MPI (Message Passing Interface)

- Developed and maintained by a large consortium of industry, academic, and government contributors
- De facto message passing standard for parallel computing
- Language bindings for C and Fortran (and formerly C++)
- Available on virtually all platforms; both proprietary and open source (MPICH) versions
  - [‘CH’ for Chameleon, the underlying communication library of the original MPICH implementation.]
- Grew out of an earlier message passing system, PVM (Parallel Virtual Machine), no longer actively maintained.
  - MPI-2 subsumed all significant PVM functionality.
  - Current version (2012) is MPI-3.

Compiling and Running

- `mpicc –o myprog myprog.c`
- `mpiexec [–n 16] [-configfile foo]`
  `./myprog <args>`
- Creates specified number of Unix processes, on nodes listed in the specified file
  - Watch Piazza for more detailed notes

MPI Process Creation/Destruction

MPI_Init(int *argc, char ***argv)

Initializes the MPI execution environment.

(Note the extra level of indirection.)

MPI_Finalize(void)

Terminates MPI execution environment. If you forget to make all processes call this routine, you may leave orphans on the system (run ps on all nodes to find them).

MPI Process Identification

MPI_Comm_size(comm, &size)

Determines the number of processes

MPI_Comm_rank(comm, &pid)

Pid is the process identifier of the caller (consecutive ints, starting with 0)

comm is typically MPI_COMM_WORLD — all processes in the program.

NB: by default the program dies if any call fails; return values can safely be ignored in C and C++.

MPI Basic Send

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

buf: address of send buffer

count: number of elements

datatype: data type of send buffer elements
(built-in or user-defined)

dest: process id (rank) of destination process

tag: message tag (ignore for now)

comm: communicator (e.g., MPI_COMM_WORLD)

MPI Basic Receive

MPI_Recv(buf, count, datatype, source, tag, comm, &status)

buf: address of receive buffer

count: size of receive buffer in elements

datatype: data type of receive buffer elements

source: source process rank or MPI_ANY_SOURCE

tag and comm: ignore for now

status: status object (indicates sender and tag)

MPI Matrix Multiply (initial C version)

```
int me, p;  const int N = …
int (*a)[N], (*b)[N], (*c)[N];
int main(int argc, char *argv[])
{
    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    /* malloc space for a, b, and c */
    /* Data distribution */ ...
    /* Computation */ ...
    /* Result gathering */ ...
    MPI_Finalize();
}
```

MPI Matrix Multiply (initial version)

```
/* Data distribution */
    if (me == 0) {
        for (i = 1; i < p; i++) {      /* assume p divides N evenly */
            MPI_Send(&a[i*N/p][0], N*N/p, MPI_INT, i, 0,
                        MPI_COMM_WORLD);       /* stripe of a; all of b */
            MPI_Send(b, N*N, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(&a[me*N/p][0], N*N/p, MPI_INT, 0, 0,
                                    MPI_COMM_WORLD, 0);
        MPI_Recv(b, N*N, MPI_INT, 0, 0,
                                    MPI_COMM_WORLD, 0);

    }
```

MPI Matrix Multiply (initial version)

