OpenMP and MPI: Summary

High Performance Computing for Science and Engineering I

December 12, 2014

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Sources

- OpenMP specifications at www.openmp.org
  - OpenMP 3.1 (2011): C/C++, Fortran and Examples
- OpenMP tutorial:
  - https://computing.llnl.gov/tutorials/openMP/
- MPI tutorial:
  - https://computing.llnl.gov/tutorials/mpi/

OpenMP

- Compilation
- OpenMP function calls
- OpenMP environment variables
- OpenMP directives (pragmas)

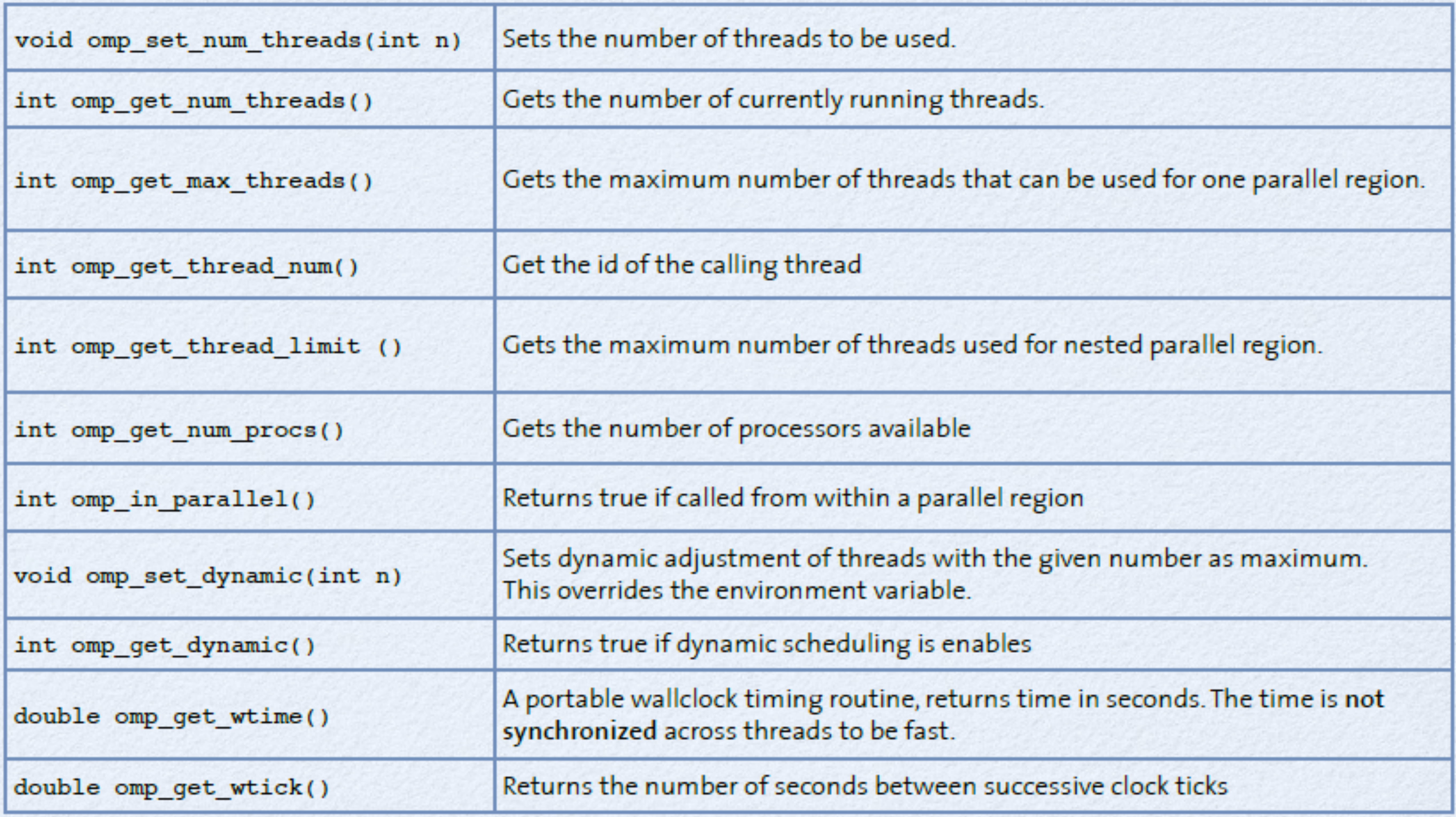
Compilation

- Compile and link using the **-fopenmp** option on the GNU compiler:
    - g++ -fopenmp openmp1.cpp
- You will need to adapt a Makefile in order to compile your code
- Useful flag: **-Wall** (enables all the warnings)
    - g++ -Wall -fopenmp openmp1.cpp
- We might need to add C++11 support
    - g++ -Wall --std=c++11 -fopenmp openmp1.cpp

OpenMP function calls

- Do not forget to include <omp.h>
    - All functions defined in this header file
- List of functions
    - int omp_get_thread_num()
    - int omp_get_num_threads()
    - void omp_set_dynamic()
    - int omp_get_dynamic()
    - void omp_set_nested()
    - int omp_get_nested()
    - double omp_get_wtime()

