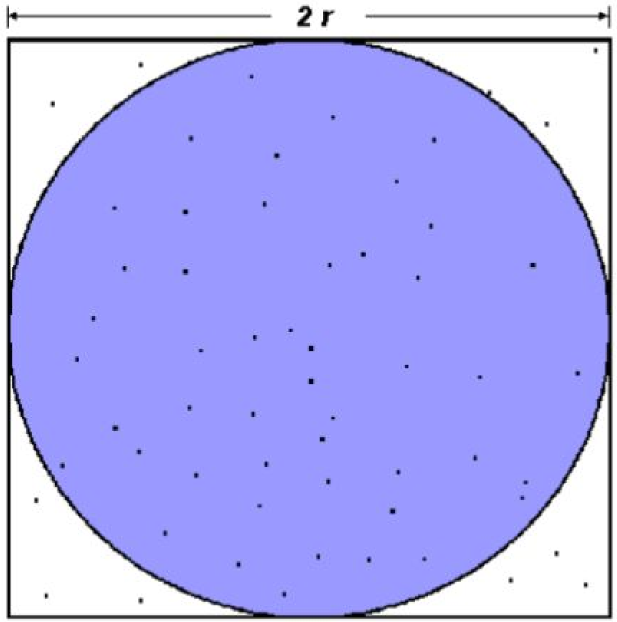
URL sources

https://github.com/lstorchi/mpisttest

Compute PI using a Monte Carlo Approach

A circle of radius R is inscribed inside a square with side length 2*R, if so the area of the circle will be $A_c = PI*R^2$ and the area of the square will be $A_s = (2*R)^2$. So the ratio of the area of the circle to the area of the square will be $A_c / A_s = PI/4$.



$A_S = (2r)^2 = 4r^2$

$A_C = \pi r^2$

$\pi = 4 \times \frac{A_C}{A_S}$

If a program picks N points (x, y) at random inside the square. If a point is inside the circle (i.e. if $x^2 + y^2 < R^2$) M is incremented by one.

Thus finally: PI =- 4 * M / N

Pseudo code

```
N = 2000
circle_count = 0
for i = 1 to N
    x = random value (0.0, 1.0)
    y = random value (0.0,  1.0)
    if x2 + y2 < 1.0
        circle_count = circle_count + 1
    endif
endfor
pi = 4 * (circle_count / N)
```

Parallel version

- Each one of the P MPI processes will generate N/P random points (clue each process should use a different seed)
- You need to sum the final value of circle_count (you may need to use the MPI_Reduce)
- As in the serial code you may now estimate the PI value
- N could be a command line argument

Exercise

- Implement the serial version of the code starting from the pseudo code (suggestion the number of random points can be a command line argument)
- Implement the parallel version and, depending on the number of cores of the VM you are using, calculate the speedup (**MPI_Wtime() Returns time in seconds since an arbitrary time in the past. clock_t clock(void) returns the number of clock ticks elapsed since the program was launched. To get the number of seconds used by the CPU, you need to divide by CLOCKS_PER_SEC.**)

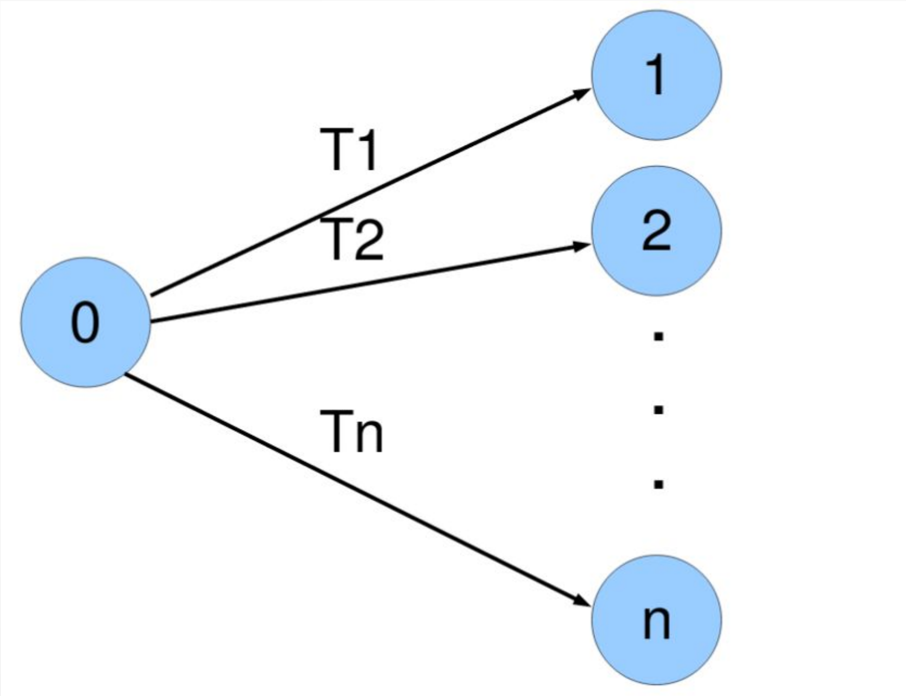
MPI functions (C API)

