CPSC 4770/6770

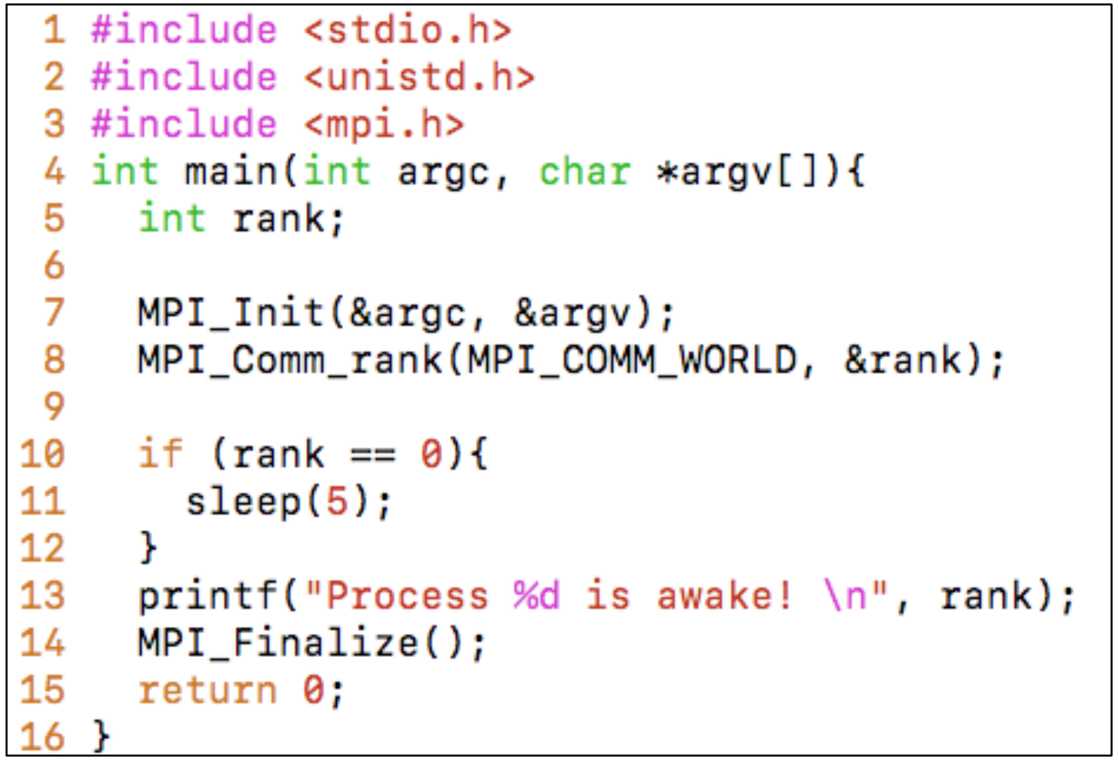
Distributed and Cluster Computing

Lecture 7: Synchronous Computing

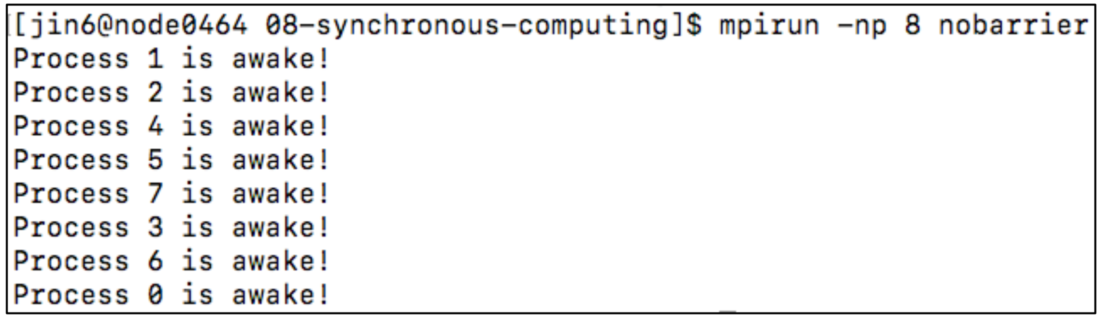
Synchronous Computation

- In a (fully) synchronous computation, all the processes synchronized at regular points, usually to exchange data or to make sure that every process has gone through the same set of procedures (to update their own data) before proceeding

nobarrier.c

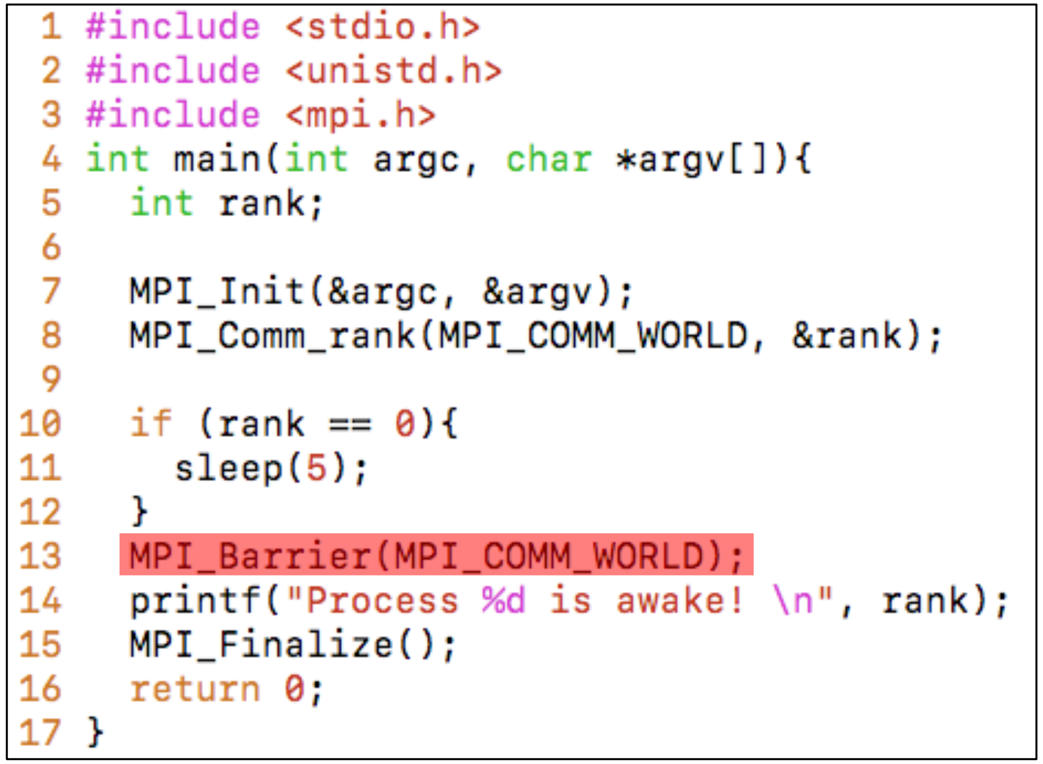


```c
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>
int main(int argc, char *argv[]){
  int rank;

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

  if (rank == 0){
    sleep(5);
  }
  printf("Process %d is awake! \n", rank);
  MPI_Finalize();
  return 0;
}
```

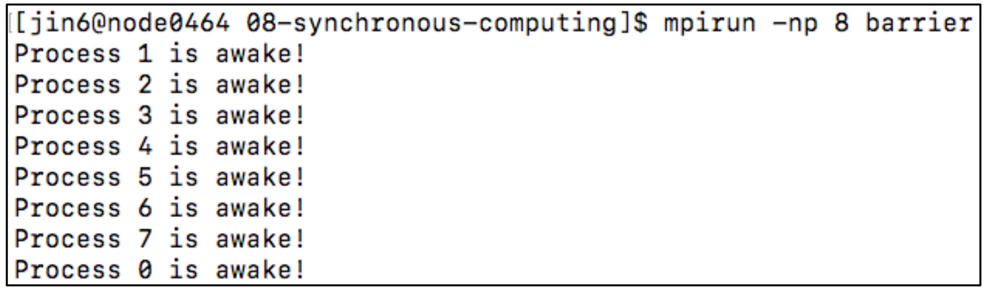


```
[jin6@node0464 08-synchronous-computing]$ mpirun -np 8 nobarrier
Process 1 is awake!
Process 2 is awake!
Process 4 is awake!
Process 5 is awake!
Process 7 is awake!
Process 3 is awake!
Process 6 is awake!
Process 0 is awake!
```

barrier.c



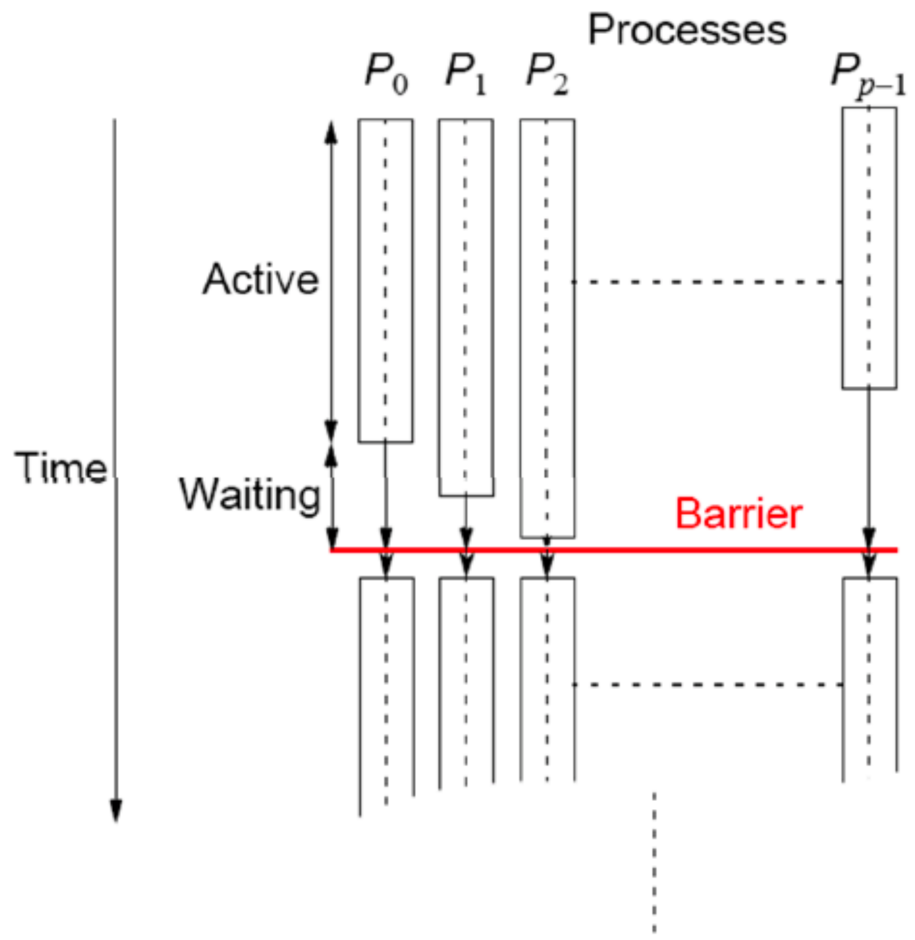
```c
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>
int main(int argc, char *argv[]){
  int rank;

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

  if (rank == 0){
    sleep(5);
  }
  MPI_Barrier(MPI_COMM_WORLD);
  printf("Process %d is awake! \n", rank);
  MPI_Finalize();
  return 0;
}
```



```
[jin6@node0464 08-synchronous-computing]$ mpirun -np 8 barrier
Process 1 is awake!
Process 2 is awake!
Process 3 is awake!
Process 4 is awake!
Process 5 is awake!
Process 6 is awake!
Process 7 is awake!
Process 0 is awake!
```

