We must communicate data. This is a cooperative process, and can only occur when both the first process executes a send operation, and the second process executes a receive operation.

Sender:

- The data to communicate.
- The amount of data to communicate.
- The destination for the message.
- A message tag so that the receiver can know which message is arriving.

send(address, length, destination, tag)

MPI_Send (&buf,count,datatype,dest,tag,comm)

MPI Datatype is very similar to a C or Fortran datatype

- int $\rightarrow$ MPI_INT
- double $\rightarrow$ MPI_DOUBLE
- char $\rightarrow$ MPI_CHAR

More complex datatypes are also possible:

- E.g., you can create a structure datatype that comprises of other datatypes: a char, an int and a double.
- Or, a vector datatype for the columns of a matrix

The "count" in MPI_SEND and MPI_RECV refers to how many datatype elements should be communicated

We must communicate data. This is a cooperative process, and can only occur when both the first process executes a send operation, and the second process executes a receive operation.
Receiver:

- The starting memory address.
- The amount of data to communicate.
- The identity of the sender.
- The message tag.

receive(address, length, source, tag, actual_length)
In **C**, MPI_Recv (&buf,count,datatype,source,tag,comm,&status)

MPI Subroutine	Description
MPI_Init	Initializes MPI
MPI_Comm_size	Finds out the total number of processes
MPI_Comm_rank	Finds out this processor's ID
MPI_Send	Sends a message
MPI_Recv	Receives a message
MPI_Finalize	Terminates MPI

- MPI_Ini: In **C**, MPI_Init (&argc,&argv)
  This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.
- MPI_Comm_size: In **C**, MPI_Comm_size (comm,&size)
  Returns the total number of MPI processes.
- MPI_Comm_rank: In **C**, MPI_Comm_rank (comm,&rank)
  Returns the rank (ID) of the calling MPI process. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1.

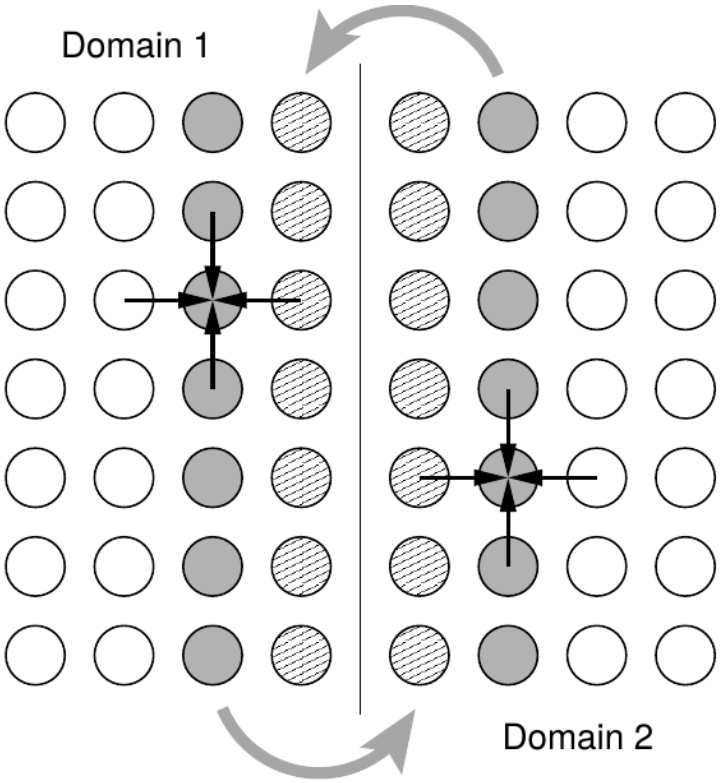
```c
/* Inclusions */
#include <stdlib.h>   /* malloc(), free() */
#include <stdio.h>    /* printf() */
#include <time.h>     /* clock() */

/* Prototypes */
inline double f(double a) { return (4.0 / (1.0 + a*a)); }

/* Example routine to compute pi using numerical integration via
   pi = 4 * int_0^1 1/(1+x^2) dx. We use a simple midpoint rule for integration, over
   subintervals of fixed size 1/n, where n is a user-input parameter. */
int main(int argc, char* argv[]) {

  /* input the number of intervals */
  int n;
  printf("Enter the number of intervals (0 quits):\n");
  scanf("%i", &n);
  if (n < 1) {
    return(-1);
  }

  /* start timer */
  time_t stime = time(NULL);

  /* set subinterval width */
  double h = 1.0 / n;

  /* perform integration over n intervals */
  double x, pi=0.0;
  int i;
  for (i=0; i<n; i++) {
    x = h * (i + 0.5);
    pi += h * f(x);
  }

  /* stop timer */
  time_t ftime = time(NULL);
  double runtime = ((double) (ftime - stime));

  /* output computed value and error */
  double pi_true = 3.14159265358979323846;
  printf(" computed pi = %.16e\n",pi);
  printf("     true pi = %.16e\n",pi_true);
  printf("       error = %.16e\n",pi_true - pi);
  printf("     runtime = %.16e\n",runtime);

} /* end main */
```