- `MPI_Init (&argc, &argv);`
- `MPI_Comm_size (MPI_COMM_WORLD, &size);`
- `MPI_Comm_rank (MPI_COMM_WORLD, &rank);`
- `MPI_Barrier (MPI_COMM_WORLD);`
- `MPI_Reduce (&circle_count, &t_circle_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);`
- `MPI_Finalize ();`

Broadcast

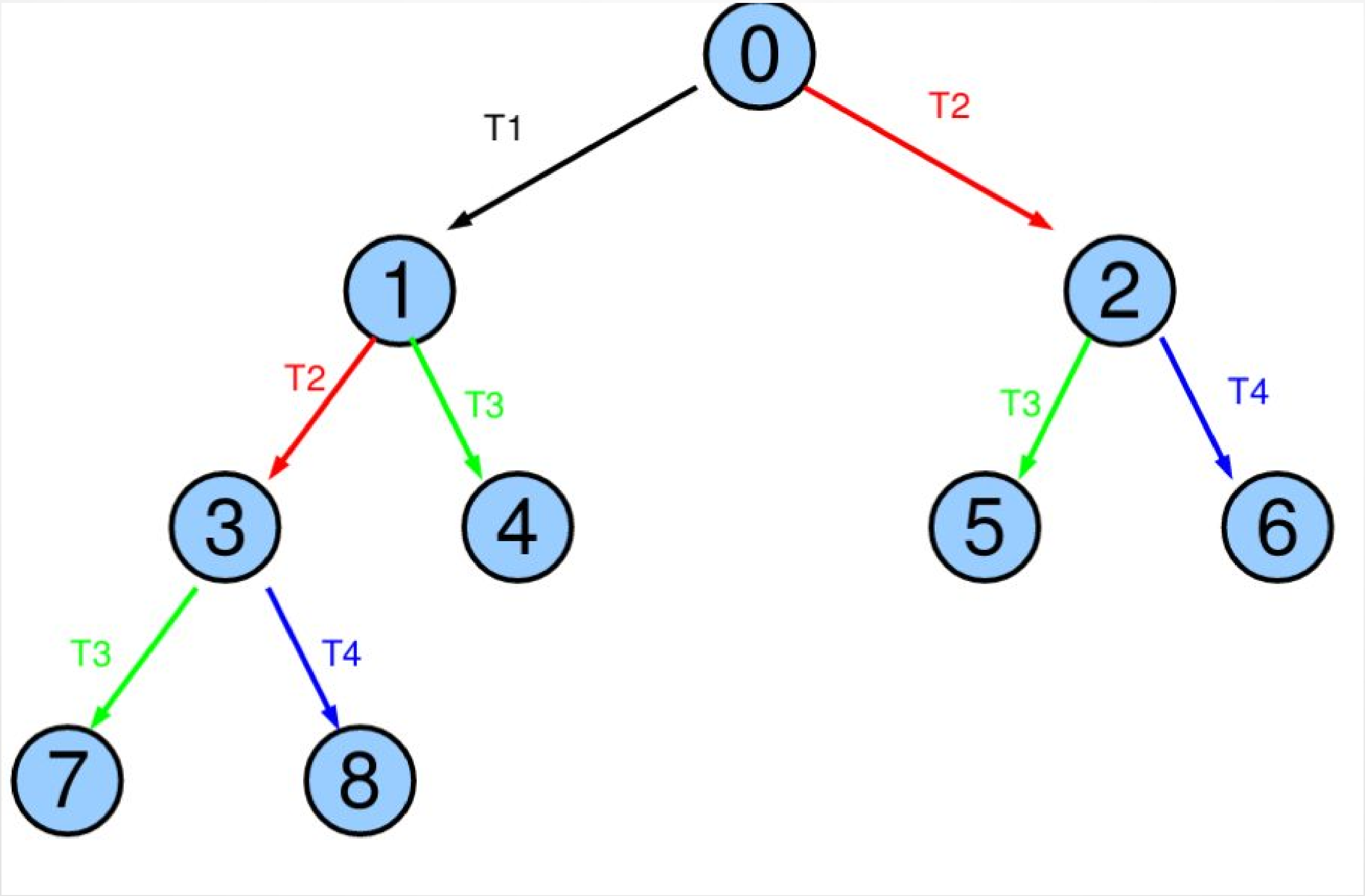
We will explore the differences between a naive approach to perform a broadcast and a more sophisticated one

