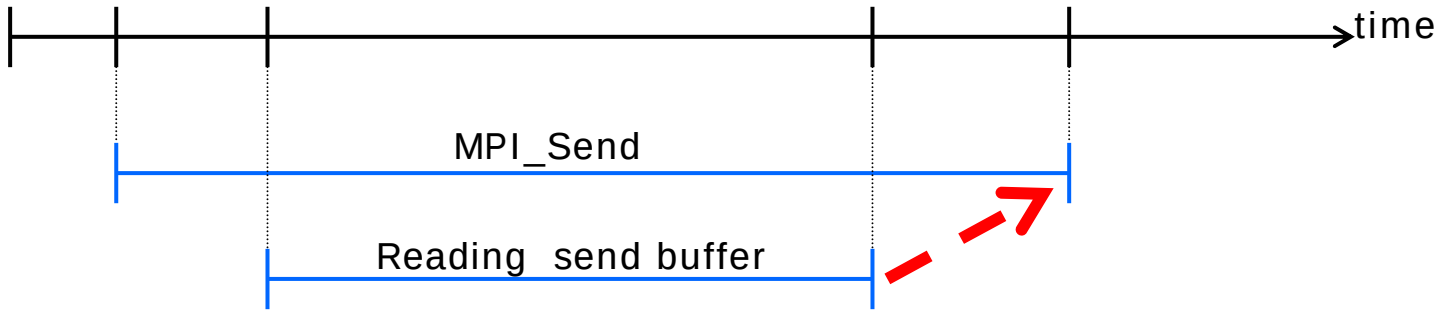
Blocking / Non-Blocking Send and Receive Operations

- `MPI_Send(void *buf, int count, …)`
  `MPI_Recv(void *buf, int count, …)`

- *Blocking Operations*: Send- and receive buffers can be used safely after the execution of the command