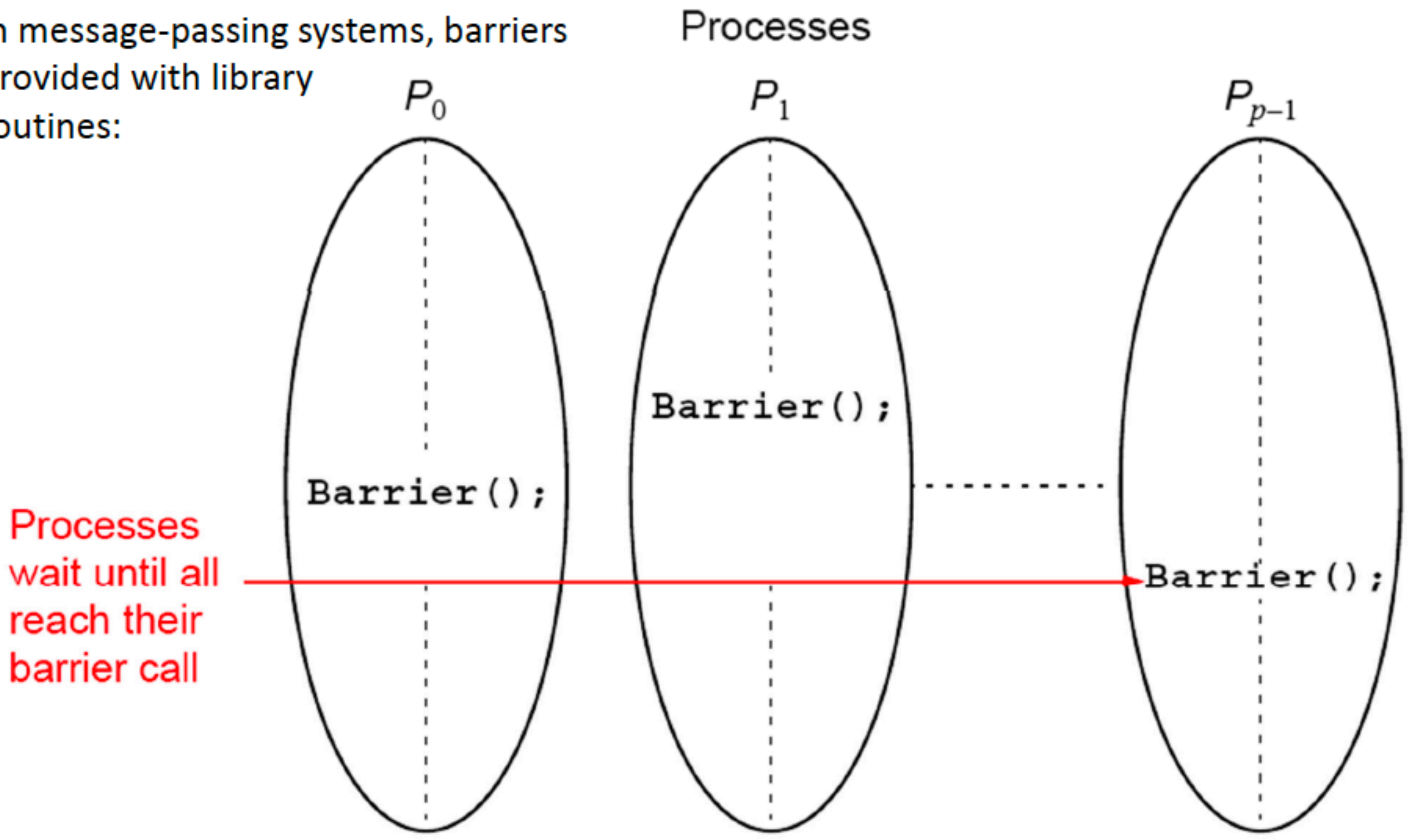
Barrier

- A basic mechanism for synchronism for synchronizing processes – inserted at the point in each process where it must wait
- All processes can continue from this point when all the processes have reached it



Barrier()

In message-passing systems, barriers provided with library routines:

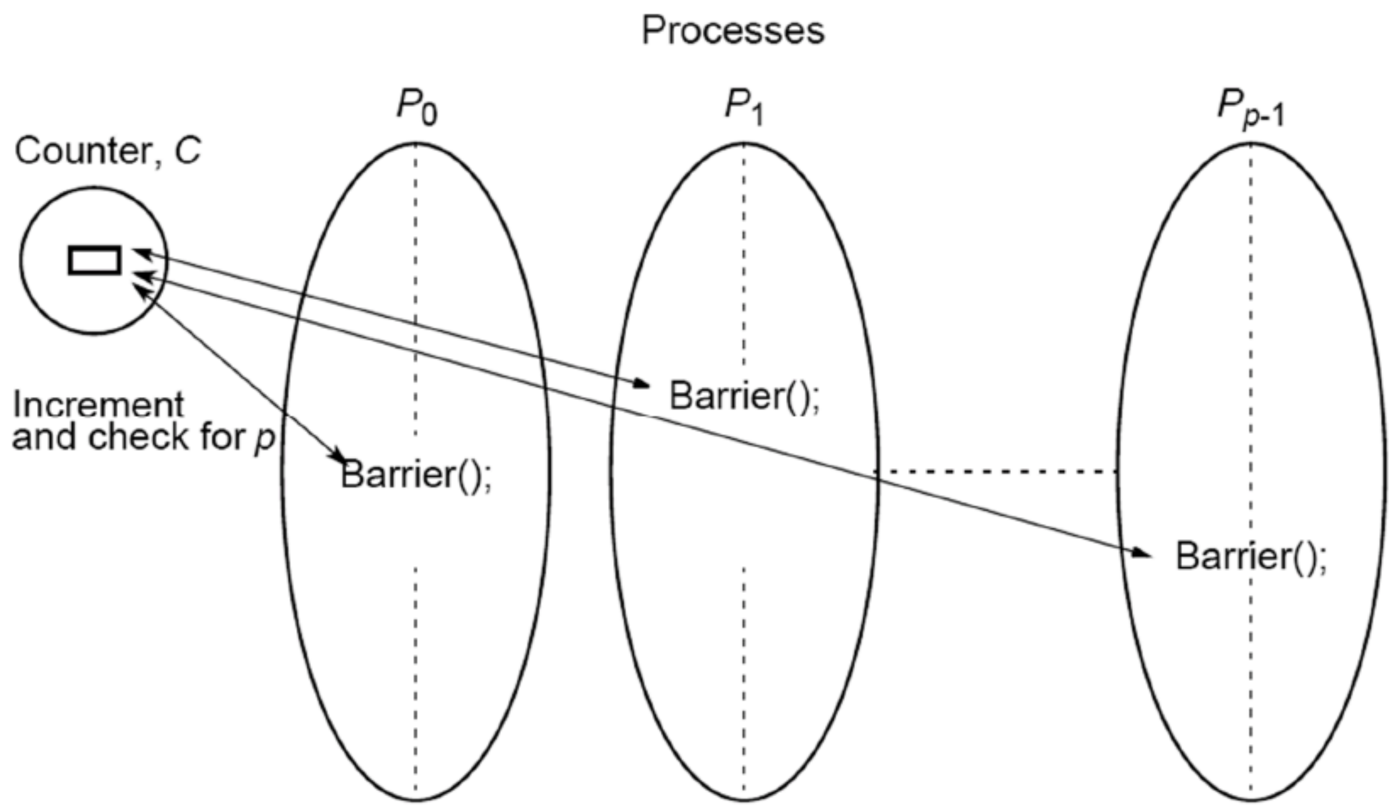


MPI_Barrier()

- Barrier with a named communicator being the only parameter
- Called by each process in the group, blocking until all members of the group have reached the barrier call and only returning then

