status status;
    MPI_Datatype column_mpi_t;
    int i, j;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    MPI_Type_vector(10, 1, 10, MPI_FLOAT, &column_mpi_t);
    MPI_Type_commit(&column_mpi_t);

    if (my_rank == 0) {
        for (i = 0; i < 10; i++)
            for (j = 0; j < 10; j++)
                A[i][j] = (float) j;
        MPI_Send(&(A[0][2]), 1, column_mpi_t, 1, 0,
            MPI_COMM_WORLD);
    } else { /* my_rank = 1 */
        MPI_Recv(&(A[0][2]), 1, column_mpi_t, 0, 0,
            MPI_COMM_WORLD, &status);
        for (i = 0; i < 10; i++)
            printf("%3.1f ", A[i][2]);
        printf("\n");
    }

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

Send 3rd column from
process 0 to 1

Non contiguous elements
In C matrixes!

Initialization

Process 1 receives and
places the column in its own
A matrix

Send a column 1 to row 1
on another process

```
/* Interessante: spediamo la colonna 1 di una matrice
 * alla riga 1 di una matrice (su un altro processo)
 *
 *
 *  Note: Dovrebbe eseguito su due processi.
  */
#include <stdio.h>
#include "mpi.h"

main(int argc, char* argv[]) {
    int p;
    int my_rank;
    float A[10][10];
    MPI_Status status;
    MPI_Datatype column_mpi_t;
    int i, j;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    MPI_Type_vector(10, 1, 10, MPI_FLOAT, &column_mpi_t);
    MPI_Type_commit(&column_mpi_t);

    if (my_rank == 0) {
        for (i = 0; i < 10; i++)
            for (j = 0; j < 10; j++)
                A[i][j] = (float) i;
        MPI_Send(&(A[0][0]), 1, column_mpi_t, 1, 0,
                 MPI_COMM_WORLD);
    } else { /* my_rank = 1 */
        for (i = 0; i < 10; i++)
            for (j = 0; j < 10; j++)
                A[i][j] = 0.0;
        MPI_Recv(&(A[0][0]), 10, MPI_FLOAT, 0, 0,
                 MPI_COMM_WORLD, &status);
        for (j = 0; j < 10; j++)
            printf("%3.1f ", A[0][j]);
        printf("\n");
    }

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

Send column

Initialization

Receive a “10 MPI_FLOAT data element”

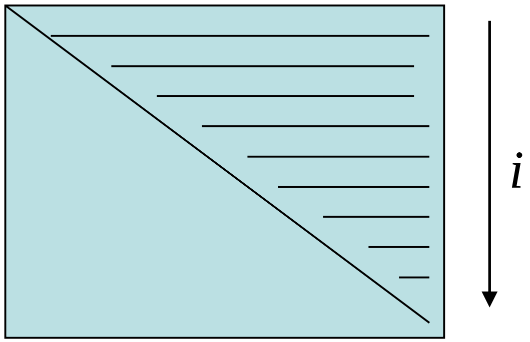
Send 3rd row from process 0 to 1 (no use of derived datatypes)

```
/* Spedisco la terza riga di una matrice dal processo 0
 * al processo 1
 * NB Non c'e' bisogno di utilizzare i derived datatypes!

 *  Note: Dovrebbe eseguito su due processi.
 */
#include <stdio.h>
#include "mpi.h"

main(int argc, char* argv[]) {
    int p;
    int my_rank;
    float A[10][10];
    MPI_Status status;
    int i, j;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0) {
        for (i = 0; i < 10; i++)
            for (j = 0; j < 10; j++)
                A[i][j] = (float) i;
        MPI_Send(&(A[2][0]), 10, MPI_FLOAT, 1, 0,
            MPI_COMM_WORLD);
    } else { /* my_rank = 1 */
        MPI_Recv(&(A[2][0]), 10, MPI_FLOAT, 0, 0,
            MPI_COMM_WORLD, &status);
        for (j = 0; j < 10; j++)
            printf("%3.1f ", A[2][j]);
        printf("\n");
    }

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

Placed in third row, but can go anywhere!!!