OpenMP function calls



`void omp_set_num_threads(int n)`	Sets the number of threads to be used.
`int omp_get_num_threads()`	Gets the number of currently running threads.
`int omp_get_max_threads()`	Gets the maximum number of threads that can be used for one parallel region.
`int omp_get_thread_num()`	Get the id of the calling thread
`int omp_get_thread_limit ()`	Gets the maximum number of threads used for nested parallel region.
`int omp_get_num_procs()`	Gets the number of processors available
`int omp_in_parallel()`	Returns true if called from within a parallel region
`void omp_set_dynamic(int n)`	Sets dynamic adjustment of threads with the given number as maximum. This overrides the environment variable.
`int omp_get_dynamic()`	Returns true if dynamic scheduling is enables
`double omp_get_wtime()`	A portable wallclock timing routine, returns time in seconds. The time is **not synchronized** across threads to be fast.
`double omp_get_wtick()`	Returns the number of seconds between successive clock ticks

OpenMP environment variables

- OpenMP provides the following environment variables for controlling the execution of parallel code
  - OMP_NUM_THREADS: max threads to use during execution
  - OMP_PROC_BIND: thread binding to cores
  - OMP_DYNAMIC: dynamic adjustment of number of threads
  - OMP_NESTED: support of nested parallelism
- Examples
  - export OMP_NUM_THREADS=4
  - export OMP_DYNAMIC=FALSE

OpenMP directives

- Parallel regions
  - #pragma omp parallel
- Synchonization
  - #pragma omp master
  - #pragma omp single
  - #pragma omp critical
  - #pragma omp barrier
- Work-sharing
  - #pragma omp section
  - #pragma omp for

omp parallel

```
#include <omp.h>

main ()  {

int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and terminate */

}
```

omp parallel

```
int A, B, C;
A = B = C = 1;
#pragma omp parallel private(B) firstprivate(C)
{
   // code
}
```

Inside the parallel region
- A is shared between threads and equal to 1
- B, C are private to each thread
- B is not initialized
- C is equal to 1

After the parallel region
- The values of B are C cannot be determined

omp master

```
#include <omp.h>

main ()  {

int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  #pragma omp master
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and terminate */

}
```

omp single

```
#include <omp.h>

main ()  {

int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only one thread does this */
  #pragma omp single
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and terminate */

}
```

omp critical

```
#include <omp.h>

main()
{

int x;
x = 0;

#pragma omp parallel shared(x)
  {

  #pragma omp critical
  x = x + 1;

  }  /* end of parallel section */

}
```

omp critical (2)

```
#include <omp.h>

int x; // what is the difference if I put it here?

main()
{
//int x;
x = 0;

#pragma omp parallel shared(x)
  {

  #pragma omp critical
  x = x + 1;

  }  /* end of parallel section */

}
```

OpenMP barrier

- OpenMP 3.1 specs, Section 2.8.3, pages 70-71:
- Summary
  - The barrier construct specifies an explicit barrier at the point at which the construct.
  - The barrier directive may not be used in place of the statement following an if, while, do, switch, or label.
- Description …
- Restrictions …

omp barrier + single vs master

```
#pragma omp parallel
{
   do_many_things();
   #pragma omp single
   {
    exchange_boundaries();
   }
   do_many_other_things();
}
```

only one thread enters

implicit barrier

```
#pragma omp parallel
{
   do_many_things();
   #pragma omp master
   {
    exchange_boundaries();
   }
   #pragma barrier
   do_many_other_things();
}
```

only thread 0 enters

omp for

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000

main ()
{

int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;

chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
  {

  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  }  /* end of parallel section */

}
```

omp sections

```
include <omp.h>
#define N     1000

main ()
{
int i;
float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i < N; i++) { a[i] = i * 1.5; b[i] = i + 22.35; }

#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp sections nowait
    {

    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    }  /* end of sections */

  }  /* end of parallel section */
}
```

omp sections (1/2)

```
//sequential code
V = alpha();
W = beta();
X = gamma(V, W);
Y = delta();
printf("%f\n", epsilon(X,Y));


#pragma omp parallel sections
{
   #pragma omp section
    V = alpha();

   #pragma omp section
   W = beta();

   #pragma omp section
   Y = delta();
}
X = gamma(V, W);
printf("%f\n", epsilon(X,Y));
```

let's assume 2 threads

omp sections (2/2)

```
#pragma omp parallel
{

   #pragma omp sections
   {
     #pragma omp section
     V = alpha();

     #pragma omp section
     W = beta();
   }

   #pragma omp sections
   {
     #pragma omp section
     X = gamma(V, W);

     #pragma omp section
     Y = delta();
   }
}
printf(“%f\n”, epsilon(X,Y));
```

omp sections

```
void XAXIS();
void YAXIS();
void ZAXIS();

void a9()
{
   #pragma omp parallel sections
   {
   #pragma omp section
      XAXIS();
   #pragma omp section
      YAXIS();
   #pragma omp section
      ZAXIS();
   }
}
```

- Quiz: What if sections is not available?

omp reduction

```
#include <omp.h>

main ()  {
int n, chunk;
float a[100], b[100], result;

/* Some initializations */
n = 100;
chunk = 10;
result = 0.0;
for (int i=0; i < n; i++)
  {
  a[i] = i * 1.0;
  b[i] = i * 2.0;
  }