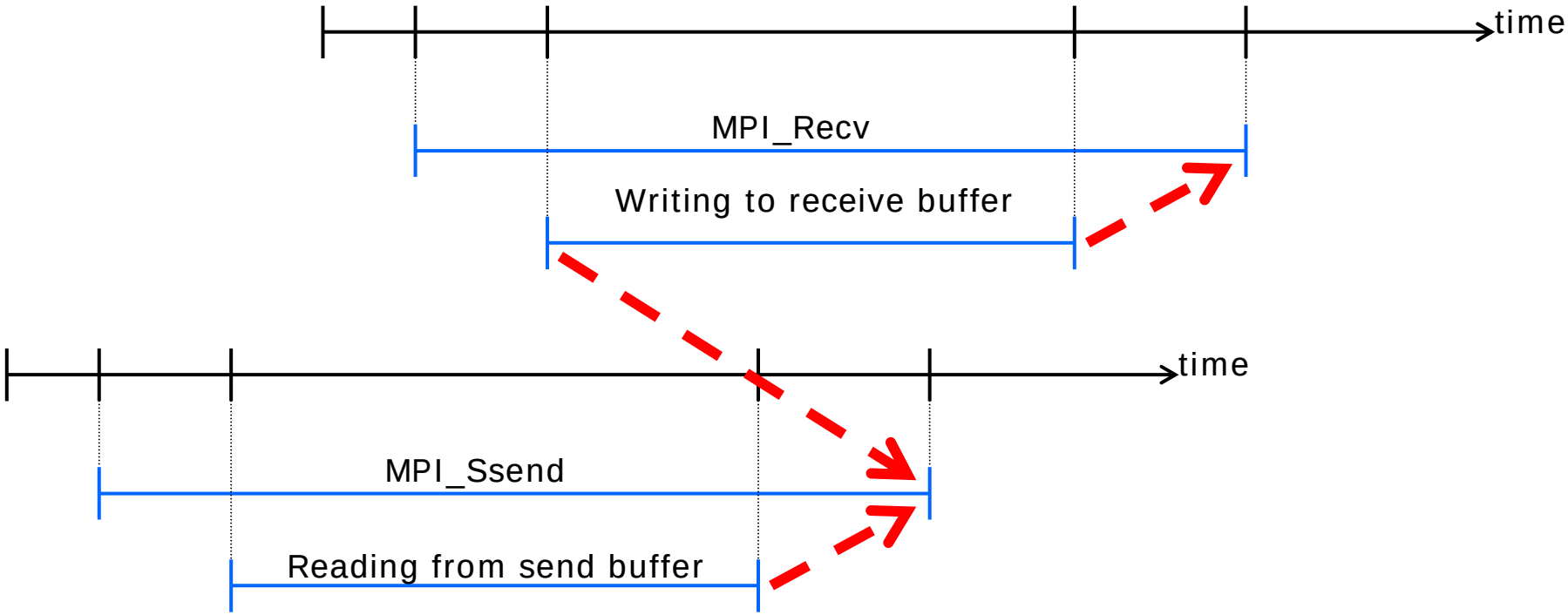


blocking receive analogously

Synchronous Send

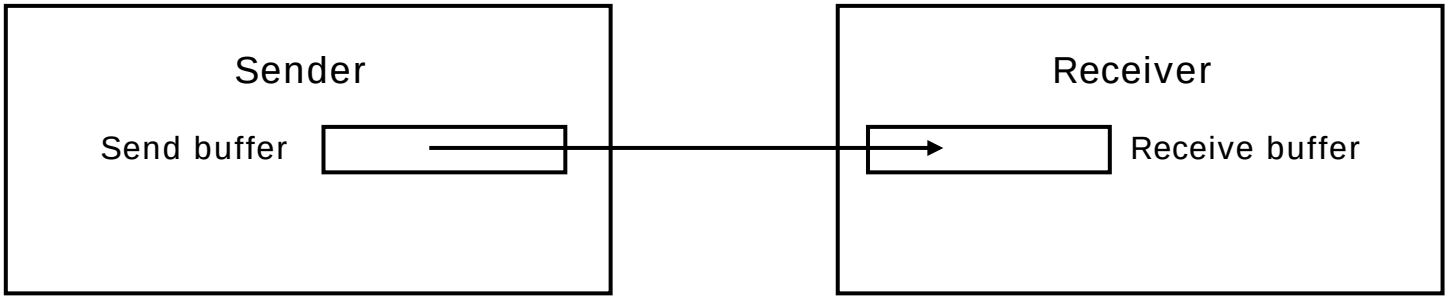
- *Synchronous Send*: MPI_Ssend does not return, before the corresponding receive operation started

  Blocking version:

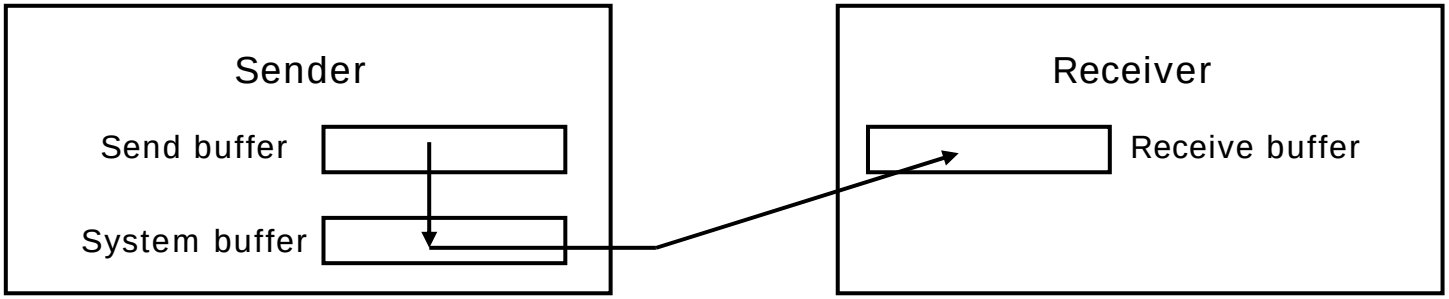


Buffered / Unbuffered Send and Receive Operations

- Unbuffered:

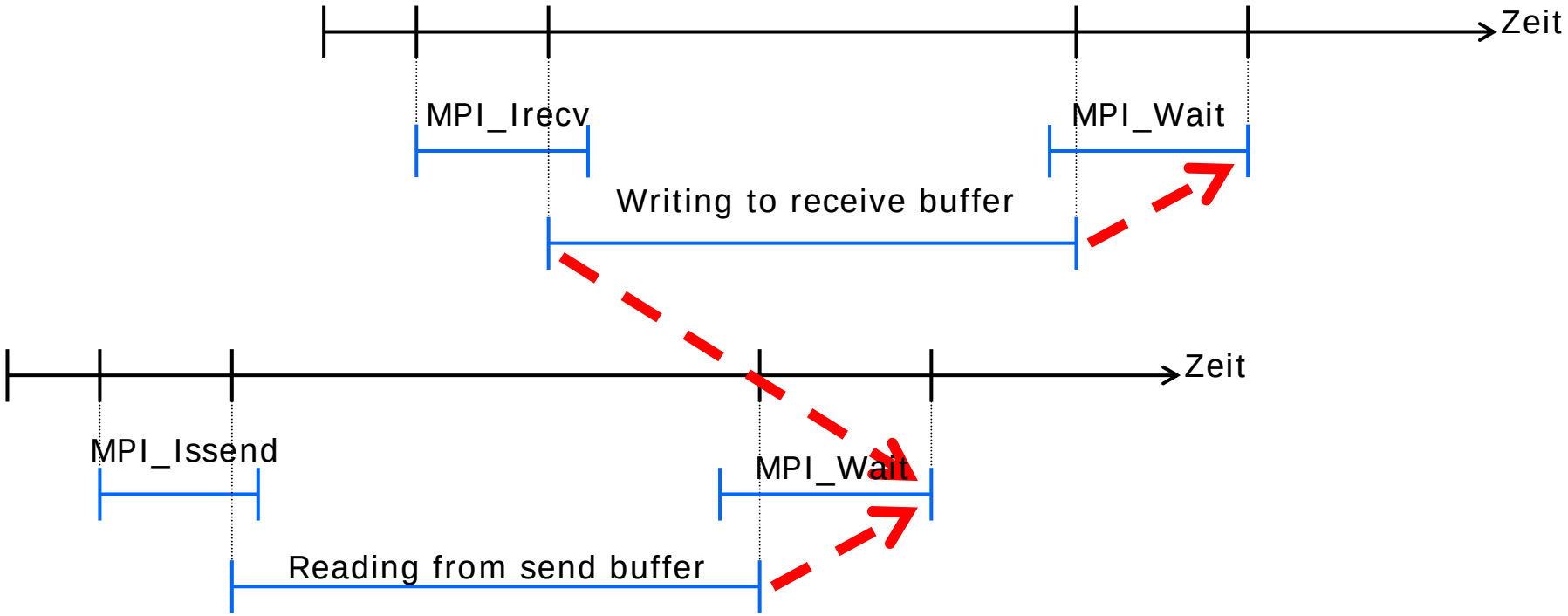


- Buffered:



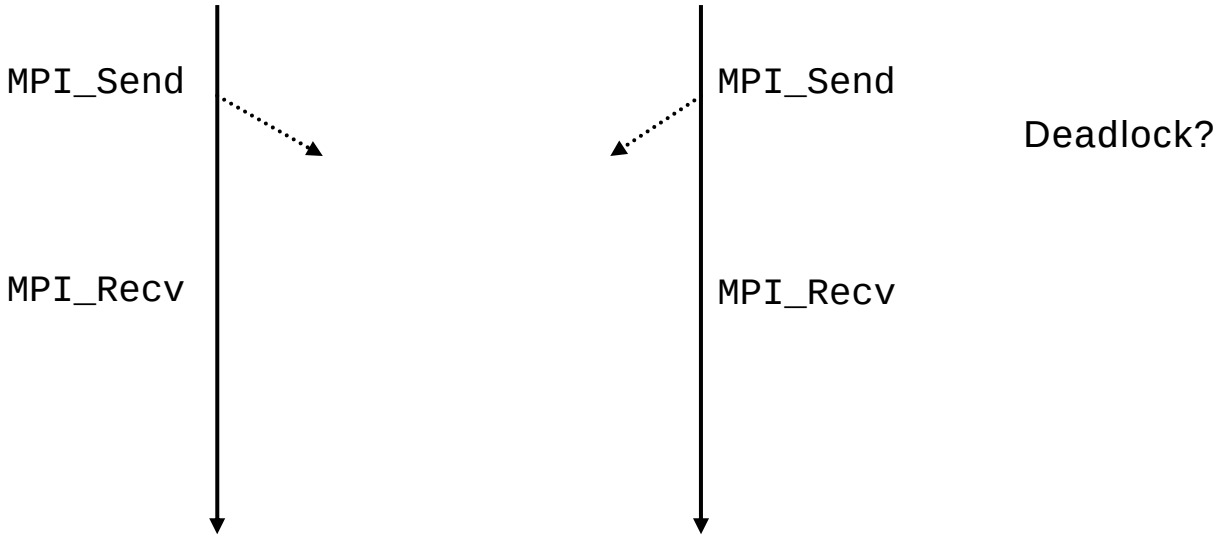
Synchronous Send

- Non-blocking:



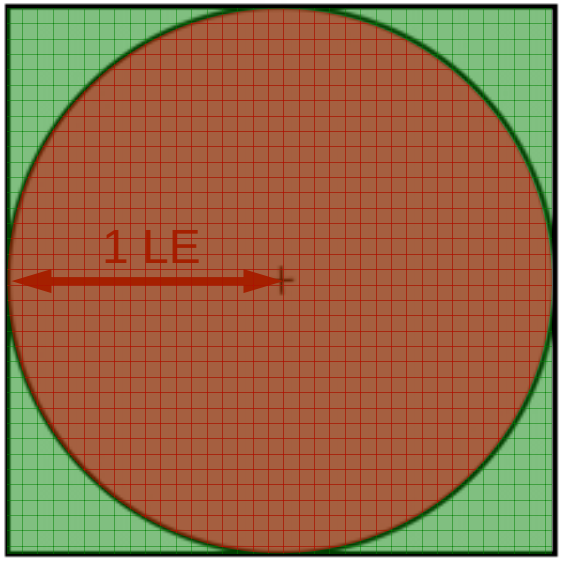
Deadlocks / Standard Send

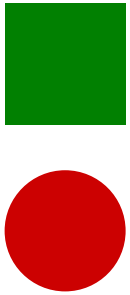
- `MPI_Send`: Could be synchronous or buffered (i.e. `MPI_Send` *could* wait for the corresponding receive)



MPI: Example

(A rather stupid way of) computing π via Monte-Carlo method





: area 4

: area π

$$\frac{\text{area of circle}}{\text{area of square}} = \frac{\pi}{4}$$

```c
/* compute pi using Monte Carlo method */
#include <math.h>
#include "mpi.h"
#include "mpe.h"

#define CHUNKSIZE   1000
/* Message Tags */
#define REQUEST 1
#define REPLY   2

int main(int argc, char *argv[])
{
    ...
    int in, out, totalin, totalout, myid, numprocs;
    int rands[CHUNKSIZE];
    double x, y, Pi, error, epsilon;
    MPI_Comm world, workers;
    ...

    /* MPI-Initialisation */
    MPI_Init(&argc, &argv);
    world = MPI_COMM_WORLD;
    ...
    MPI_Comm_size(world, &numprocs);
    MPI_Comm_rank(world, &myid);
    server = numprocs - 1; /* last process is server */
    if (myid == 0)
        sscanf(argv[1], "%lf", &epsilon);
    MPI_Bcast(&epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ...
```

```
32
33
34      /* server process */
35      if (myid == server){
36          do {
37              MPI_Recv(&request, 1, MPI_INT,
38                       MPI_ANYSOURCE, REQUEST, world, &status);
39              if (request) {
40                  for (i = 0; i < CHUNKSIZE; i++)
41                      rands[i] = random();
42                  MPI_Send(rands, CHUNKSIZE, MPI_INT,
43                           status.MPI_SOURCE, REPLY, world);
44              }
45          }
46          while (request > 0);
47      }
48
49
```

```
50      /* worker process */
51      else {
52          request = 1;
53          done = in = out = 0;
54          ...
55          MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
56          ...
57
58          while (!done) {
59              request = 1;
60              MPI_Recv(rands, CHUNKSIZE, MPI_INT,
61                       server, REPLY, world, &status);
62              for (i = 0, i < CHUNKSIZE;) {
63                  x = (((double) rands[i++])/max) * 2 - 1;
64                  y = (((double) rands[i++])/max) * 2 - 1;
65                  if (x*x + y*y < 1.0)
66                      in++;
67                  else
68                      out++;
69              }
70              MPI_Allreduce(&in,  &totalin,  1, MPI_INT, MPI_SUM,
71                            workers);
72              MPI_Allreduce(&out, &totalout, 1, MPI_INT, MPI_SUM,
73                            workers);
74              ...
75              done = (error < epsilon);
76              request = (done) ? 0 : 1;
77              MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
78              ...
79          }
80      }
81
```