Send the upper triangle of a matrix from 0 to 1

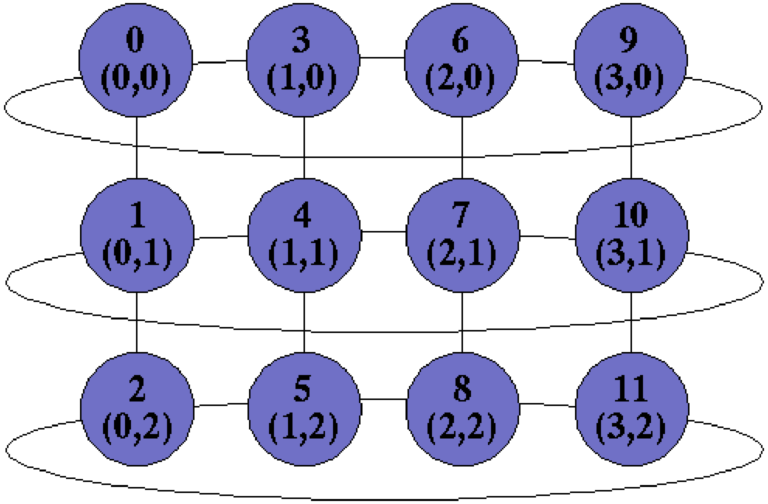
The trick is here! Locate the various Rows of the triangle!



```
/* Interessante: Spedisco la porzione triangolo superiore
 * di una matrice da 0 a 1
 *
 *  Note: Dovrebbe eseguito su due processi.
*/
#include <stdio.h>
#include "mpi.h"

#define n 10

main(int argc, char* argv[]) {
    int p;
    int my_rank;
    float         A[n][n];          /* Complete Matrix */
    float         T[n][n];          /* Upper Triangle  */
    int           displacements[n];
    int           block_lengths[n];
    MPI_Datatype  index_mpi_t;
    int           i, j;
    MPI_Status    status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    for (i = 0; i < n; i++) {
        block_lengths[i] = n-i;
        displacements[i] = (n+1)*i;
    }
    MPI_Type_indexed(n, block_lengths, displacements,
        MPI_FLOAT, &index_mpi_t);
    MPI_Type_commit(&index_mpi_t);

    if (my_rank == 0) {
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                A[i][j] = (float) i + j;
        MPI_Send(A, 1, index_mpi_t, 1, 0, MPI_COMM_WORLD);
    } else {/* my_rank == 1 */
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                T[i][j] = 0.0;
        MPI_Recv(T, 1, index_mpi_t, 0, 0, MPI_COMM_WORLD, &status);
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++)
                printf("%4.1f ", T[i][j]);
            printf("\n");
        }
    }

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

Virtual Topologies

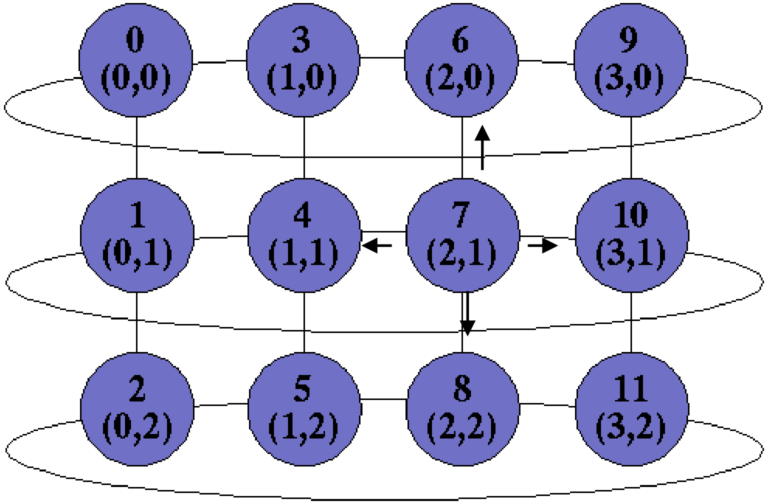
Esempio



```
#include<mpi.h>
#define TRUE 1
#define FALSE 0
void main(int argc, char *argv[]){
         int rank; MPI_Comm vu; int dim[2],period[2],reorder;
         int up,down,right,left;
         MPI_Init(&argc, &argv);
         MPI_Comm_rank(MPI_COMM_WORLD,&rank);
         dim[0]=4; dim[1]=3;
         period[0]=TRUE; period[1]=FALSE; reorder=TRUE;
         MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
         if(rank==9)
         { MPI_Cart_shift(vu,0,1,&left,&right);
           MPI_Cart_shift(vu,1,1,&up,&down);
           printf("P:%d I miei vicini sono destra:%d giù:%d sinistra:%d
                 sopra:%d\n", rank, right, down, left, up); }
         MPI_Finalize();
}
```