  #pragma omp parallel for schedule(static,chunk) reduction(+:result)
  for (int i=0; i < n; i++)
    result = result + (a[i] * b[i]);

printf("Final result= %f\n",result);

}
```

Be careful with the reductions

- avoid false sharing
- minimize synchronization

MPI

- Compilation and Execution
    - mpic++ and mpiexec
- MPI function calls
    - Initialization
    - Message passing
        - blocking and nonblocking
    - Collective communications

Compilation and Execution

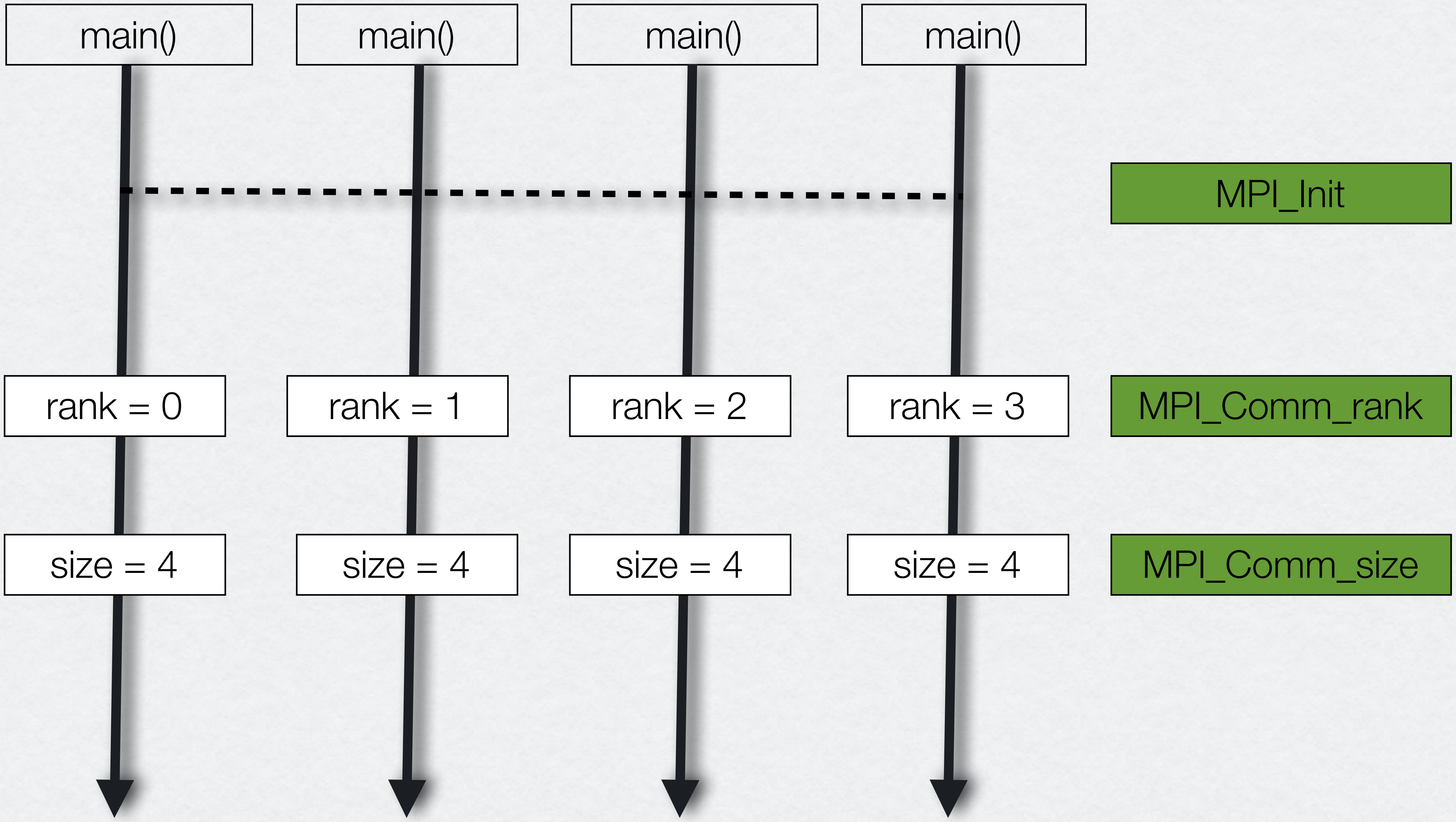
- First, you need to load the MPI environment
  - module load mpi/mpich-x86_64
- Compile
  - mpic++ -Wall --std=c++11 mpi1.cpp
- Execute with 4 processes
  - mpiexec -n 4 ./a.out
- Useful option: -prepend-rank (equivalently -l)
  - mpiexec -prepend-rank -n 4 ./a.out

MPI runtime environment

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

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv); // initialize the environment

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Hello from process %d of %d\n", rank, size);

  MPI_Finalize(); // cleanup
  return 0;
}
```