Naive (flat tree) need n-1 point to point communications, T(msize) = time to send a message of size msize, so Time = (n-1) * T(msize) so O(n)



Broadcast

Now using a binary tree we are able to reduce the communication time (Time = $O(\log_2(n))$)



Bcast pseudocode

```
fromrank = (int) ((myrank-1)/2)

if (myrank > 0)
    Recv data from fromrank

torank1 = 2 * myrank + 1;
torank2 = 2 * myrank + 2;

if (torank1 < size)
    Send data to torank1
if (torank2 < size)
    Send data to torank2
```

Exercise

- Implement the two version of the broadcast the one using the flat tree and the other using the binary tree. You will breadcast in both cases a vector of dimension N , where N again could be a command line argument.

MPI functions (C API)

- `int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`

```
MPI_Send (sbuf, bufdim, MPI_DOUBLE, torank1, torank1, MPI_COMM_WORLD);
```

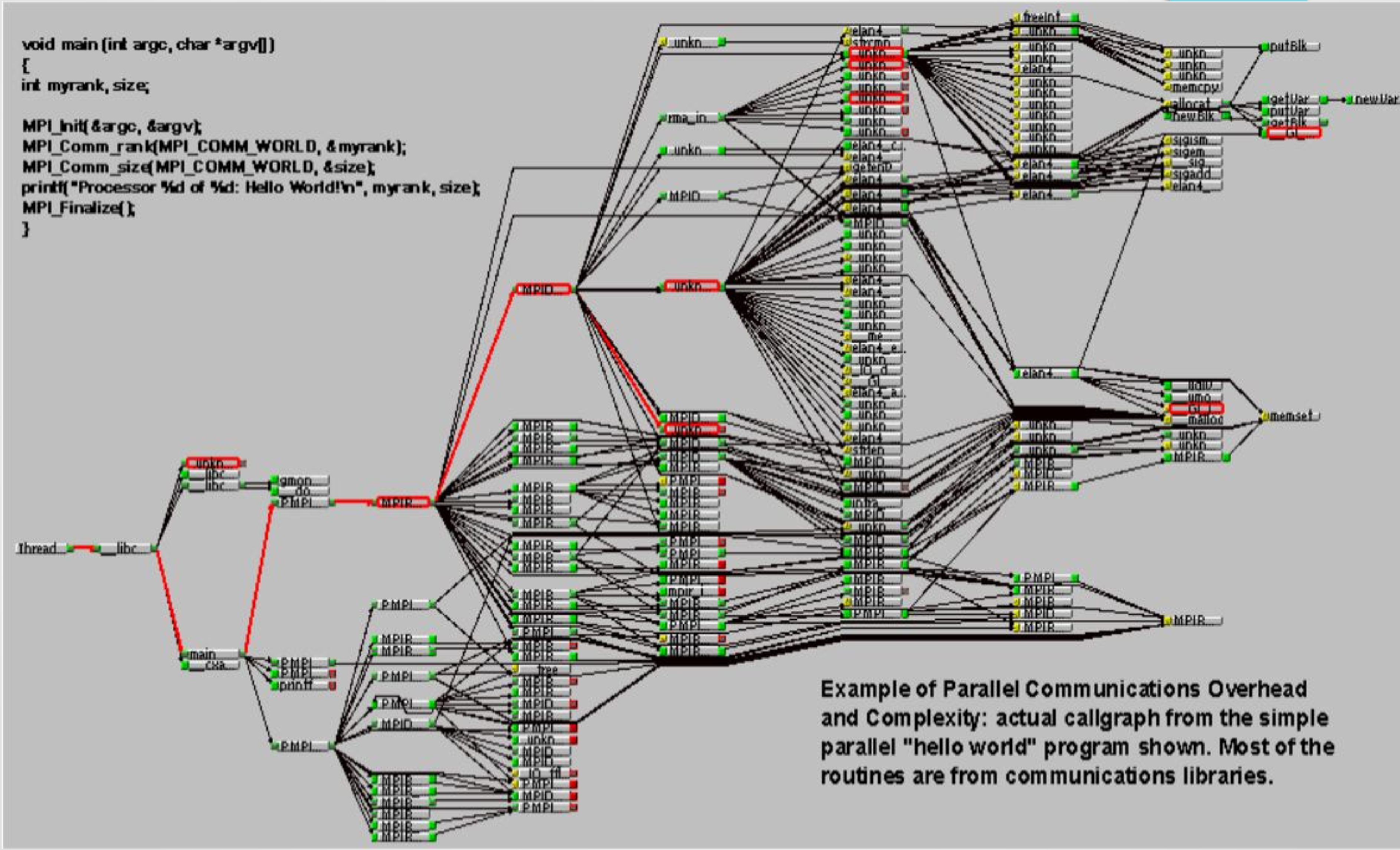
MPI functions (C API)