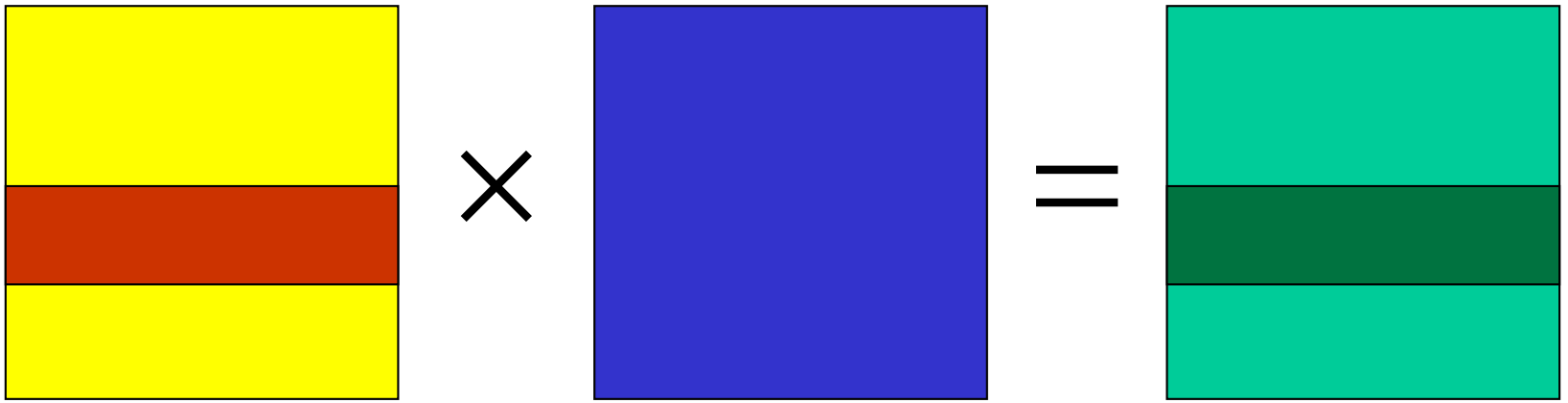
```
/* Computation */

    for (i = me*N/p; i < (me+1)*N/p; i++) {
        for (j = 0; j < N; j++) {
            c[i][j] = 0;
            for (k = 0; k < N; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
```

MPI Matrix Multiply (initial version)

```
/* Result gathering */
   if (me != 0) {
      MPI_Send(&c[me*N/p][0], N*N/p, MPI_INT, 0, 0,
                  MPI_COMM_WORLD);
   } else {
      for (i = 1; i < p; i++) {
         MPI_Recv(&c[i*N/p][0], N*N/p, MPI_INT, i, 0,
                  MPI_COMM_WORLD, 0);
      }
   }
```

Work Performed by One Process



We don’t send the unused portions of A and C.
Why reserve space for them?

MPI Matrix Multiply (with index renaming)

NB: if me != 0, a is malloc-ed smaller in this example

```
/* Data distribution */
   if (me == 0) {
      for (i = 1; i < p; i++) {
         MPI_Send(&a[i*N/p][0], N*N/p, MPI_INT, i, 0,
                  MPI_COMM_WORLD);
         MPI_Send(b, N*N, MPI_INT, i, 0, MPI_COMM_WORLD);
      }
   } else {
      MPI_Recv(a, N*N/p, MPI_INT, 0, 0, MPI_COMM_WORLD, 0);
      MPI_Recv(b, N*N, MPI_INT, 0, 0, MPI_COMM_WORLD, 0);
   }
```

MPI Matrix Multiply (with index renaming)

```
/* Computation */

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

MPI Matrix Multiply (with index renaming)