Counter Implementation

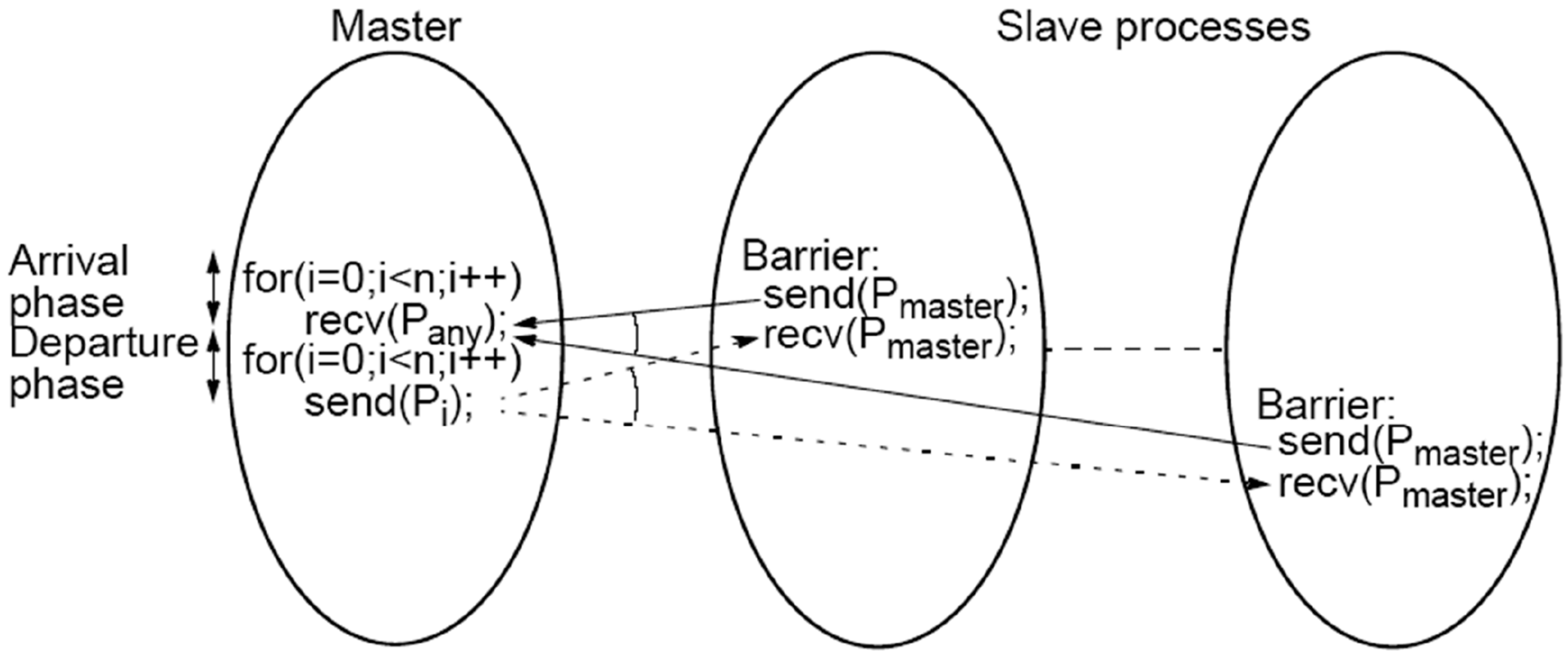
Centralized counter implementation (a linear barrier):



Counter-based barriers often have two phases:

- A process enters **arrival phase** and does not leave this phase until all processes have arrived in this phase
- Then processes move to **departure phase** and are released

Counter Implementation (Cont.)



Master:

```
for (i = 0; i < n; i++)      /*count slaves as they reach barrier*/
          recv(Pany);

for (i = 0; i < n; i++)     /* release slaves */
          send(Pi);
```

Slave processes:

```
send(Pmaster);
recv(Pmaster);
```

Tree Implementation

- A more efficient implementation in O(log p) steps

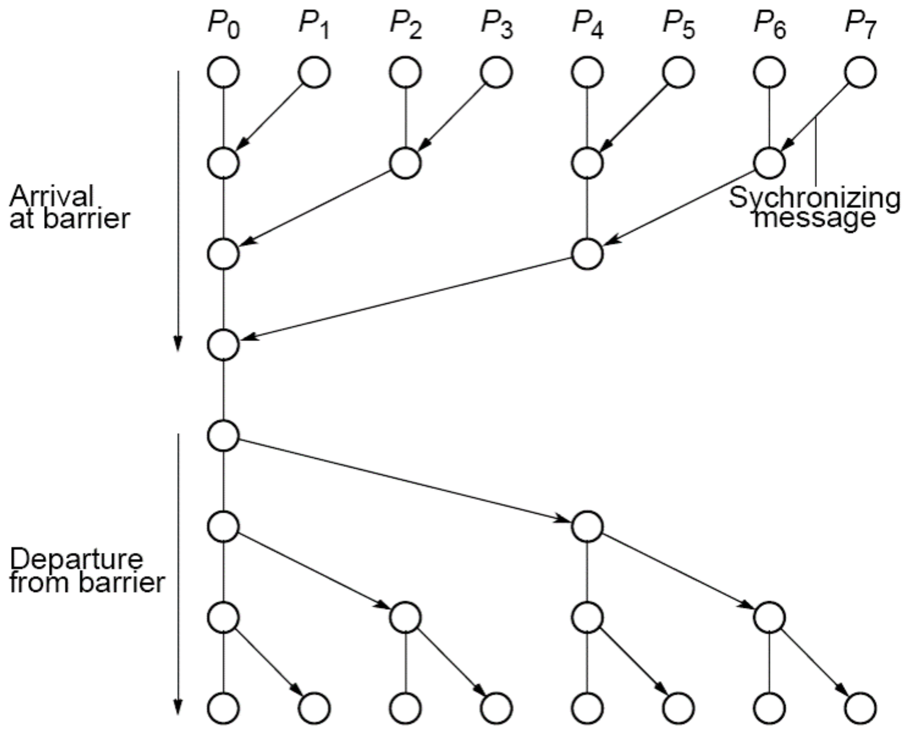
Suppose 8 processes, *P*0, *P*1, *P*2, *P*3, *P*4, *P*5, *P*6, *P*7:

1st stage: *P*1 sends message to *P*0; (when *P*1 reaches its barrier)
*P*3 sends message to *P*2; (when *P*3 reaches its barrier)
*P*5 sends message to *P*4; (when *P*5 reaches its barrier)
*P*7 sends message to *P*6; (when *P*7 reaches its barrier)

2nd stage: *P*2 sends message to *P*0; (*P*2 & *P*3 reached their barrier)
*P*6 sends message to *P*4; (*P*6 & *P*7 reached their barrier)

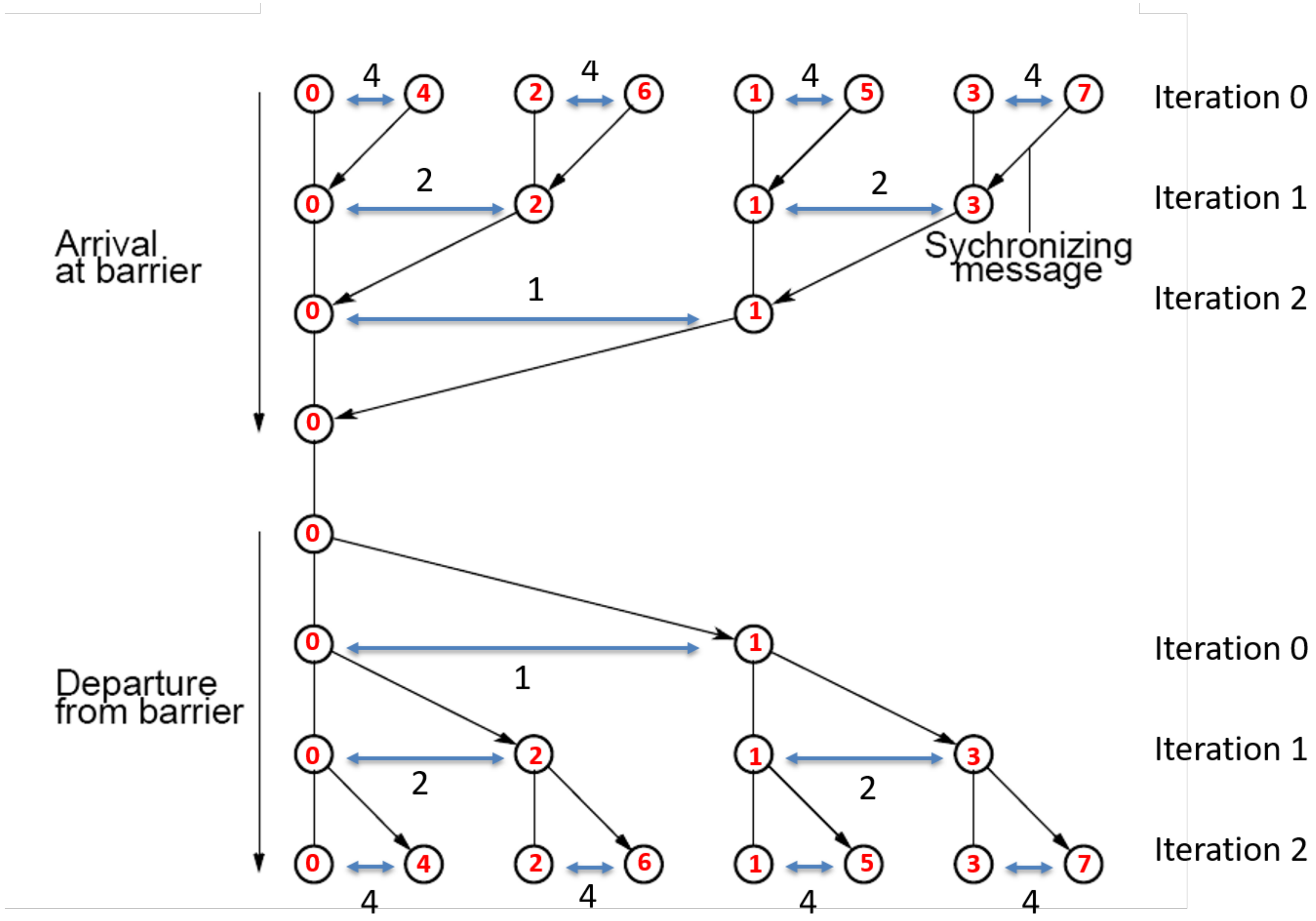
3rd stage: *P*4 sends message to *P*0; (*P*4, *P*5, *P*6, & *P*7 reached barrier)
*P*0 terminates arrival phase;
(when *P*0 reaches barrier & received message from *P*4)

Release with a reverse tree construction.