Homework: parallelizing the serial π computation program (i) using openMP (ii) using MPI

Basics of parallelization

Data parallelism

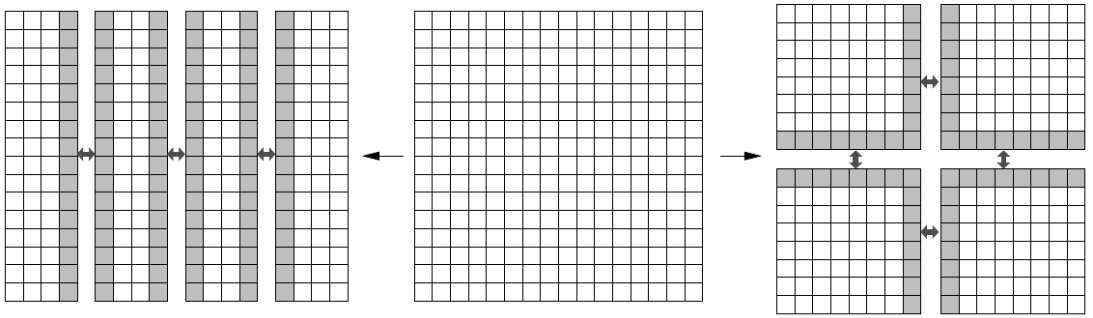
- Medium-grained loop parallelism
- Coarse-grained parallelism by domain decomposition



Load balancing: computational effort should be equal for all domains

Domain decomposition

After load balance, it comes reducing the communication overhead



Cutting into stripes is simple but needs more communication than optimal decomposition.

Scalability metrics

- Algorithmic limitations
- Serialized executions
- Startup overhead
- Communication

Scalability metrics

The overall size of the problem is $T^s = s + p$, where s is the serial (nonparallelizable) part and p is the perfectly parallelizable fraction.
Strong scaling: solving the same problem on N workers will require a runtime of

$$T^p = s + p/N$$

where N is the number of workers.
Weak scaling: scale the problem size

$$T^p = s + pN$$

Application speedup can be defined as the quotient of parallel and serial performance for fixed problem size.

Scalability metrics

Speedup:

$$P^p = \frac{1}{s + \frac{1-s}{N}}$$

Scalability:

$$S^p = \frac{1}{s + \frac{1-s}{N}}$$

For a fixed problem size, scalability is limited.

Scalability metrics

In the case of weak scaling

$$P^p = S^p = \frac{s + (1-s) * N^{\alpha}}{s + (1-s) * N^{\alpha-1}}$$

Weak scaling allows unlimitted performace.
Parallel efficiency = speedup/N

$$\epsilon = s/N + 1 - s$$