```
/* Result gathering */
    if (me != 0) {
        MPI_Send(c, N*N/p, MPI_INT, 0, 0,
                        MPI_COMM_WORLD);
    } else {
        for (i = 1; i < p; i++) {
            MPI_Recv(&c[i*N/p][0], N*N/p, MPI_INT, i,
                        0, MPI_COMM_WORLD, 0);
        }
    }
```

Global Operations (1 of 2)

- So far, we have only looked at point-to-point or one-to-one message passing facilities.
- Often, it is useful to have one-to-many or many-to-one message communication.
- This is what MPI's global operations do.

Global Operations (2 of 2)

- MPI_Barrier
- MPI_Bcast
- MPI_Scatter
- MPI_Gather
- MPI_Allgather
- MPI_Alltoall
- MPI_Reduce
- MPI_Allreduce

Barrier

MPI_Barrier(comm)

Global barrier synchronization, as with shared memory: all processes wait until all have arrived.

Broadcast

MPI_Bcast(inbuf, incnt, intype, root, comm)

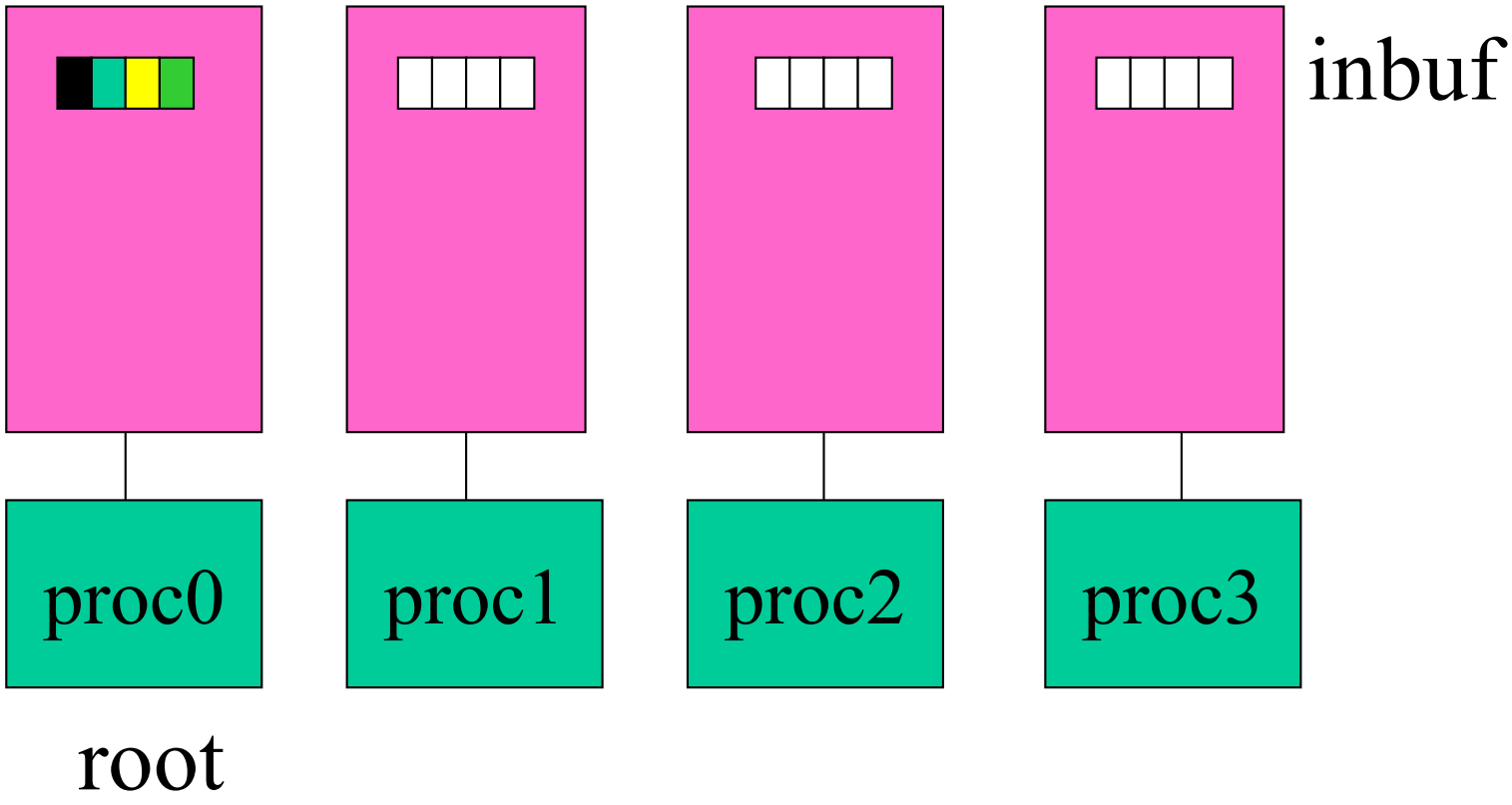
inbuf: address of input buffer (on root);
address of output buffer (elsewhere)