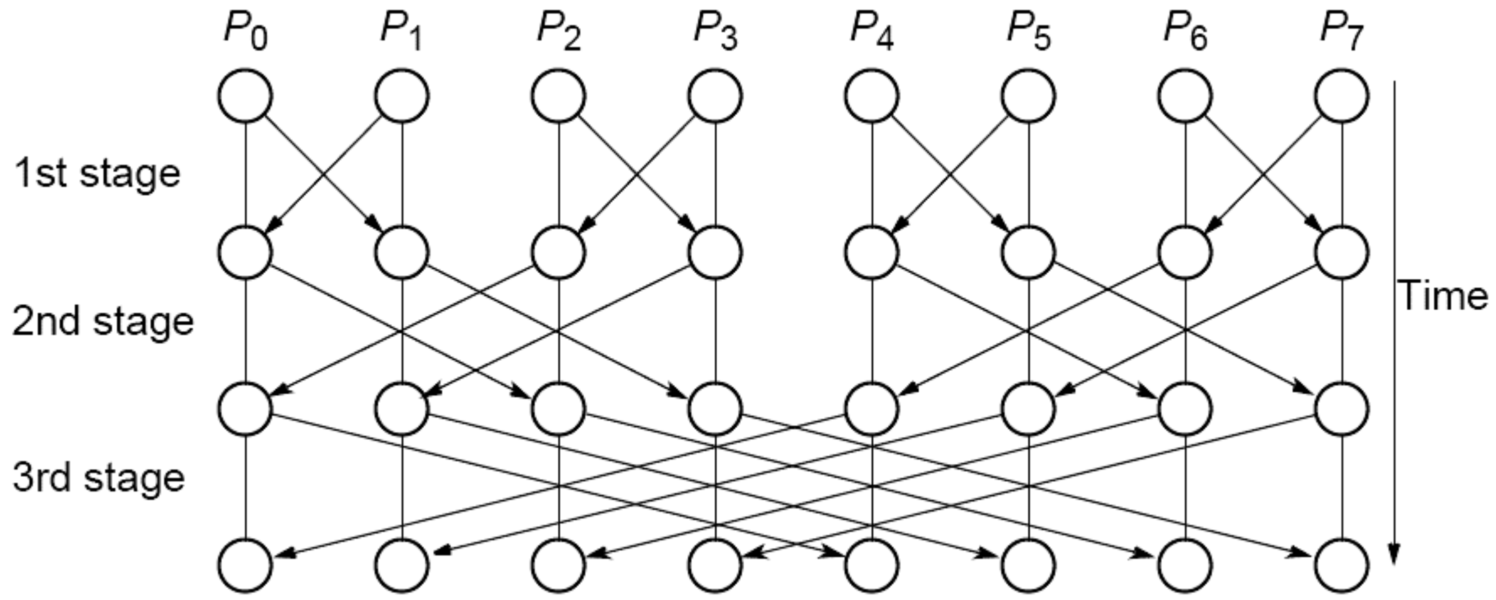
Tree barrier construction

Tree Implementation (Cont.)



Butterfly Barrier

1st stage	$P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$
2nd stage	$P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$
3rd stage	$P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$



Safety and Deadlock

- When all processes send their messages first and then receive all of their messages, it is "unsafe" because it relies upon buffering in the send()s. The amount of buffering is not specified in MPI.
- If insufficient storage available, send routine may be delayed from returning until storage becomes available or until the message can be sent without buffering.
- Then, a locally blocking send() could behave as a synchronous send(), only returning when the matching recv() is executed. Since a matching recv() would never be executed if all the send()s are synchronous, deadlock would occur.

- Alternate the order of the send()s and recv()s in adjacent processes so that only one process performs the send()s first.
- Then even synchronous send()s would not cause deadlock.

MPI Safe Message Passing Routines

MPI offers several methods for safe communication:

- Combined send and receive routines:

  `MPI_Sendrecv()`

  which is guaranteed not to deadlock

- Buffered send()s:

  `MPI_Bsend()`

  here the user provides explicit storage space

- Nonblocking routines:

  `MPI_Isend()` and MPI_Irecv()

  which return immediately.

  Separate routine used to establish whether message has been received:
  `MPI_Wait()`, `MPI_Waitall()`, `MPI_Waitany()`, `MPI_Test()`, `MPI_Testall()`, or `MPI_Testany()`.

Data Parallel Computation

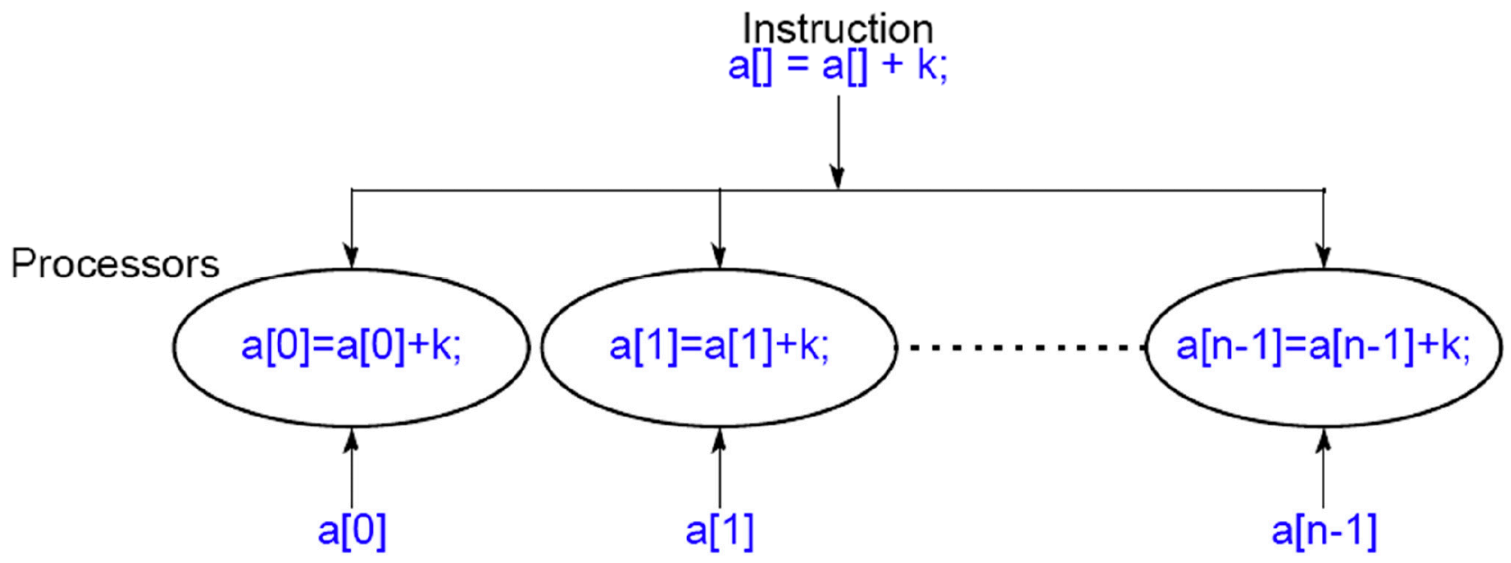
- A form of computation that implicitly has synchronization requirements
  - The same operation is performed on different data elements simultaneously
  - Ease of programming (essentially only one program)
  - Can scale easily to larger problem

A Simple Example

- To add the same constant to each element of an array:

```
for (i = 0; i < n; i++)
        a[i] = a[i] + k;
```

The statement **a[i] = a[i] + k;** could be executed simultaneously by multiple processors, each using a different index **i (0<i<=n)**.



A Simple Example (Cont.)

To add **k** to each element of an array, **a**, we can write

```
forall (i = 0; i < n; i++)
        a[i] = a[i] + k;
```

A special ”parallel” construct to specify data parallel operations

But, MPI does not have “forall” construct

To add **k** to the elements of an array:

```
i = myrank;
a[i] = a[i] + k;      /* body */
barrier(mygroup);
```

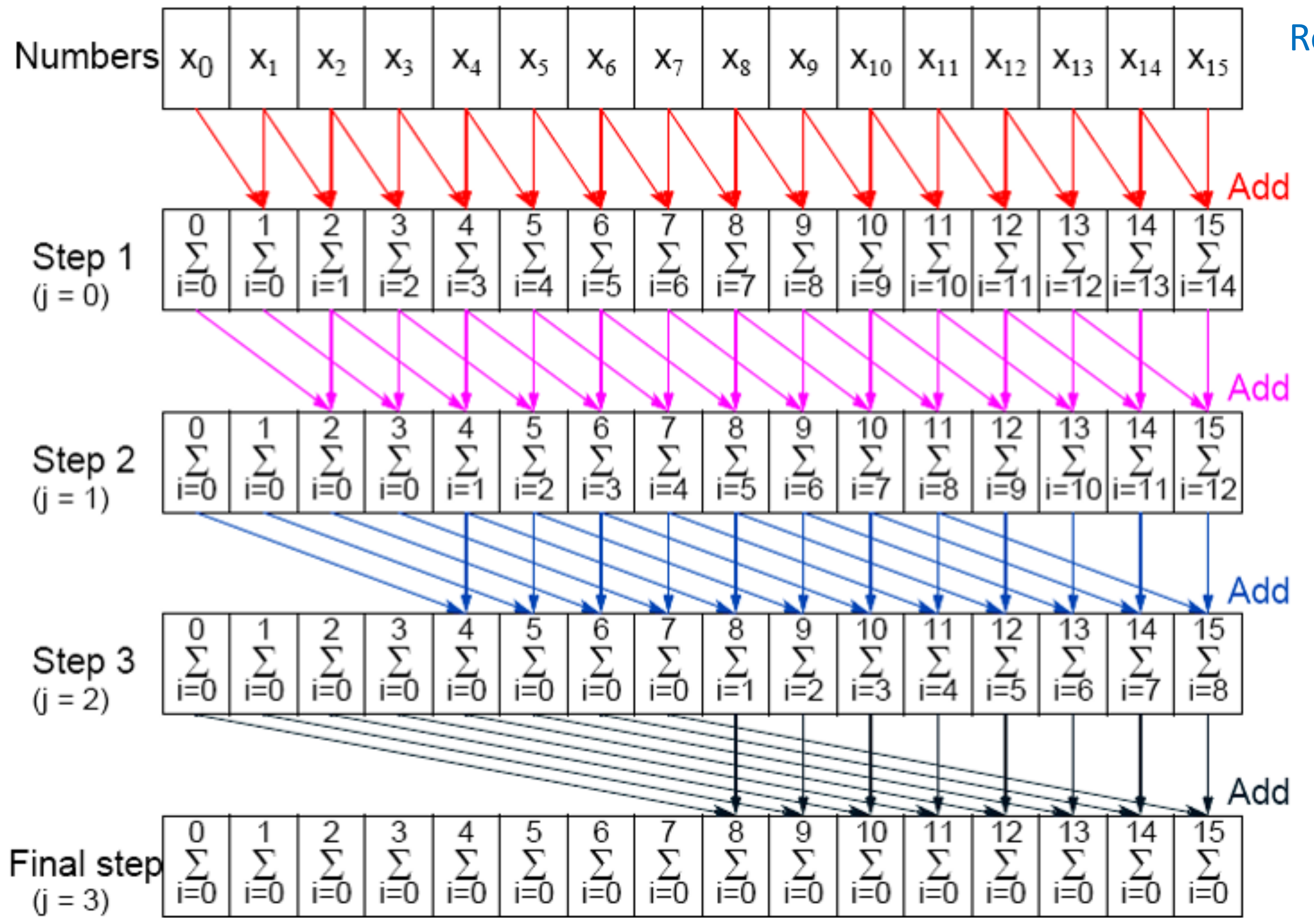
where **myrank** is a process rank between 0 and n - 1.