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`

```
MPI_Status status;

MPI_Recv (sbuf, bufdim, MPI_DOUBLE, fromrank, myrank, MPI_COMM_WORLD, &status);
```

MPI complexity

```
void main (int argc, char *argv[])
{
int myrank, size;

MPI_Init( &argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf( "Processor %d of %d: Hello World!\n", myrank, size);
MPI_Finalize( );
}
```



Example of Parallel Communications Overhead and Complexity: actual callgraph from the simple parallel "hello world" program shown. Most of the routines are from communications libraries.

Serial code optimization

Parallel computing era, however …. to be cache friendly:

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

```
for (i=0; i<N; i++)
  for (k=0; k<N; k++)
    for (j=0; j<N; j++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Serial code optimization

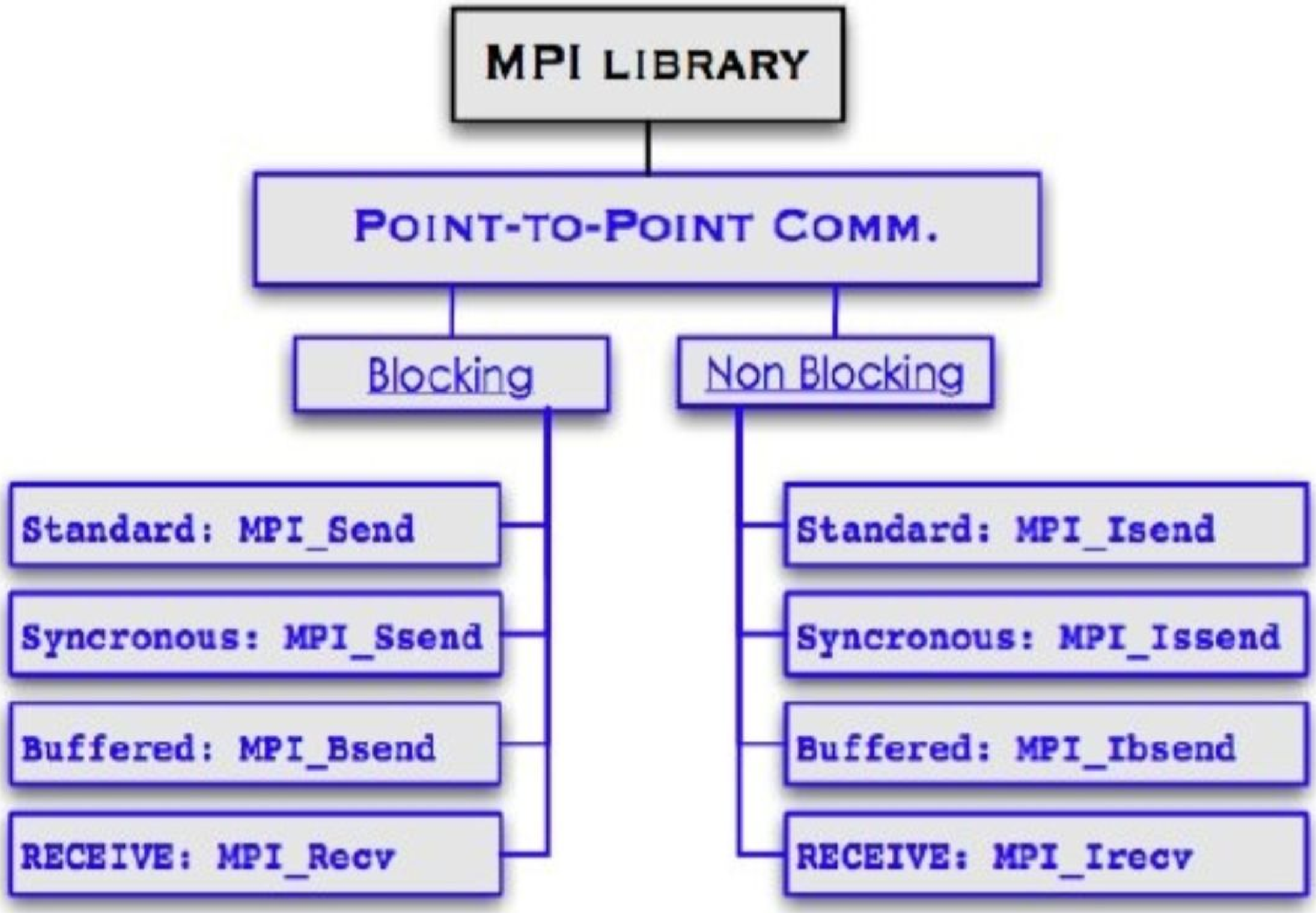
```
[redo@banquo serialopt (master)]$ ./mm.1
Time to initialize 0.050375 s.
Time 10.007438 s.
Total time 10.057895 s.

Mflops ----------------> 213.512236
Check -------------> 268364458.846206
[redo@banquo serialopt (master)]$ ./mm.3
Time to initialize 0.027267 s.
Time 2.971154 s.
Total time 2.998506 s.