```
----------------------------
P:9 I miei vicini sono destra:0 giù:10 sinistra:6 sopra:-1
```

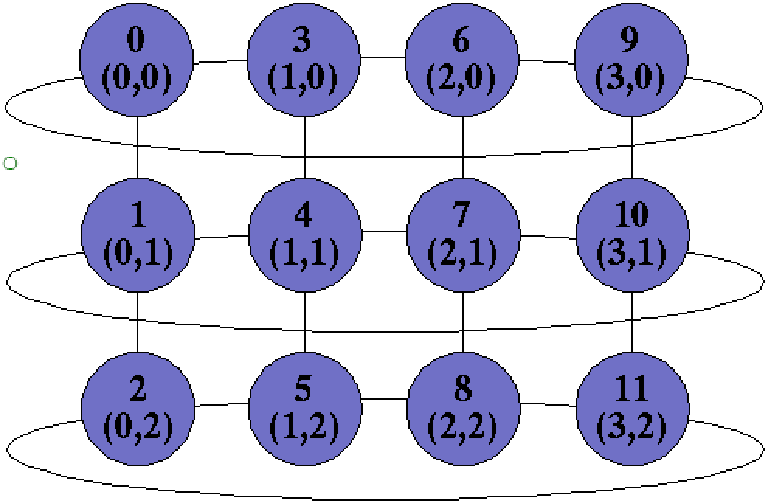
```c
// Il processo 7 spedisce un messaggio ai propri vicini
// NB No deadlock!

#include<mpi.h>
#define TRUE 1
#define FALSE 0
void main(int argc, char *argv[]){
    int rank, msg;
    MPI_Comm vu;
    MPI_Status status;
    int dim[2],period[2],reorder;
    int up,down,right,left;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    dim[0]=4; dim[1]=3;
    period[0]=TRUE; period[1]=FALSE; reorder=TRUE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
    MPI_Cart_shift(vu,0,1,&left,&right);
    MPI_Cart_shift(vu,1,1,&up,&down);

    if(rank==7)
    {
      printf("P:%d I miei vicini sono destra:%d giù:%d sinistra:%d sopra:%d\n",
          rank, right, down, left, up);
      MPI_Send(&rank, 1, MPI_INT, left, 0, vu);
      MPI_Send(&rank, 1, MPI_INT, up, 0, vu);
      MPI_Send(&rank, 1, MPI_INT, down, 0, vu);
      MPI_Send(&rank, 1, MPI_INT, right, 0, vu);

    }
    if (rank==4){
        MPI_Recv(&msg, 1, MPI_INT, right, 0, vu, &status);
        printf("P%d Ho rivecuto da %d\n", rank, msg);
    }
    if (rank==6){
        MPI_Recv(&msg, 1, MPI_INT, down, 0, vu, &status);
        printf("P%d Ho rivecuto da %d\n", rank, msg);
    }
     if (rank==10){
        MPI_Recv(&msg, 1, MPI_INT, left, 0, vu, &status);
        printf("P%d Ho rivecuto da %d\n", rank, msg);
    }
     if (rank==8){
            MPI_Recv(&msg, 1, MPI_INT, up, 0, vu, &status);
            printf("P%d Ho rivecuto da %d\n", rank, msg);
     }
 MPI_Finalize();
}
```



Ogni proc chiama MPI_Cart_shift

Gira solo con 12 procs!

Notate come “ricevo” in ordine:
ES: 4 riceve da right (=7)
7 spedisce a left (=4)



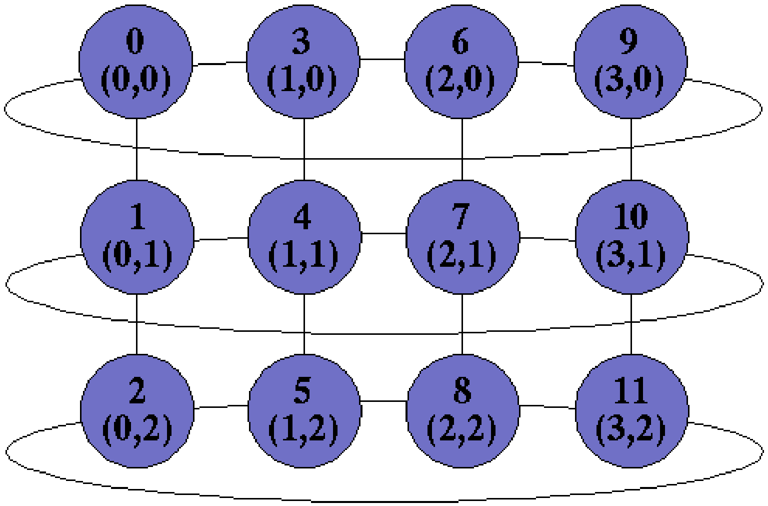
```c
// Ogni Processo spedisce ai propri vicini il proprio rango
// NB Dead lock?