A Prefix Sum Problem Example

- Given a list of numbers, x0, …, xn-1, compute all the partial summations, i.e.:
    - x0 + x1;
    - x0 + x1 + x2;
    - x0 + x1 + x2 + x3;
    - x0 + x1 + x2 + x3 + x4;
    - …
- Can also be defined with associative operations other than addition
- Widely studied with practical applications in areas such as processor allocation, data compaction, sorting, and polynomial evaluation

Sequential code:
```
sum[0] = x[0];
for (i = 1; i < n; i++)
    sum[i] = sum[i-1] + x[i];
```

Data Parallel Method for Prefix Sum Operation


Numbers
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
Add
Step 1 (j = 0)
$\sum_{i=0}^{0}$ $\sum_{i=0}^{1}$ $\sum_{i=1}^{2}$ $\sum_{i=2}^{3}$ $\sum_{i=3}^{4}$ $\sum_{i=4}^{5}$ $\sum_{i=5}^{6}$ $\sum_{i=6}^{7}$ $\sum_{i=7}^{8}$ $\sum_{i=8}^{9}$ $\sum_{i=9}^{10}$ $\sum_{i=10}^{11}$ $\sum_{i=11}^{12}$ $\sum_{i=12}^{13}$ $\sum_{i=13}^{14}$ $\sum_{i=14}^{15}$
Add
Step 2 (j = 1)
$\sum_{i=0}^{0}$ $\sum_{i=0}^{1}$ $\sum_{i=0}^{2}$ $\sum_{i=0}^{3}$ $\sum_{i=1}^{4}$ $\sum_{i=2}^{5}$ $\sum_{i=3}^{6}$ $\sum_{i=4}^{7}$ $\sum_{i=5}^{8}$ $\sum_{i=6}^{9}$ $\sum_{i=7}^{10}$ $\sum_{i=8}^{11}$ $\sum_{i=9}^{12}$ $\sum_{i=10}^{13}$ $\sum_{i=11}^{14}$ $\sum_{i=12}^{15}$
Add
Step 3 (j = 2)
$\sum_{i=0}^{0}$ $\sum_{i=0}^{1}$ $\sum_{i=0}^{2}$ $\sum_{i=0}^{3}$ $\sum_{i=0}^{4}$ $\sum_{i=0}^{5}$ $\sum_{i=0}^{6}$ $\sum_{i=0}^{7}$ $\sum_{i=1}^{8}$ $\sum_{i=2}^{9}$ $\sum_{i=3}^{10}$ $\sum_{i=4}^{11}$ $\sum_{i=5}^{12}$ $\sum_{i=6}^{13}$ $\sum_{i=7}^{14}$ $\sum_{i=8}^{15}$
Add
Final step (j = 3)
$\sum_{i=0}^{0}$ $\sum_{i=0}^{1}$ $\sum_{i=0}^{2}$ $\sum_{i=0}^{3}$ $\sum_{i=0}^{4}$ $\sum_{i=0}^{5}$ $\sum_{i=0}^{6}$ $\sum_{i=0}^{7}$ $\sum_{i=0}^{8}$ $\sum_{i=0}^{9}$ $\sum_{i=0}^{10}$ $\sum_{i=0}^{11}$ $\sum_{i=0}^{12}$ $\sum_{i=0}^{13}$ $\sum_{i=0}^{14}$ $\sum_{i=0}^{15}$

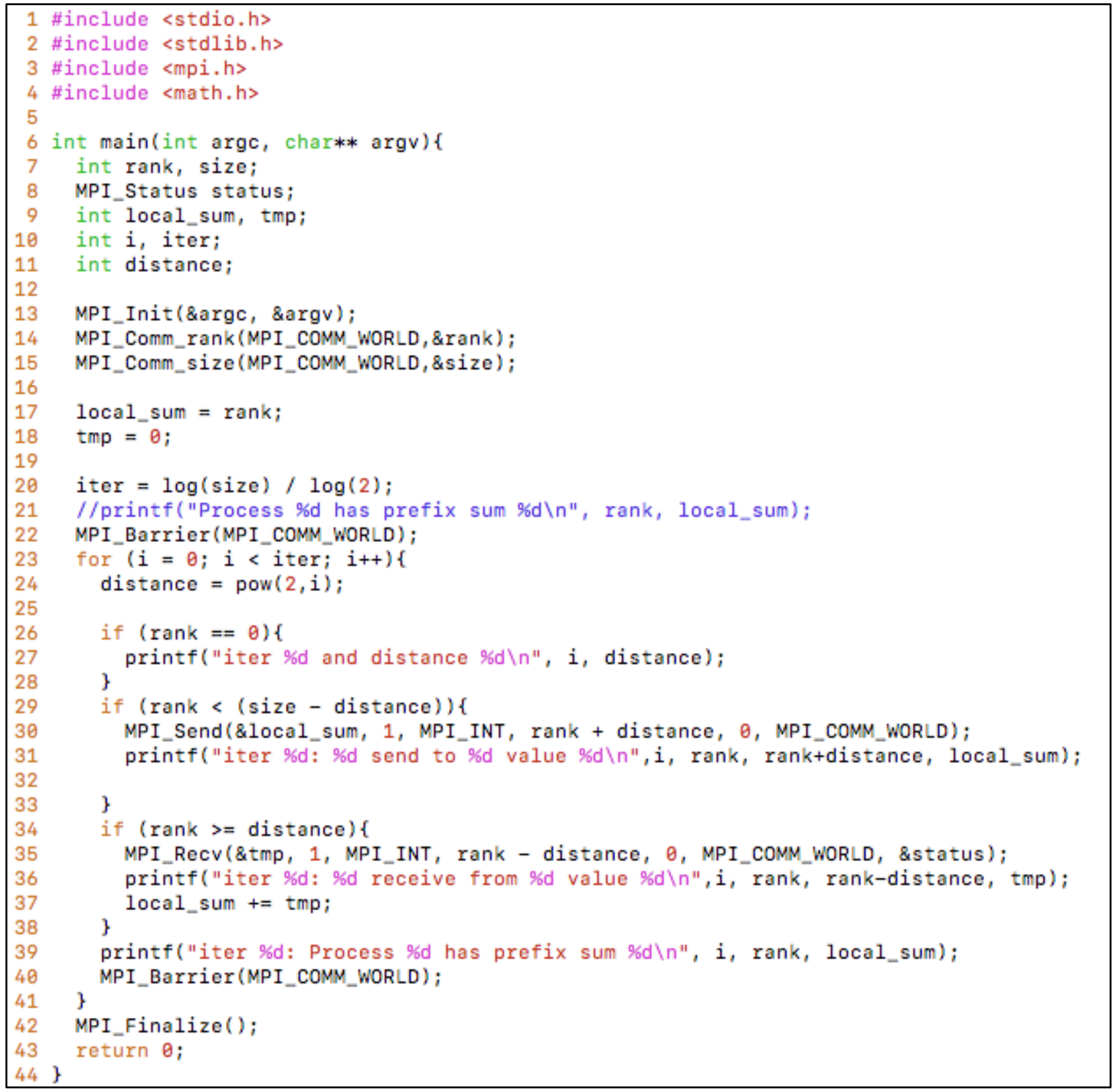
Rewrite the sequential code as:

```
for (j = 0; j < log(n); j++)     // at each step
    for (i = 2^j; i < n; i++)     // add to accumulating sum
        x[i] = x[i] + x[i - 2^j];
```