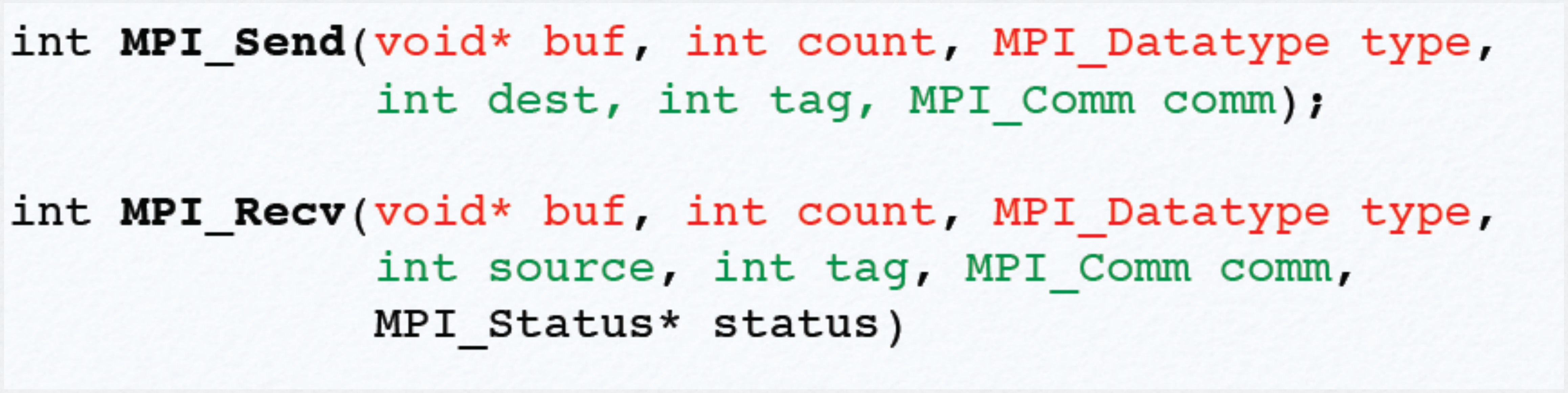
SPMD execution model

- The mpirun / mpiexec utility (spawner) starts the executable on the target cores



Point-to-Point Communication

- Messages are sent and received through MPI_Send and MPI_Recv calls



```
int MPI_Send(void* buf, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm);

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

- An MPI_Recv matches a message sent by MPI_Send if tag, source and dest match
  - MPI_ANY_TAG, MPI_ANY_SOURCE can be used for MPI_Recv

Blocking communication

```
#include <mpi.h>
int main(int argc, char *argv[])
{
  int rank;
  MPI_Status status;

  MPI_Init(&argc, &argv); // initialize the environment
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    int x = 33;
    MPI_Send(&x, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);
  }
  if (rank == 1) {
    int y;
    MPI_Recv(&y, 1, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);
  }
  MPI_Finalize();
  return 0;
}
```

Non-blocking communication

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

main(int argc, char *argv[])  {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
 MPI_Request reqs[4];
 MPI_Status stats[2];

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

 prev = rank-1;
 next = rank+1;
 if (rank == 0)  prev = numtasks - 1;
 if (rank == (numtasks - 1))  next = 0;

 MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
 MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

 MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
 MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

 { /* do some work */ }

 MPI_Waitall(4, reqs, stats);

 MPI_Finalize();
}
```