#include<mpi.h>
#define TRUE 1
#define FALSE 0
void main(int argc, char *argv[]){
    int rank, msg;
    MPI_Comm vu;
    MPI_Status status;
    int dim[2],period[2],reorder;
    int up,down,right,left;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    dim[0]=4; dim[1]=3;
    period[0]=TRUE; period[1]=TRUE; reorder=FALSE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
    MPI_Cart_shift(vu,0,1,&left,&right);
    MPI_Cart_shift(vu,1,1,&up,&down);

    printf("P:%d I miei vicini sono destra:%d giù:%d sinistra:%d sopra:%d\n",
        rank, right, down, left, up);
    MPI_Send(&rank, 1, MPI_INT, left, 0, vu);
    MPI_Send(&rank, 1, MPI_INT, up, 0, vu);
    MPI_Send(&rank, 1, MPI_INT, down, 0, vu);
    MPI_Send(&rank, 1, MPI_INT, right, 0, vu);

    MPI_Recv(&msg, 1, MPI_INT, right, 0, vu, &status);
    printf("P%d Ho ricevuto da %d\n", rank, msg);
    MPI_Recv(&msg, 1, MPI_INT, down, 0, vu, &status);
    printf("P%d Ho ricevuto da %d\n", rank, msg);
    MPI_Recv(&msg, 1, MPI_INT, left, 0, vu, &status);
    printf("P%d Ho ricevuto da %d\n", rank, msg);
    MPI_Recv(&msg, 1, MPI_INT, up, 0, vu, &status);
    printf("P%d Ho ricevuto da %d\n", rank, msg);

MPI_Finalize();
}
```

Ogni processo calcola
i propri vicini

Gira solo con 12 procs!

Non “dovrebbe” andare in deadlock!
Spediamo solo un intero!!!
Meglio Send/Receive non Bloccanti!

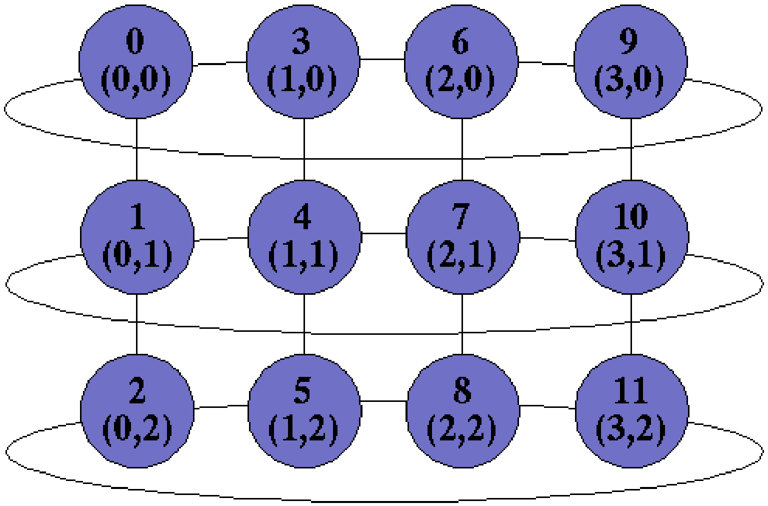
```c
// Ogni Processo spedisce ai propri vicini il proprio rango
// NB Dead lock!!!!
// perchè utilizzo per prima Receive bloccanti!