incnt: number of elements

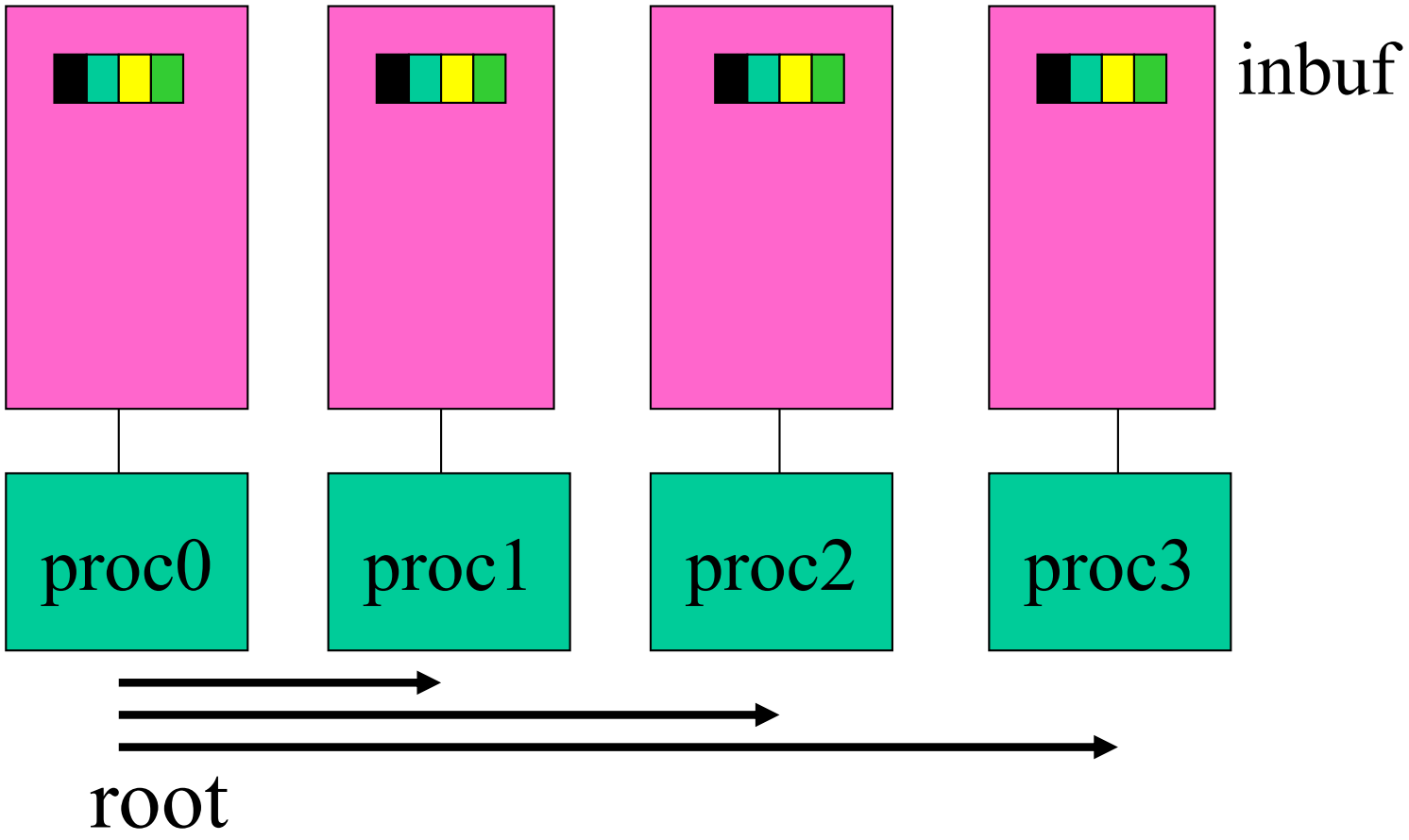
intype: type of elements

root: rank of root (sender) process

Before Broadcast



After Broadcast



Scatter

MPI_Scatter(inbuf, incnt, intype, outbuf,
outcnt, outtype, root, comm)

inbuf: address of input buffer (on root)

incnt: number of elements sent to each process

intype: type of input elements

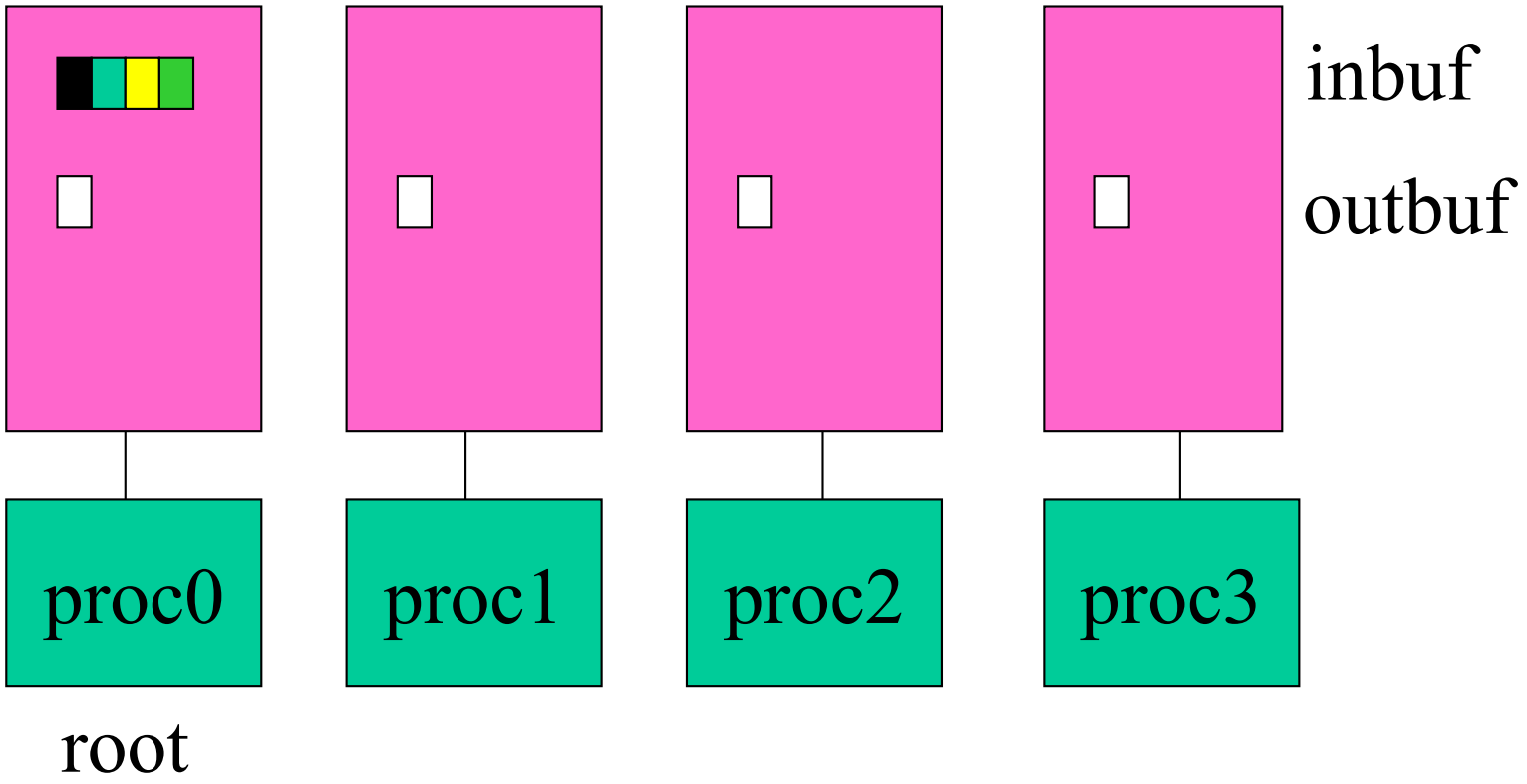
outbuf: address of output buffer (on non-root)

outcnt: num. elements received by each process