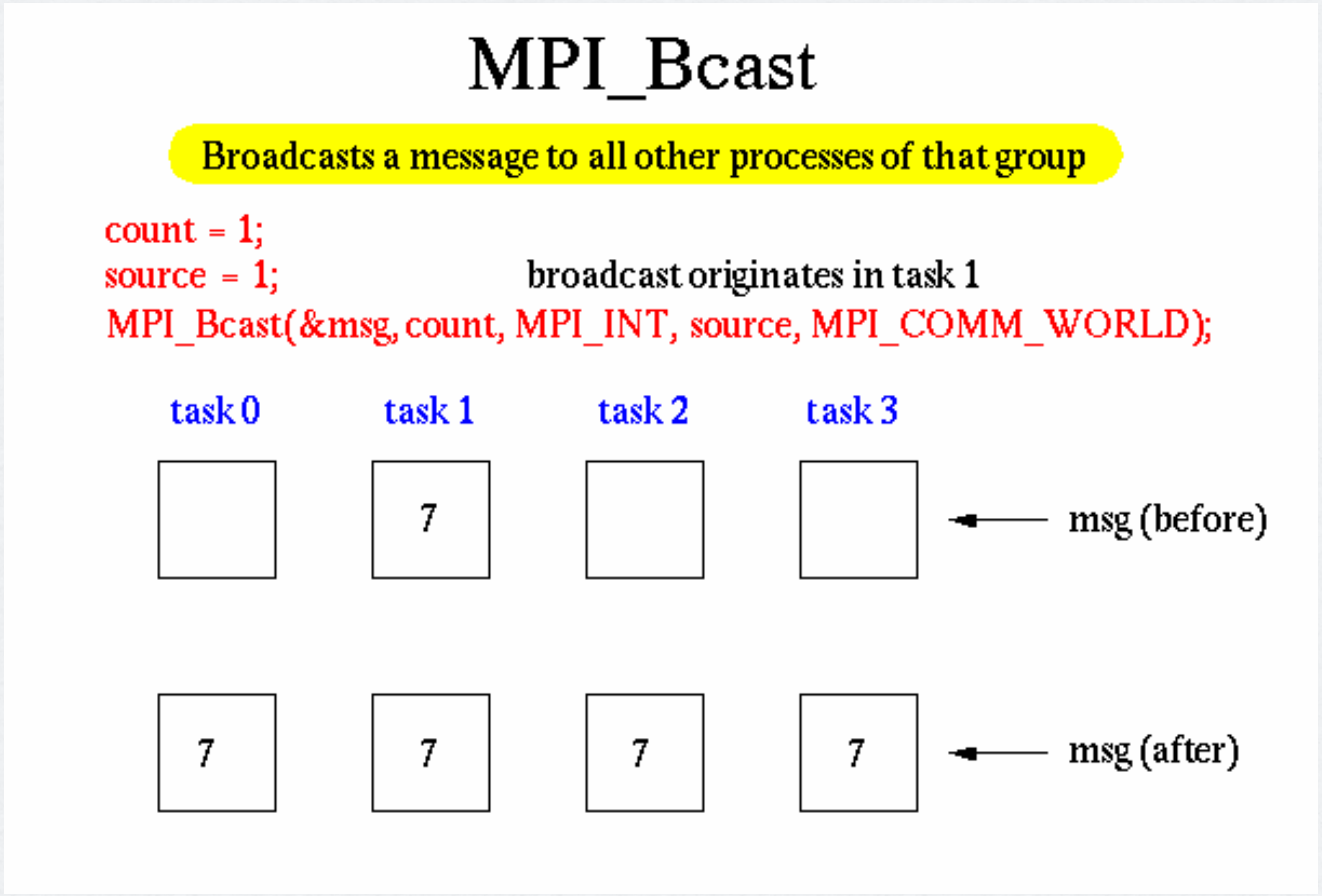
MPI_Barrier

- Synchronization operation. Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call. Then all tasks are free to proceed.

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

int main(int argc, char *argv[])
{
   MPI_Init(&argc, &argv); // initialize the environment

   int rank, size;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Barrier(MPI_COMM_WORLD);
   printf("Hello from process %d of %d\n", rank, size);

   MPI_Finalize(); // cleanup
   return 0;
}
```

MPI_Bcast

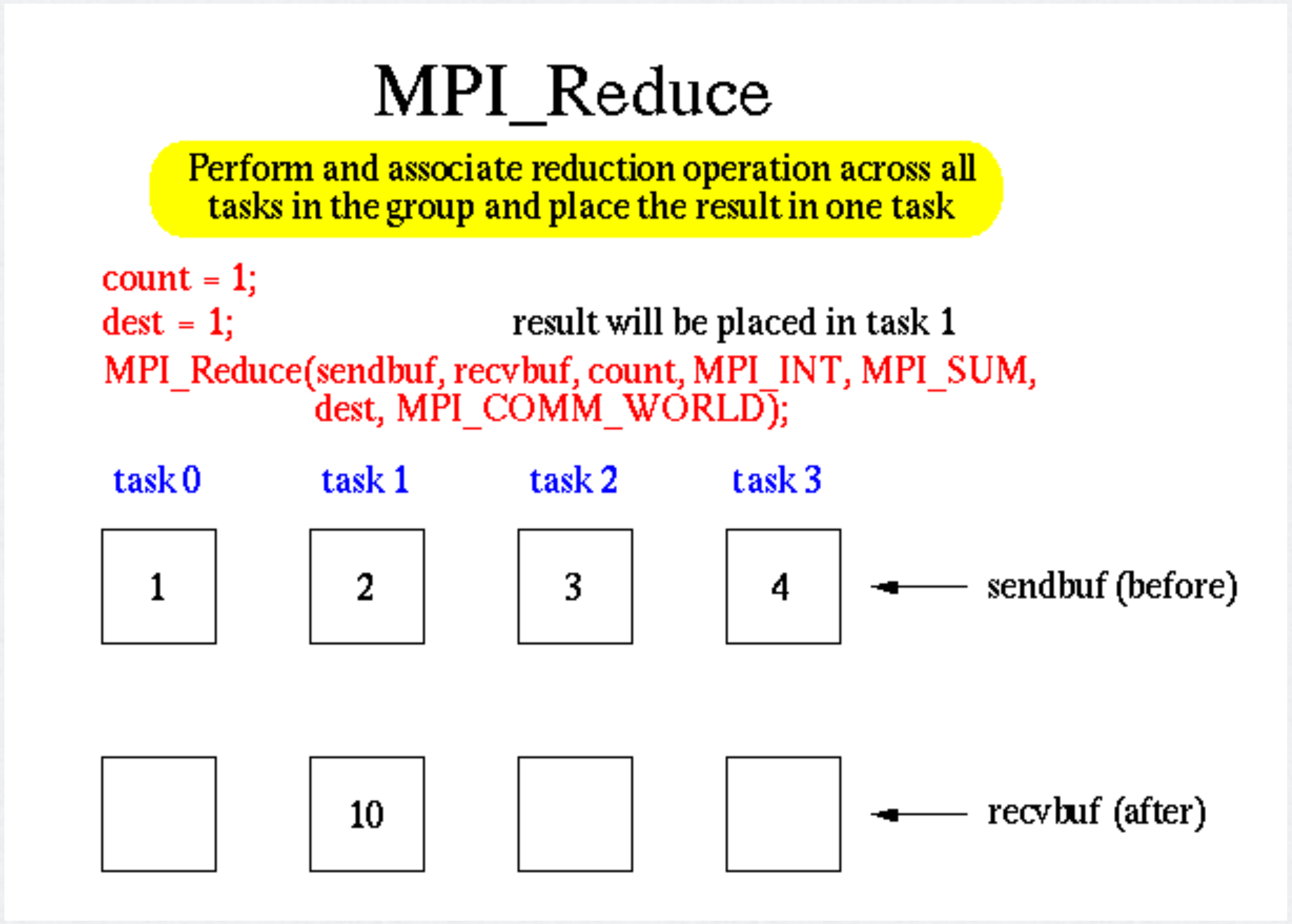
- Data movement operation. Broadcasts (sends) a message from the process with rank "root" to all other processes in the group


MPI_Bcast
Broadcasts a message to all other processes of that group
broadcast originates in task 1
task 0
task 1
task 2
task 3
7
msg (before)
7
7
7
7
msg (after)