#include<mpi.h>
#define TRUE 1
#define FALSE 0
void main(int argc, char *argv[]){
    int rank, msg;
    MPI_Comm vu;
    MPI_Status status;
    int dim[2],period[2],reorder;
    int up,down,right,left;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    dim[0]=4; dim[1]=3;
    period[0]=TRUE; period[1]=TRUE; reorder=FALSE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
    MPI_Cart_shift(vu,0,1,&left,&right);
    MPI_Cart_shift(vu,1,1,&up,&down);

    printf("P:%d I miei vicini sono destra:%d giù:%d sinistra:%d sopra:%d\n"
        rank, right, down, left, up);

    MPI_Recv(&msg, 1, MPI_INT, right, 0, vu, &status);
    printf("P%d Ho ricevuto da %d\n", rank, msg);
    MPI_Recv(&msg, 1, MPI_INT, down, 0, vu, &status);
    printf("P%d Ho ricevuto da %d\n", rank, msg);
    MPI_Recv(&msg, 1, MPI_INT, left, 0, vu, &status);
    printf("P%d Ho ricevuto da %d\n", rank, msg);
    MPI_Recv(&msg, 1, MPI_INT, up, 0, vu, &status);
    printf("P%d Ho ricevuto da %d\n", rank, msg);

    MPI_Send(&rank, 1, MPI_INT, left, 0, vu);
    MPI_Send(&rank, 1, MPI_INT, up, 0, vu);
    MPI_Send(&rank, 1, MPI_INT, down, 0, vu);
    MPI_Send(&rank, 1, MPI_INT, right, 0, vu);


MPI_Finalize();
}
```



Ogni processo calcola i propri vicini

Gira solo con 12 procs!

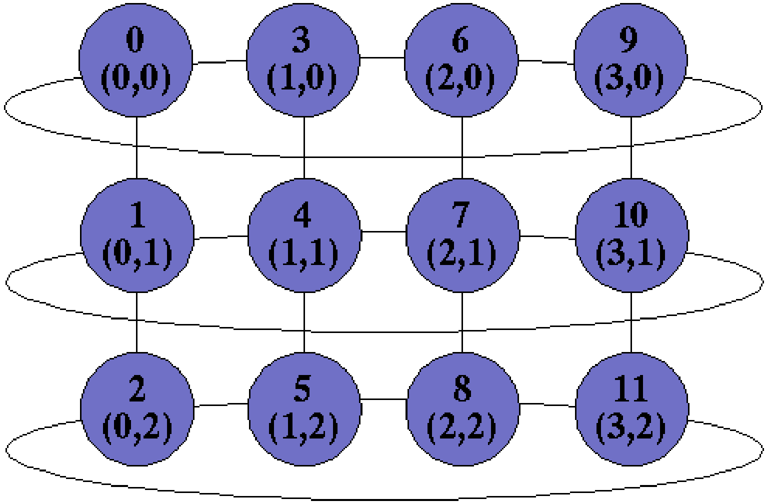
Ho sicuramente deadlock perchè utilizzo per prima receive bloccanti!
Utilizzando prima Send bloccanti, come prima, dovrei evitare deadlock (MPI a runtime decide buffered o sincrono...)



```
// Ogni Processo spedisce ai propri vicini il proprio rango
// NB Dead lock!!!
// Send sincrone

#include<mpi.h>
#define TRUE 1
#define FALSE 0
void main(int argc, char *argv[]){
    int rank, msg;
    MPI_Comm vu;
    MPI_Status status;
    int dim[2],period[2],reorder;
    int up,down,right,left;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    dim[0]=4; dim[1]=3;
    period[0]=TRUE; period[1]=TRUE; reorder=FALSE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
    MPI_Cart_shift(vu,0,1,&left,&right);
    MPI_Cart_shift(vu,1,1,&up,&down);

    printf("P:%d I miei vicini sono destra:%d giù:%d sinistra:%d sopra:%d\n",
        rank, right, down, left, up);
    MPI_Ssend(&rank, 1, MPI_INT, left, 0, vu);
    MPI_Ssend(&rank, 1, MPI_INT, up, 0, vu);
    MPI_Ssend(&rank, 1, MPI_INT, down, 0, vu);
    MPI_Ssend(&rank, 1, MPI_INT, right, 0, vu);