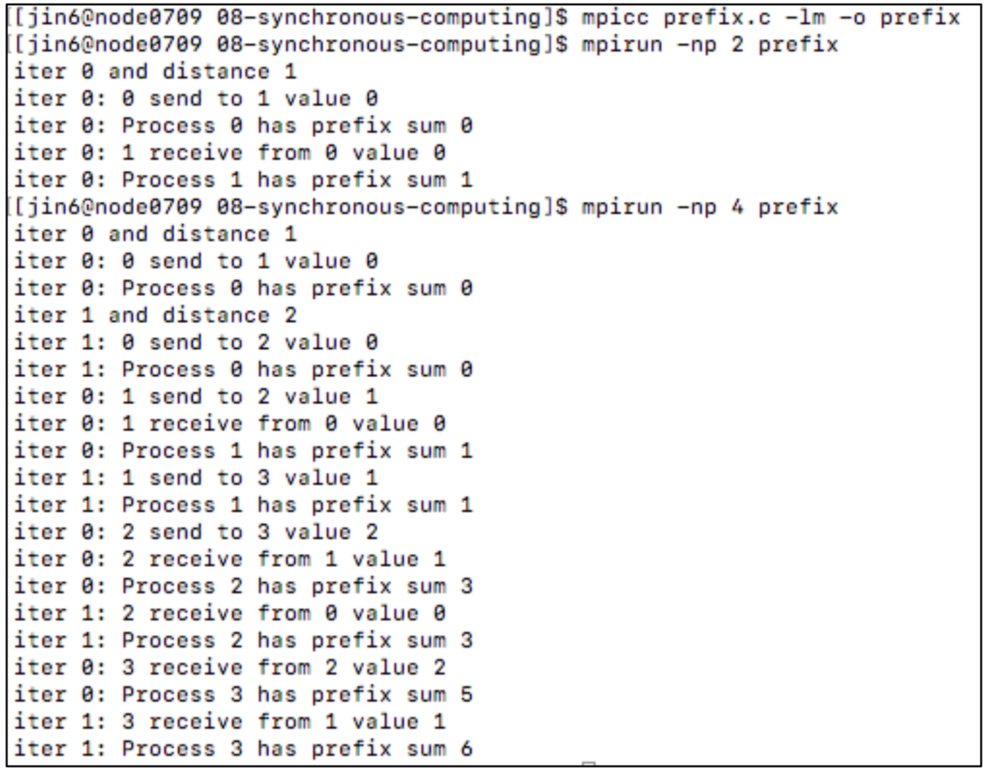
```
for (j = 0; j < log(n); j++)     // at each step
    forall (i = 0; i < n; i++)     // add to accumulating sum
        if (i >= 2^j) x[i] = x[i] + x[i - 2^j];
```

```
for (j = 0; j < log(n); j++)  {    //for each synchronous iteration
        i = myrank;     // find value of i to be used
        if (i >= 2^j) x[i] = x[i] + x[i - 2^j];  // body using specific i
        barrier(mygroup);
}
```

prefix.c



```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

int main(int argc, char** argv){
  int rank, size;
  MPI_Status status;
  int local_sum, tmp;
  int i, iter;
  int distance;

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

  local_sum = rank;
  tmp = 0;

  iter = log(size) / log(2);
  //printf("Process %d has prefix sum %d\n", rank, local_sum);
  MPI_Barrier(MPI_COMM_WORLD);
  for (i = 0; i < iter; i++){
    distance = pow(2,i);

    if (rank == 0){
      printf("iter %d and distance %d\n", i, distance);
    }
    if (rank < (size - distance)){
      MPI_Send(&local_sum, 1, MPI_INT, rank + distance, 0, MPI_COMM_WORLD);
      printf("iter %d: %d send to %d value %d\n",i, rank, rank+distance, local_sum);

    }
    if (rank >= distance){
      MPI_Recv(&tmp, 1, MPI_INT, rank - distance, 0, MPI_COMM_WORLD, &status);
      printf("iter %d: %d receive from %d value %d\n",i, rank, rank-distance, tmp);
      local_sum += tmp;
    }
    printf("iter %d: Process %d has prefix sum %d\n", i, rank, local_sum);
    MPI_Barrier(MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```