```
count = 1;
source = 1;
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

MPI_Reduce

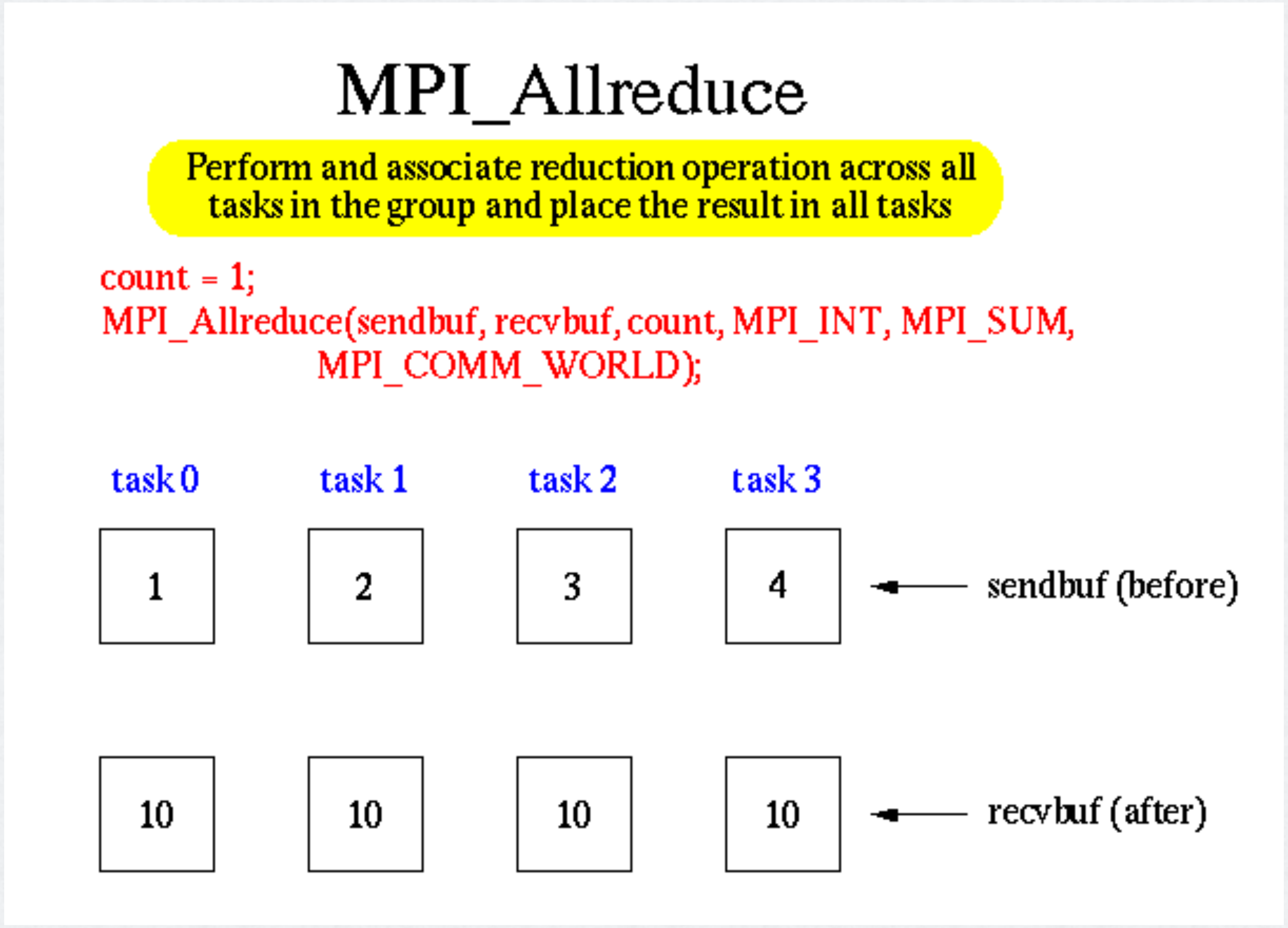
- Collective computation operation. Applies a reduction operation on all tasks in the group and places the result in one task.