    MPI_Recv(&msg, 1, MPI_INT, right, 0, vu, &status);
    printf("P%d Ho ricevuto da %d\n", rank, msg);
    MPI_Recv(&msg, 1, MPI_INT, down, 0, vu, &status);
    printf("P%d Ho ricevuto da %d\n", rank, msg);
    MPI_Recv(&msg, 1, MPI_INT, left, 0, vu, &status);
    printf("P%d Ho ricevuto da %d\n", rank, msg);
    MPI_Recv(&msg, 1, MPI_INT, up, 0, vu, &status);
    printf("P%d Ho ricevuto da %d\n", rank, msg);

MPI_Finalize();
}
```

Send sincrone!
Deadlock sicuro!!!
Ogni processo si mette
a spedire e si aspetta una
receive corrispondente che
non c’e’!

```
// Ogni Processo spedisce ai propri vicini il proprio rango
// NB NO Deadlock!!!!
// Receive non bloccanti!
// Notate le printf dopo le Irecv che possono essere "sbagliate"


#include<mpi.h>
#define TRUE 1
#define FALSE 0
void main(int argc, char *argv[]){
    int rank, msg;
    MPI_Comm vu;
    MPI_Status status;
    MPI_Request req[4];
    int dim[2],period[2],reorder;
    int up,down,right,left;
    int i;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    dim[0]=4; dim[1]=3;
    period[0]=TRUE; period[1]=TRUE; reorder=FALSE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
    MPI_Cart_shift(vu,0,1,&left,&right);
    MPI_Cart_shift(vu,1,1,&up,&down);

    printf("P:%d I miei vicini sono destra:%d giù:%d sinistra:%d sopra:%d\n",
        rank, right, down, left, up);

    MPI_Irecv(&msg, 1, MPI_INT, right, 0, vu, &req[0]);
    printf("P%d Ho forse ricevuto da %d\n", rank, msg);
    MPI_Irecv(&msg, 1, MPI_INT, down, 0, vu, &req[1]);
    printf("P%d Ho forse ricevuto da %d\n", rank, msg);
    MPI_Irecv(&msg, 1, MPI_INT, left, 0, vu, &req[2]);
    printf("P%d Ho forse ricevuto da %d\n", rank, msg);
    MPI_Irecv(&msg, 1, MPI_INT, up, 0, vu, &req[3]);
    printf("P%d Ho forse ricevuto da %d\n", rank, msg);


    MPI_Send(&rank, 1, MPI_INT, left, 0, vu);
    MPI_Send(&rank, 1, MPI_INT, up, 0, vu);
    MPI_Send(&rank, 1, MPI_INT, down, 0, vu);
    MPI_Send(&rank, 1, MPI_INT, right, 0, vu);

    for (i=0; i<4; i++)
          MPI_Wait(&req[i], &status);

    printf("P%d Ho sicuramente ricevuto da tutti \n", rank);


MPI_Finalize();
}
```



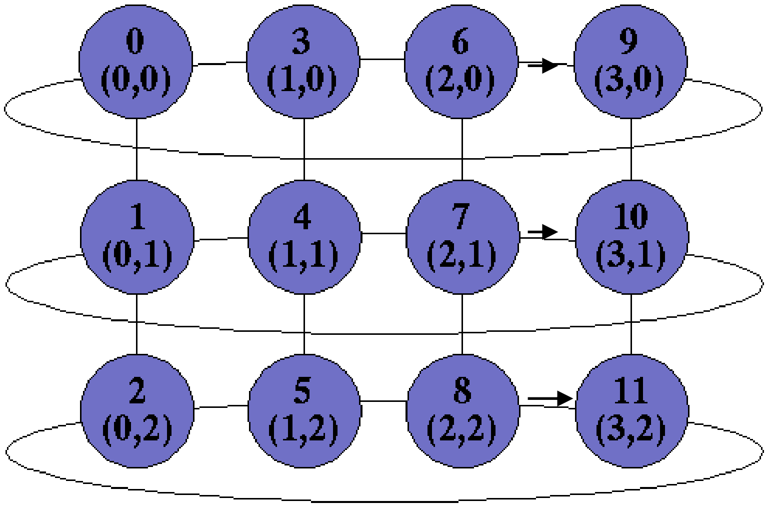
Gira solo con 12 procs!

Receive non bloccanti!
`msg` potrebbe essere “indefinito”

Potrei anche rendere queste
non bloccanti (v. dopo) – Fatelo!

Devo “aspettare” tutti!

Sicuramente tutti hanno ricevuto!