```
[jin6@node0709 08-synchronous-computing]$ mpicc prefix.c -lm -o prefix
[jin6@node0709 08-synchronous-computing]$ mpirun -np 2 prefix
iter 0 and distance 1
iter 0: 0 send to 1 value 0
iter 0: Process 0 has prefix sum 0
iter 0: 1 receive from 0 value 0
iter 0: Process 1 has prefix sum 1
[jin6@node0709 08-synchronous-computing]$ mpirun -np 4 prefix
iter 0 and distance 1
iter 0: 0 send to 1 value 0
iter 0: Process 0 has prefix sum 0
iter 1 and distance 2
iter 1: 0 send to 2 value 0
iter 1: Process 0 has prefix sum 0
iter 0: 1 send to 2 value 1
iter 0: 1 receive from 0 value 0
iter 0: Process 1 has prefix sum 1
iter 1: 1 send to 3 value 1
iter 1: Process 1 has prefix sum 1
iter 0: 2 send to 3 value 2
iter 0: 2 receive from 1 value 1
iter 0: Process 2 has prefix sum 3
iter 1: 2 receive from 0 value 0
iter 1: Process 2 has prefix sum 3
iter 0: 3 receive from 2 value 2
iter 0: Process 3 has prefix sum 5
iter 1: 3 receive from 1 value 1
iter 1: Process 3 has prefix sum 6
```

Synchronized Computations

- Can be classified as:
  - Fully synchronous

Or

  - Locally synchronous

- In fully synchronous, all processes involved in the computation must be synchronized
- In locally synchronous, processes only need to synchronize with a set of logically nearby processes, not all processes involved in the computation

Fully Synchronous Problems – Cellular Automata

- The problem space is divided into cells
- Each cell can be in one of a finite number of states
- Cells affected by their neighbors according to certain rules, and all cells are affected simultaneously in a “generation”
- Rules re-applied in subsequent generations so that cells evolve, or change state, from generation to generation
- Most famous cellular automata is the ”Game of Life” devised by John Horton Conway, a Cambridge mathematician

The Game of Life

- Board game - theoretically infinite two-dimensional array of cells. Each cell can hold one “organism” and has eight neighboring cells, including those diagonally adjacent. Initially, some cells occupied. The following rules apply:
  - 1. Every organism with two or three neighboring organisms survives for the next generation
  - 2. Every organism with four or more neighbors dies from overpopulation
  - 3. Every organism with one neighbor or none dies from isolation
  - 4. Each empty cell adjacent to exactly three occupied neighbors will give birth to an organism
- These rules were derived by Conway “after a long period of experimentation”

Sharks and Fishes

- An ocean could be modeled as a three-dimensional array of cells
- Each cell can hold one fish or one shark (but not both)
- Fish and sharks follow “rules”

Fish

1. If there is one empty adjacent cell, the fish moves to this cell.
2. If there is more than one empty adjacent cell, the fish moves to one cell chosen at random.
3. If there are no empty adjacent cells, the fish stays where it is.
4. If the fish moves and has reached its breeding age, it gives birth to a baby fish, which is left in the vacating cell.
5. Fish die after *x* generations.

Sharks

1. If one adjacent cell is occupied by a fish, the shark moves to this cell and eats the fish.
2. If more than one adjacent cell is occupied by a fish, the shark chooses one fish at random, moves to the cell occupied by the fish, and eats the fish.
3. If no fish are in adjacent cells, the shark chooses an unoccupied adjacent cell to move to in a similar manner as fish move.
4. If the shark moves and has reached its breeding age, it gives birth to a baby shark, which is left in the vacating cell.
5. If a shark has not eaten for *y* generations, it dies.

Serious Applications for Cellular Automata

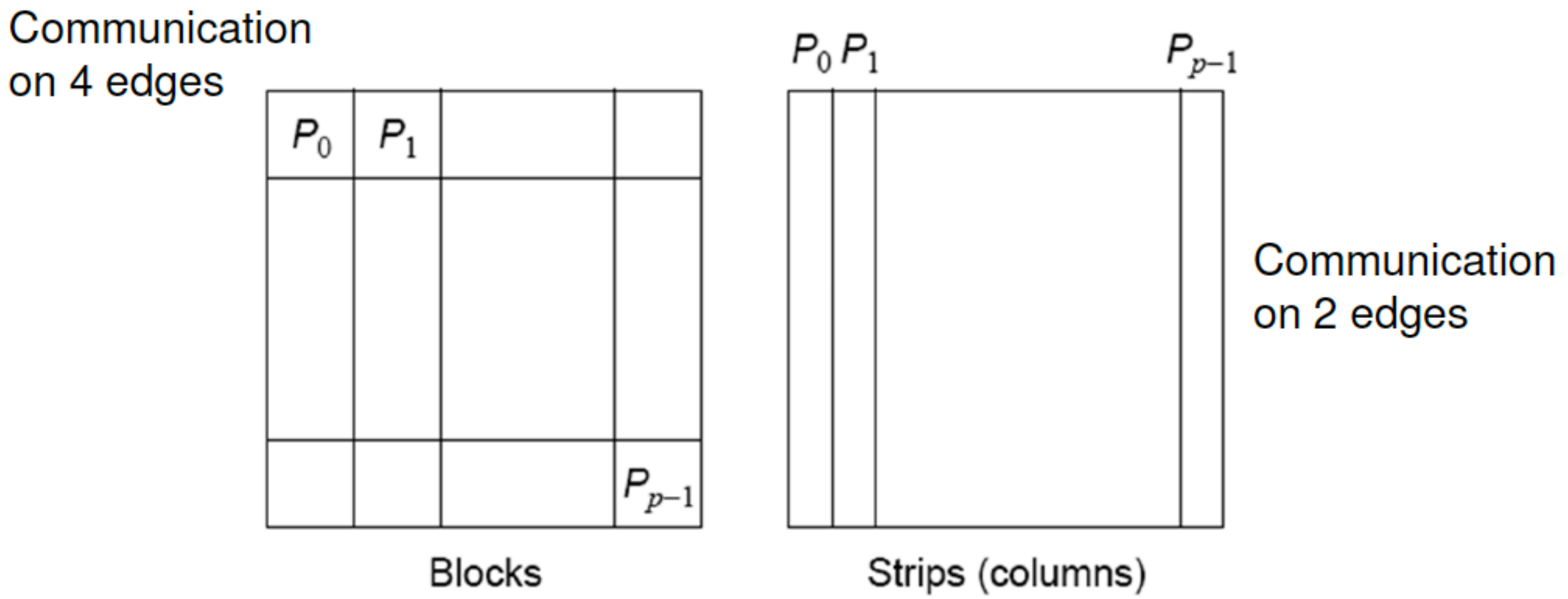
- Examples
    - fluid/gas dynamics
        - the movement of fluids and gases around objects
        - diffusion of gases
    - biological growth
    - airflow across an airplane wing
    - erosion/movement of sand at a beach or riverbank

Partially Synchronous Computations

- Computations in which individual processes operate without needing to synchronize with other processes on every iteration
- Important idea because synchronizing processes very significantly slows the computation and a major cause for reduced performance of parallel programs

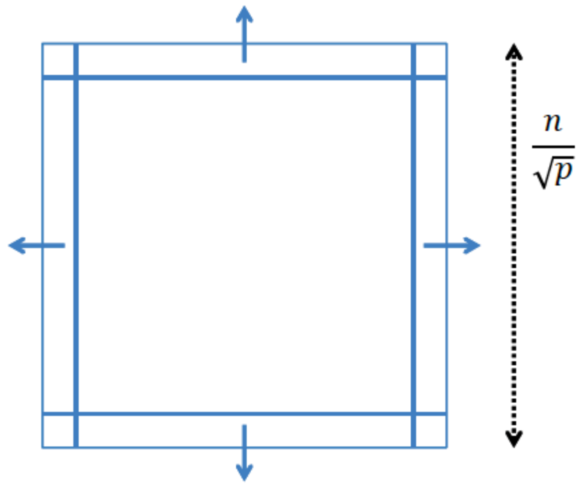
Partitioning

- Points could be partitioned into square blocks or strips:

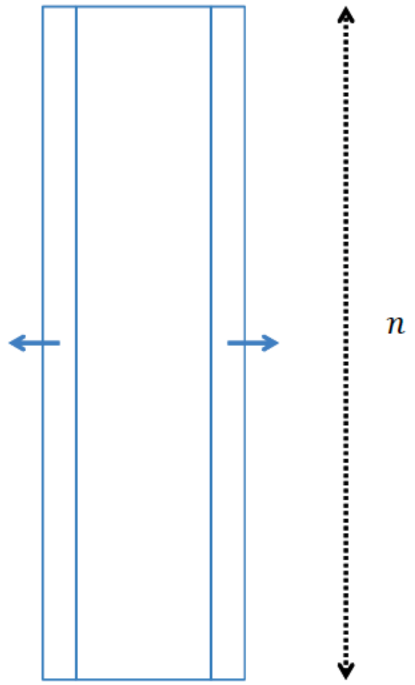


- With n^2 points, p processors, and equal partitions, each partition holds n^2/p points

Communication Consequences of Partitioning



Square blocks



Strips

Block Partition:

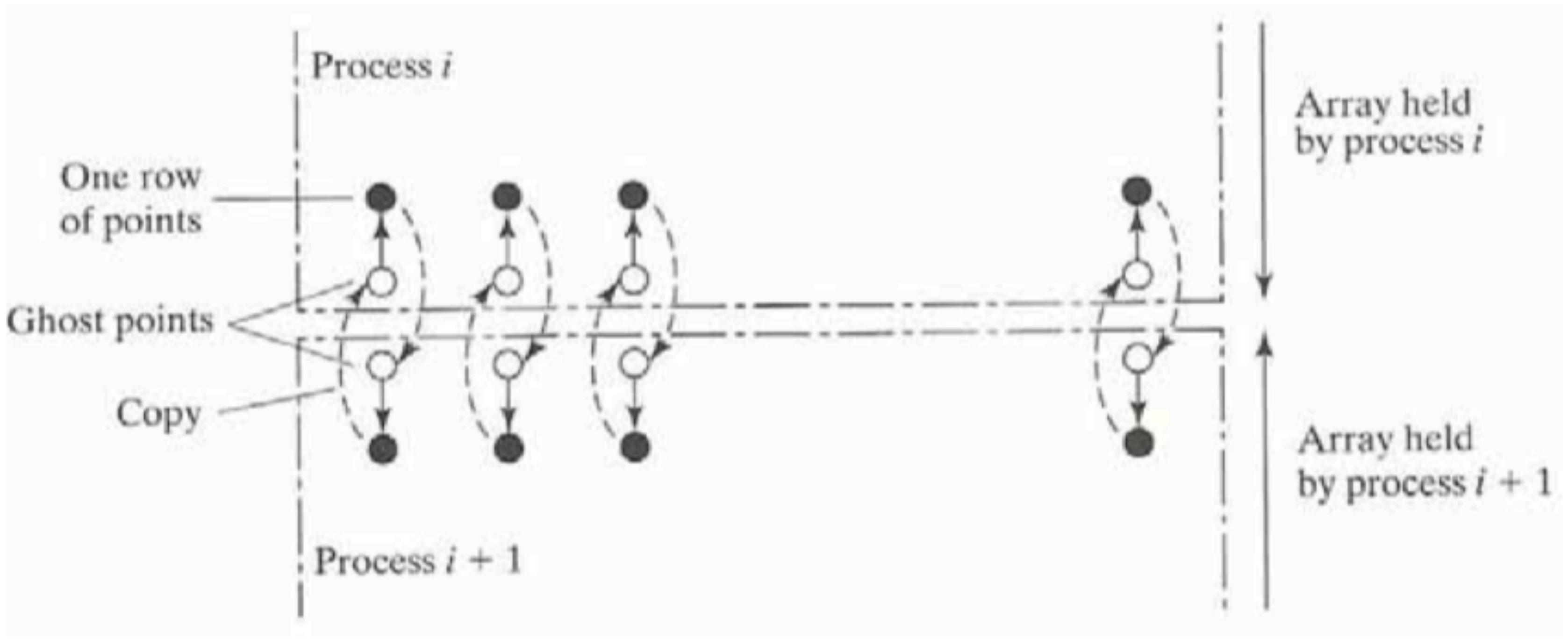
$$t_{comm\_block} = 8\left(t_{startup} + \frac{n}{\sqrt{p}} t_{data}\right)$$

Stripe Partition:

$$t_{comm\_stripe} = 4\left(t_{startup} + n t_{data}\right)$$

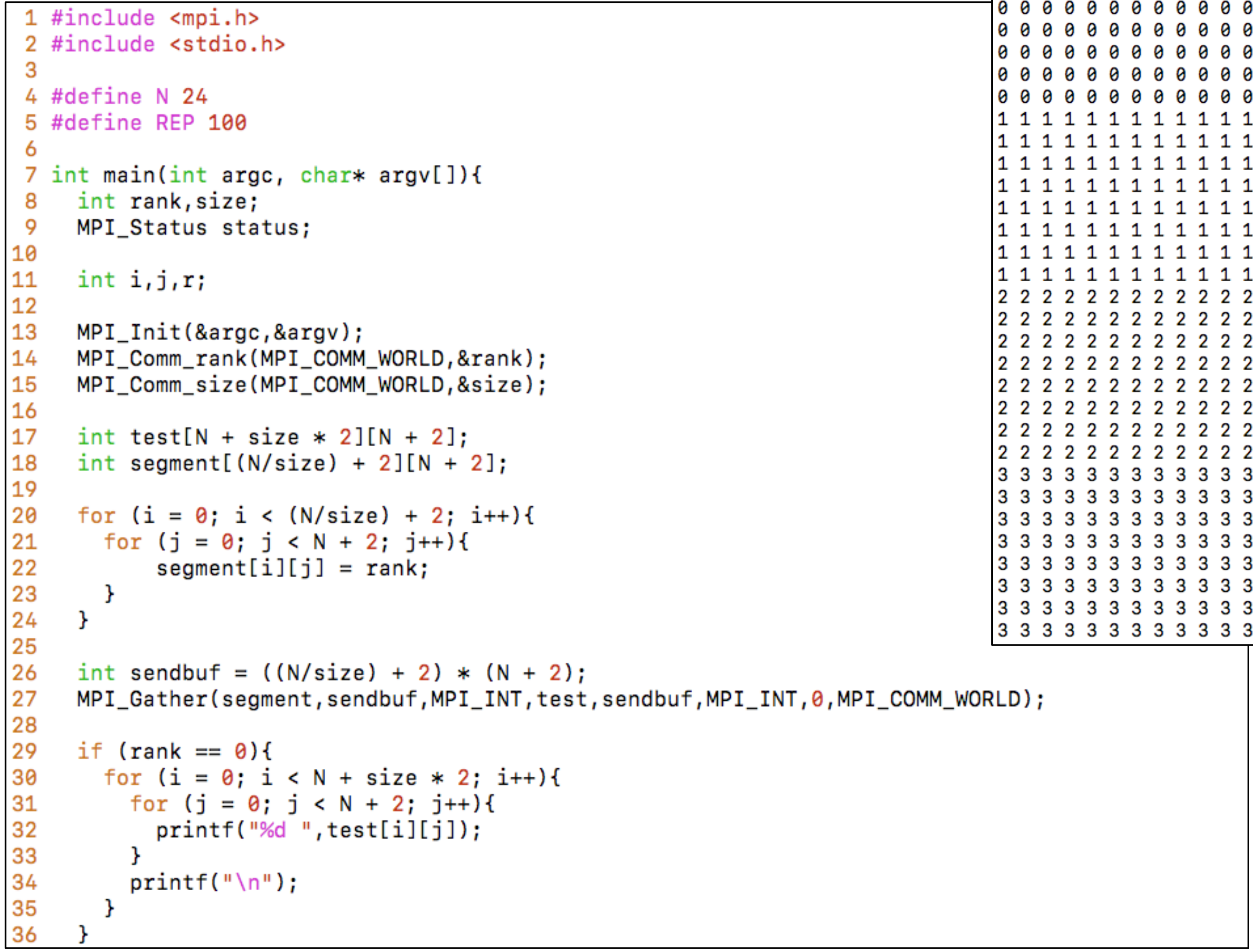
In general, strip partition best for large communication startup time, and block partition best for small startup time.

Row Major Order Partition and Ghost Points

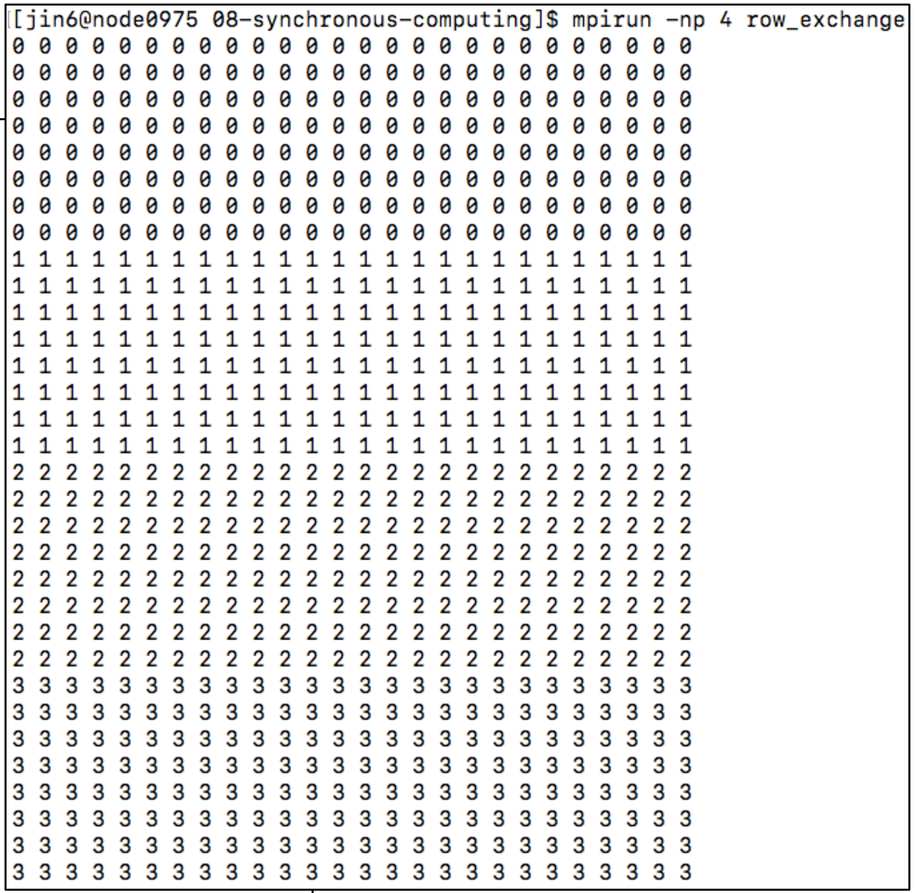


Configurating array into contiguous rows for each process, with ghost points

row_exchange.c

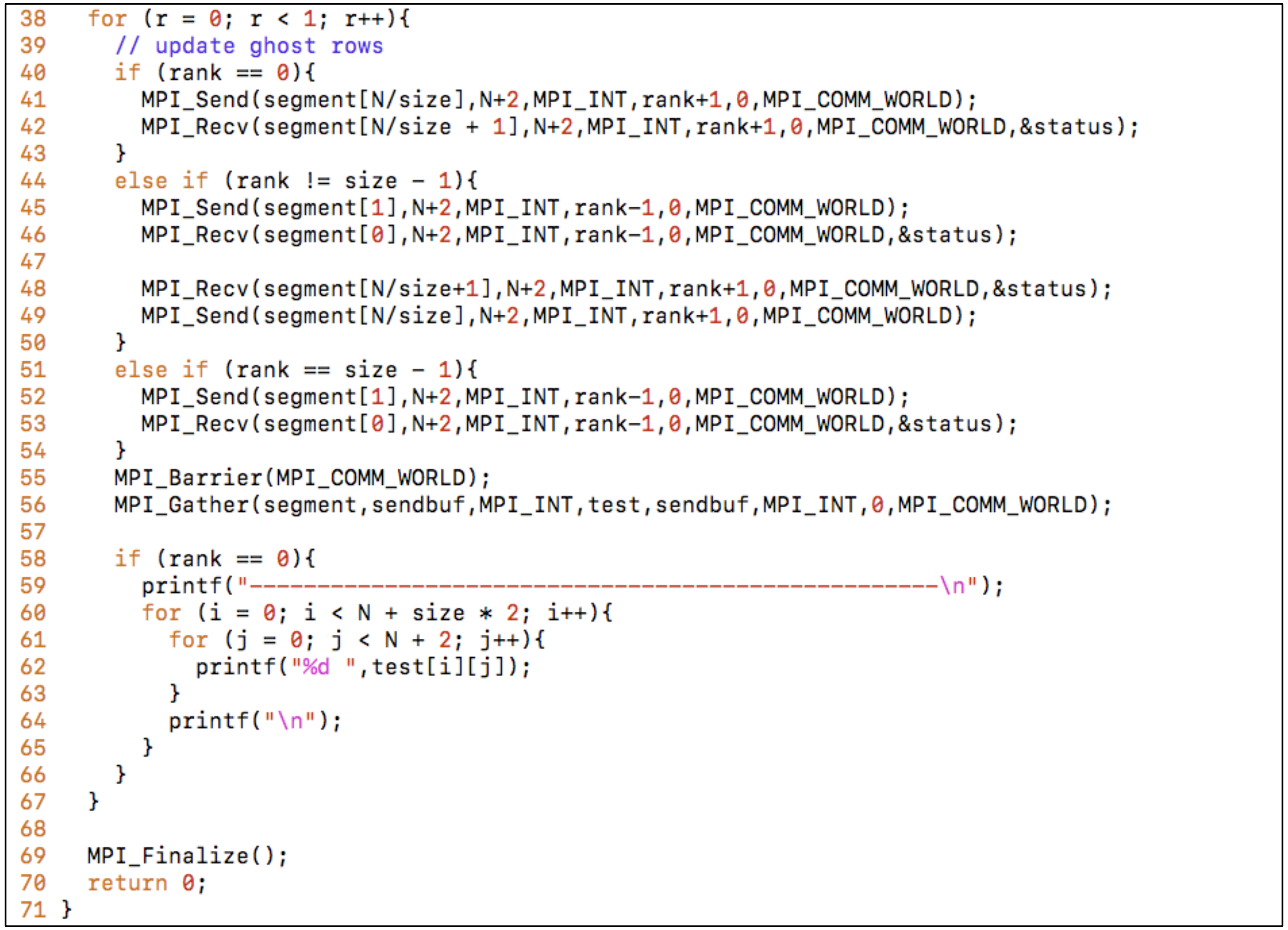