MPI_Reduce
Perform and associate reduction operation across all tasks in the group and place the result in one task
result will be placed in task 1
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
10
recvbuf (after)

```
count = 1;
dest = 1;
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
           dest, MPI_COMM_WORLD);
```

MPI_Allreduce

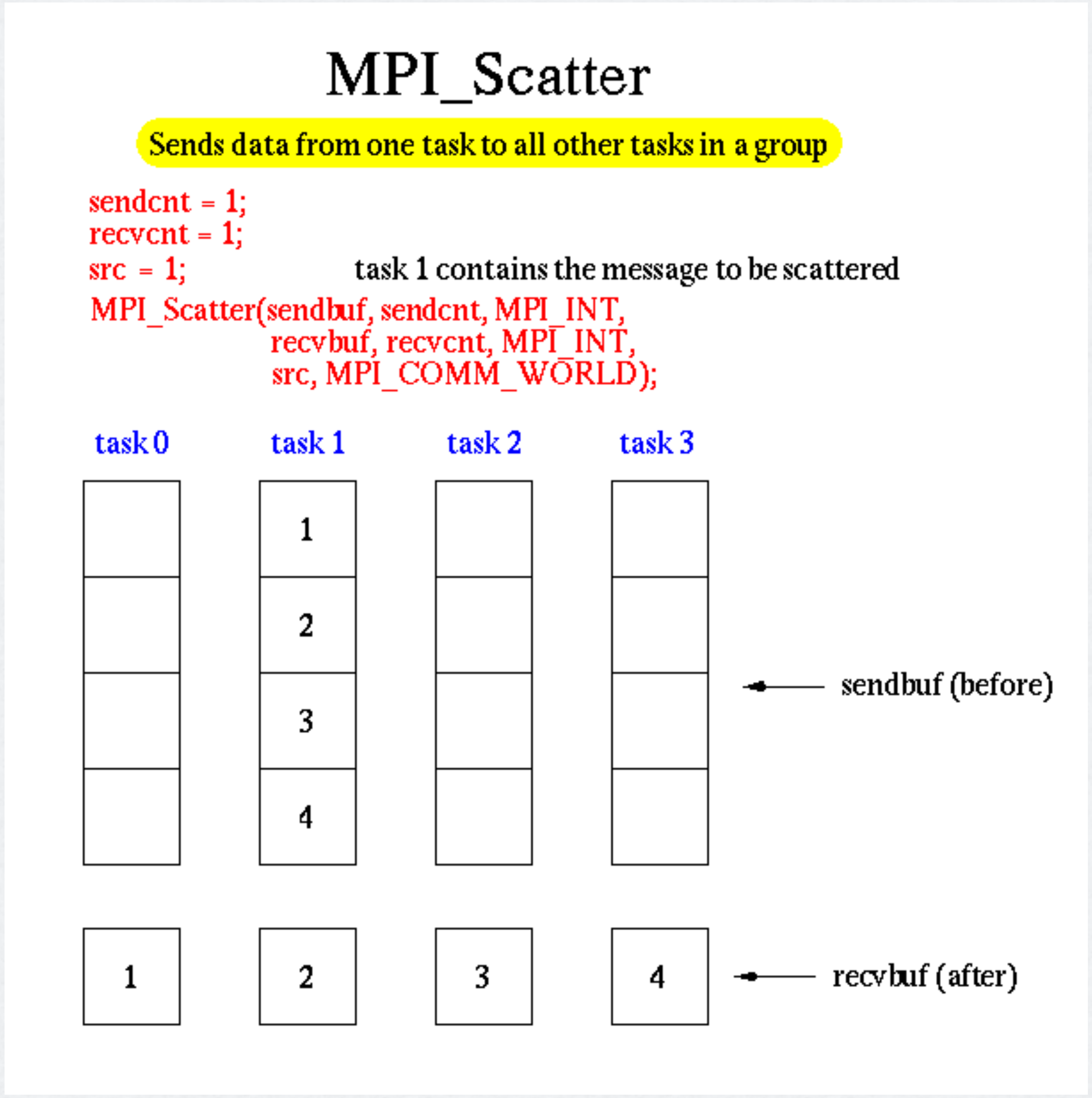
- Collective computation operation + data movement. Applies a reduction operation and places the result in all tasks in the group.
    - Equivalent to MPI_Reduce + MPI_Bcast


MPI_Allreduce
Perform and associate reduction operation across all tasks in the group and place the result in all tasks
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
10
10
10
10
recvbuf (after)

```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
              MPI_COMM_WORLD);
```