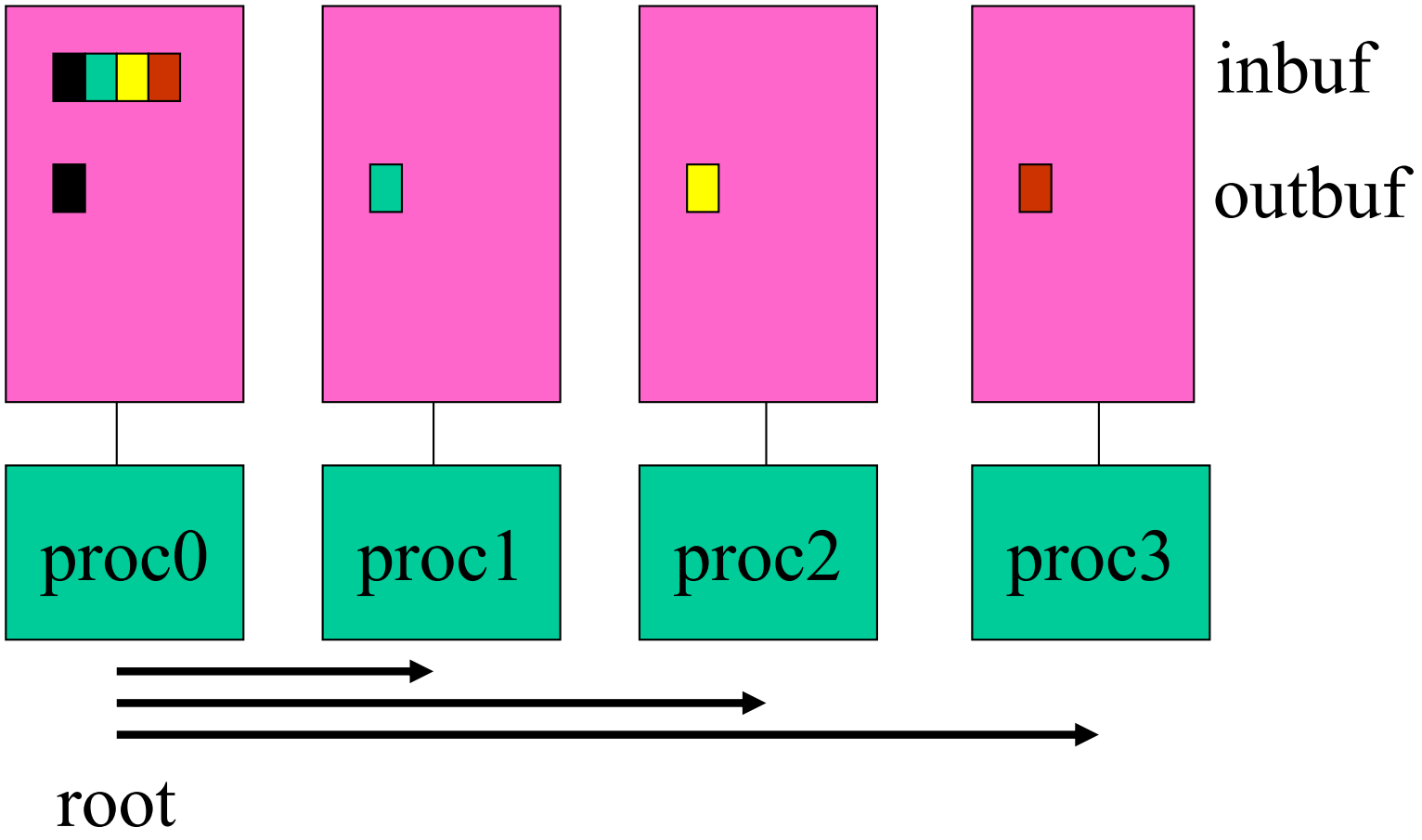
outtype: type of output elements (usually = intype)

root: rank of root process

Before Scatter



After Scatter



Gather

MPI_Gather(inbuf, incnt, intype, outbuf,
outcnt, outtype, root, comm)

inbuf: address of input buffer

incnt: number of input elements

intype: type of input elements

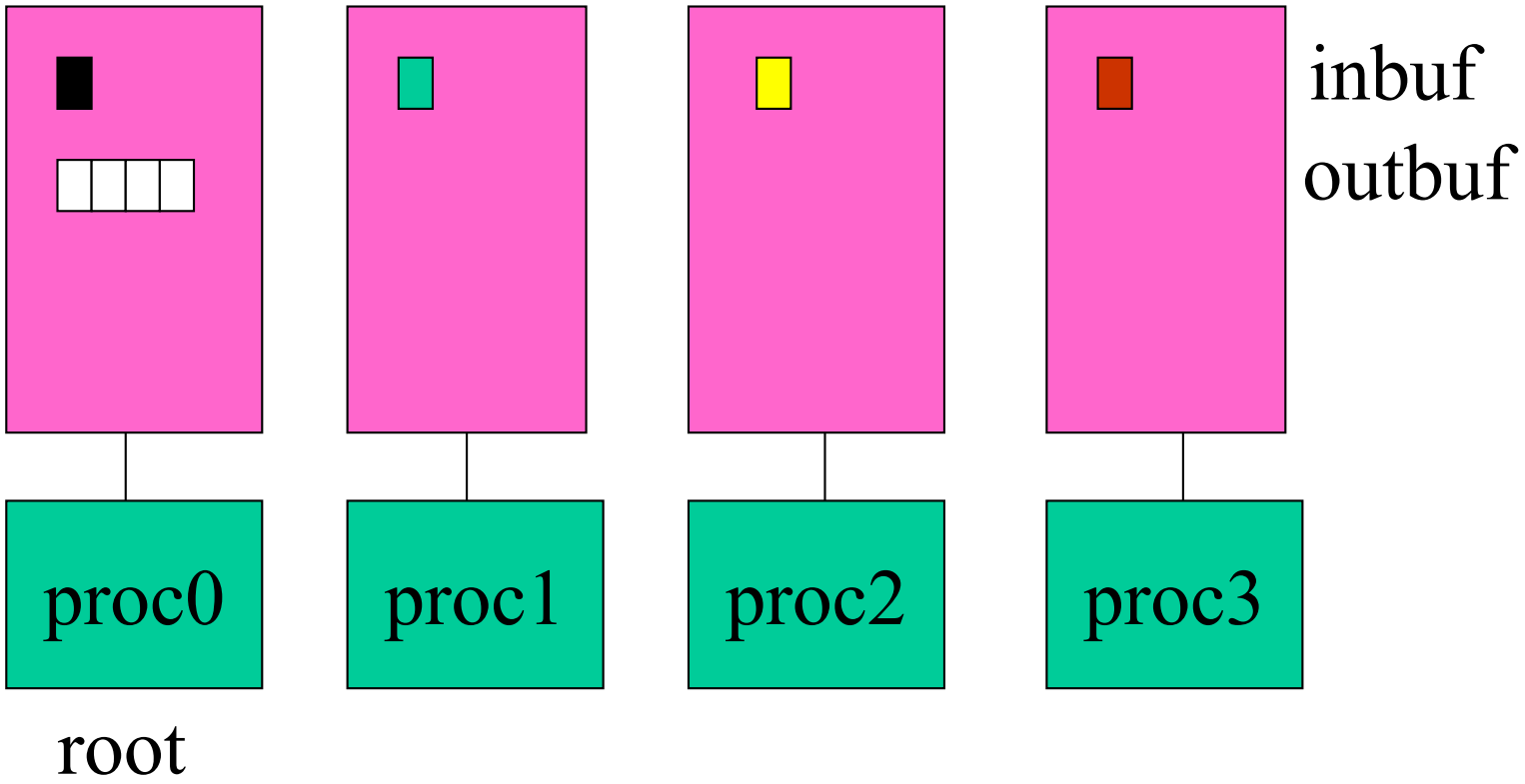
outbuf: address of output buffer

outcnt: number of output elements

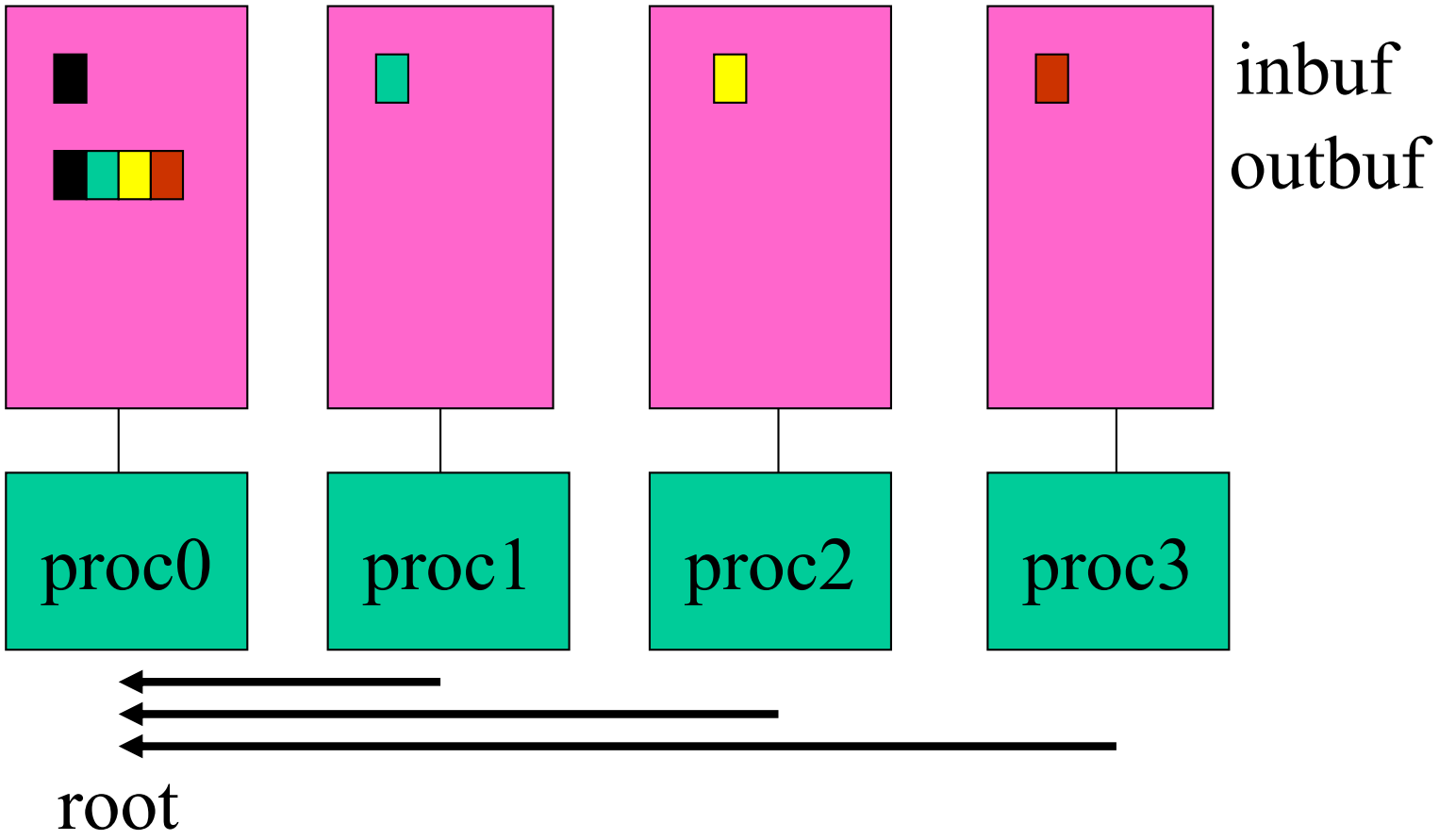
outtype: type of output elements

root: rank of root process

Before Gather



After Gather



MPI Matrix Multiply with index renaming and scatter/gather

```
main(int argc, char *argv[])
{
    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Scatter(a, N*N/p, MPI_INT, a, N*N/p,
                    MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(b, N*N, MPI_INT, 0,
                    MPI_COMM_WORLD);
    . . .
```

MPI Matrix Multiply with index renaming and scatter/gather

```
    for (i = 0; i < N/p; i++)
        for (j = 0; j < N; j++) {
            c[i][j] = 0;
            for (k = 0; k < N; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
    MPI_Gather(c, N*N/p, MPI_INT, c, N*N/p,
                        MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Finalize();
}
```

Additional Features

- Nonblocking communication
- “One-sided” communication
  - Essentially remote reads and writes
- User-defined datatypes
- Nontrivial communicators
- Parallel I/O
- Fault tolerance (ULFM)