```
// Ogni Processo della colonna 2 spedisce un messaggio
// ai processi a destra
// NB Dead lock?

#include<mpi.h>
#define TRUE 1
#define FALSE 0
void main(int argc, char *argv[]){
    int rank, rankc, msg;
    MPI_Comm vu;
    MPI_Status status;
    int coords[2];
    int dim[2],period[2],reorder;
    int up,down,right,left;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    dim[0]=4; dim[1]=3;
    period[0]=TRUE; period[1]=FALSE; reorder=TRUE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
    MPI_Cart_shift(vu,0,1,&left,&right);
    MPI_Cart_shift(vu,1,1,&up,&down);

    MPI_Cart_coords(vu, rank, 2, coords);

    if (coords[0]==2){

      printf("P:%d I miei vicini sono destra:%d giù:%d sinistra:%d sopra:%d\n",
          rank, right, down, left, up);
      MPI_Send(&rank, 1, MPI_INT, right, 0, vu);
    }

    if (coords[0]==3){
        MPI_Recv(&msg, 1, MPI_INT, left, 0, vu, &status);
        printf("P%d Ho rivecuto da %d\n", rank, msg);
    }

MPI_Finalize();
}
```

Gira solo con 12 procs!

Individuo la colonna 2

Colonna 3

No deadlock!

Gira solo con 16 procs!

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Output

```
rank= 0 coords= 0 0  neighbors(u,d,l,r)= -3 4 -3 1
rank= 0                inbuf(u,d,l,r)= -3 4 -3 1
rank= 1 coords= 0 1  neighbors(u,d,l,r)= -3 5 0 2
rank= 1                inbuf(u,d,l,r)= -3 5 0 2
rank= 2 coords= 0 2  neighbors(u,d,l,r)= -3 6 1 3
rank= 2                inbuf(u,d,l,r)= -3 6 1 3
        . . . . .

rank= 14 coords= 3 2  neighbors(u,d,l,r)= 10 -3 13 15
rank= 14                inbuf(u,d,l,r)= 10 -3 13 15
rank= 15 coords= 3 3  neighbors(u,d,l,r)= 11 -3 14 -3
rank= 15                inbuf(u,d,l,r)= 11 -3 14 -3
```

Determina i vicini! (vicinato von Neumann)

No bloccanti?
No Deadlock!

1. Creazione di una topologia 4 x 4 Cartesiana da 16 processori –
2. Scambiare il proprio rango con i 4 vicini.

```c
#include "mpi.h"
#include <stdio.h>
#define SIZE 16
#define UP    0
#define DOWN  1
#define LEFT  2
#define RIGHT 3

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source, dest, outbuf, i, tag=1,
   inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,},
   nbrs[4], dims[2]={4,4},
   periods[2]={0,0}, reorder=0, coords[2];

MPI_Request reqs[8];
MPI_Status stats[8];
MPI_Comm cartcomm;

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

 if (numtasks == SIZE) {
   MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
   MPI_Comm_rank(cartcomm, &rank);
   MPI_Cart_coords(cartcomm, rank, 2, coords);
   MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
   MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);

   outbuf = rank;

   for (i=0; i<4; i++) {
      dest = nbrs[i];
      source = nbrs[i];
      MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
                MPI_COMM_WORLD, &reqs[i]);
      MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
                MPI_COMM_WORLD, &reqs[i+4]);
      }

   MPI_Waitall(8, reqs, stats);

   printf("rank= %d coords= %d %d  neighbors(u,d,l,r)= %d %d %d %d\n",
          rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],
          nbrs[RIGHT]);
   printf("rank= %d                  inbuf(u,d,l,r)= %d %d %d %d\n",
          rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]);
   }
 else
   printf("Must specify %d processors. Terminating.\n",SIZE);

 MPI_Finalize();
 }
```