MPI_Scatter

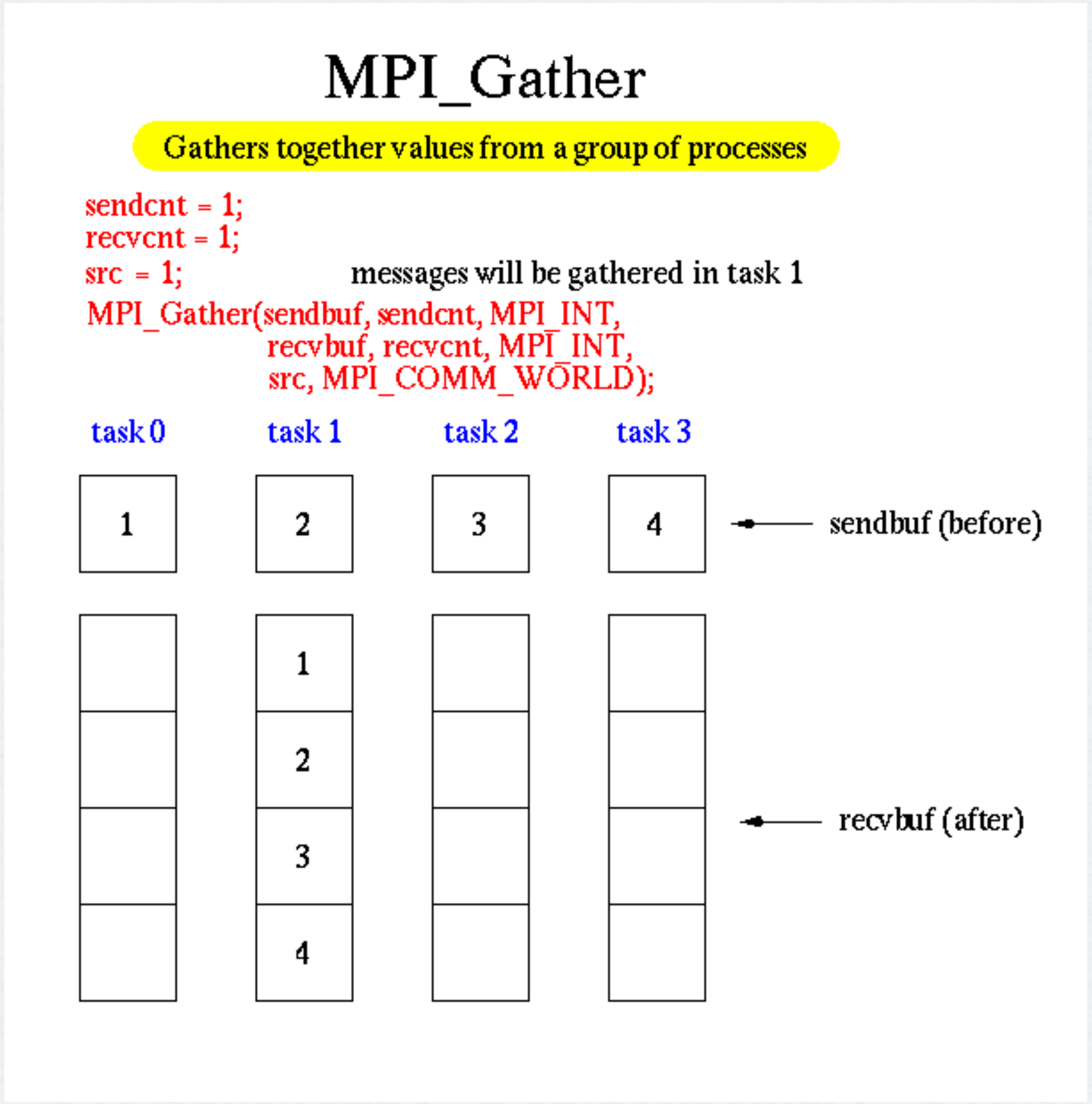
- Data movement operation. Distributes distinct messages from a single source task to each task in the group.



```
sendcnt = 1;
recvcnt = 1;
src  = 1;                task 1 contains the message to be scattered
MPI_Scatter(sendbuf, sendcnt, MPI_INT,
            recvbuf, recvcnt, MPI_INT,
            src, MPI_COMM_WORLD);
```

MPI_Gather

- Data movement operation. Gathers distinct messages from each task in the group to a single destination task (reverse of MPI_Scatter)


MPI_Gather
Gathers together values from a group of processes
messages will be gathered in task 1
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
1
2
3
4
recvbuf (after)

```
sendcnt = 1;
recvcnt = 1;
src  = 1;
MPI_Gather(sendbuf, sendcnt, MPI_INT,
           recvbuf, recvcnt, MPI_INT,
           src, MPI_COMM_WORLD);
```

Measuring time

```
#include <mpi.h>
#include <stdio.h>  // printf
#include <unistd.h> // sleep

int main( int argc, char *argv[] )
{
    double t1, t2;

    MPI_Init(&argc, &argv);
    t1 = MPI_Wtime();
    sleep(2);
    t2 = MPI_Wtime();
    printf("Elapsed time =%f seconds\n", t2-t1);
    MPI_Finalize();
    return 0;
}
```

Implementation of MPI_Bcast

```
#include <mpi.h>

int main(int argc , char **argv)
{
   int size,  rank;
   double data;

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

   srand48(rank);

   for (int k = 0; k < 10; k++) {
      if (!rank) data = drand48();

      MPI_Bcast(&data, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      printf("Step %d: I am Process %d Data = %f\n", k, rank, data);
   }
    MPI_Finalize();
}
```

Implementation of MPI_Bcast

```
#include <mpi.h>

int main(int argc , char **argv)
{
   int size,  rank;
   double data;

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

   srand48(rank);

   for (int k = 0; k < 10; k++) {
      if (!rank) data = drand48();

      if (!rank) {
         for (i = 1; i < size; i++)
            MPI_Send(&data, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
      } else {
            MPI_Recv(&data, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
      }

      printf("Step %d: I am Process %d Data = %f\n", k, rank, data);
   }
    MPI_Finalize();
}
```

Next semester: MPI + OpenMP

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

int main(int argc, char *argv[])
{
    int rank, nprocs;

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

    omp_set_num_threads(2);
 #pragma omp parallel
    {
      printf("Hello, world.  I am %d of %d  thread=%d\n", rank,
              nprocs, omp_get_thread_num());
    }

    MPI_Finalize();
    return 0;
}
```