Mflops ----------------> 716.184543
Check -------------> 268364458.846206
```

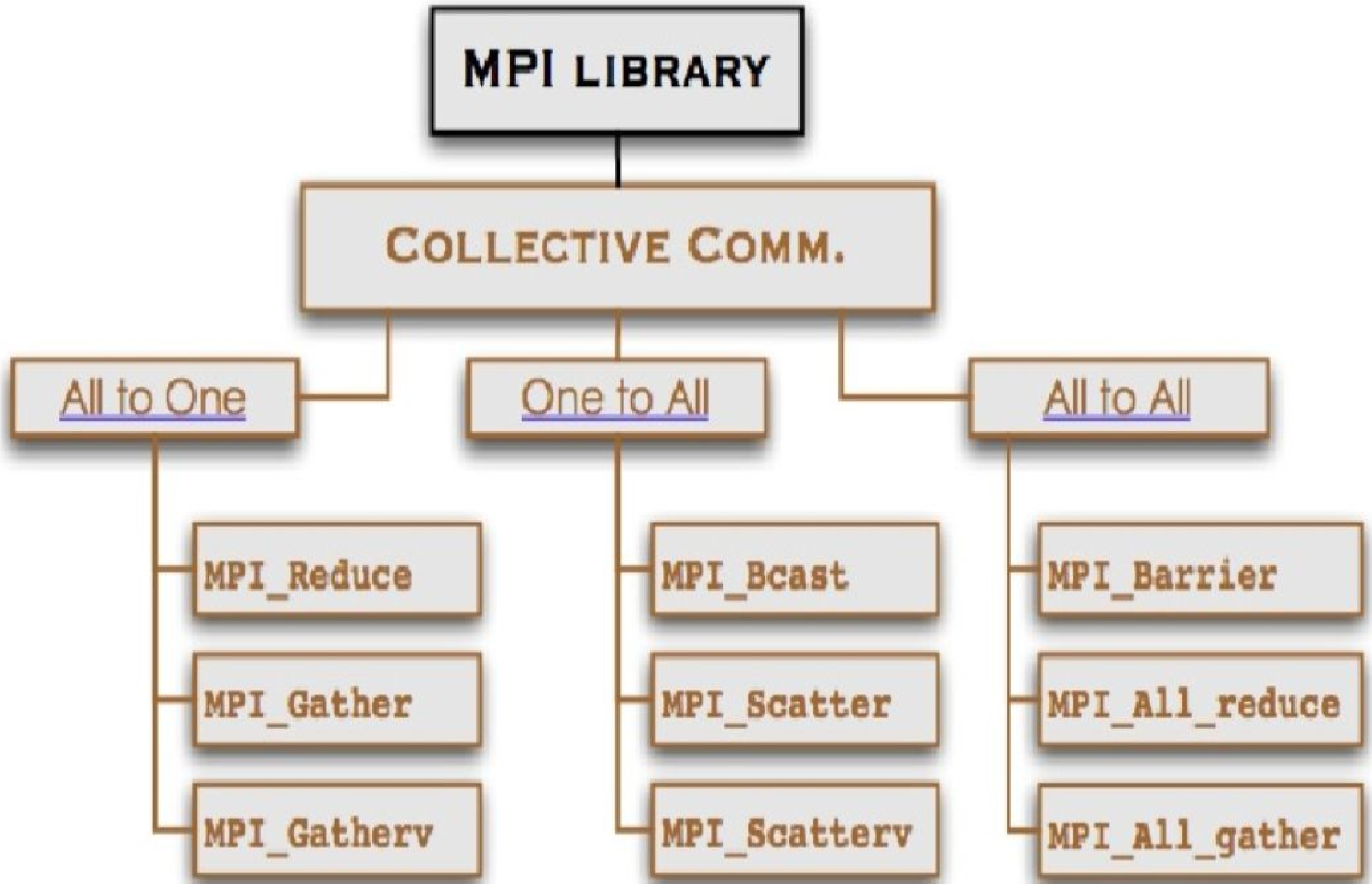
```
[redo@banquo serialopt (master)]$ diff mm.1.c mm.3.c
37,38c37,38
<       for (j=0; j<N; j++)
<         for (k=0; k<N; k++)
---
>       for (k=0; k<N; k++)
>         for (j=0; j<N; j++)
```

Serial code optimization

Keep the pipeline full, loop unrolling:

```
for (i=0; i<N; i++) {
  for (k=0; k<N; k++) {
   for (j=0; j<N; j +=8) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
      c[i][j+1] = c[i][j+1] + a[i][k] * b[k][j+1];
      c[i][j+2] = c[i][j+2] + a[i][k] * b[k][j+2];
      c[i][j+3] = c[i][j+3] + a[i][k] * b[k][j+3];
      c[i][j+4] = c[i][j+4] + a[i][k] * b[k][j+4];
      c[i][j+5] = c[i][j+5] + a[i][k] * b[k][j+5];
      c[i][j+6] = c[i][j+6] + a[i][k] * b[k][j+6];
      c[i][j+7] = c[i][j+7] + a[i][k] * b[k][j+7];
    }
  }
}
```


MPI LIBRARY
POINT-TO-POINT COMM.
Blocking
Non Blocking
Standard: MPI_Send
Syncronous: MPI_Ssend
Buffered: MPI_Bsend
RECEIVE: MPI_Recv
Standard: MPI_Isend
Syncronous: MPI_Issend
Buffered: MPI_Ibsend
RECEIVE: MPI_Irecv


MPI LIBRARY
COLLECTIVE COMM.
All to One
One to All
All to All
MPI_Reduce
MPI_Gather
MPI_Gatherv
MPI_Bcast
MPI_Scatter
MPI_Scatterv
MPI_Barrier
MPI_All_reduce
MPI_All_gather