```c
#include <mpi.h>
#include <stdio.h>

#define N 24
#define REP 100

int main(int argc, char* argv[]){
  int rank,size;
  MPI_Status status;

  int i,j,r;

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

  int test[N + size * 2][N + 2];
  int segment[(N/size) + 2][N + 2];

  for (i = 0; i < (N/size) + 2; i++){
    for (j = 0; j < N + 2; j++){
        segment[i][j] = rank;
    }
  }

  int sendbuf = ((N/size) + 2) * (N + 2);
  MPI_Gather(segment,sendbuf,MPI_INT,test,sendbuf,MPI_INT,0,MPI_COMM_WORLD);

  if (rank == 0){
    for (i = 0; i < N + size * 2; i++){
      for (j = 0; j < N + 2; j++){
        printf("%d ",test[i][j]);
      }
      printf("\n");
    }
  }
```

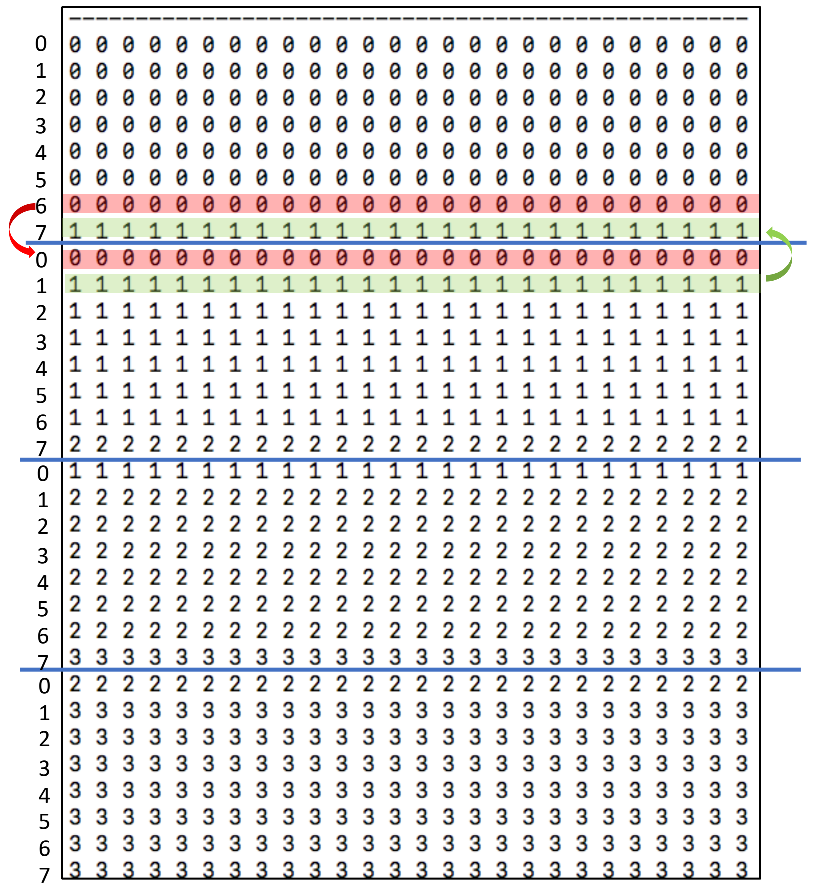


```
[jin6@node0975 08-synchronous-computing]$ mpirun -np 4 row_exchange
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

row_exchange.c (Cont.)



```c
  for (r = 0; r < 1; r++){
    // update ghost rows
    if (rank == 0){
      MPI_Send(segment[N/size],N+2,MPI_INT,rank+1,0,MPI_COMM_WORLD);
      MPI_Recv(segment[N/size + 1],N+2,MPI_INT,rank+1,0,MPI_COMM_WORLD,&status);
    }
    else if (rank != size - 1){
      MPI_Send(segment[1],N+2,MPI_INT,rank-1,0,MPI_COMM_WORLD);
      MPI_Recv(segment[0],N+2,MPI_INT,rank-1,0,MPI_COMM_WORLD,&status);

      MPI_Recv(segment[N/size+1],N+2,MPI_INT,rank+1,0,MPI_COMM_WORLD,&status);
      MPI_Send(segment[N/size],N+2,MPI_INT,rank+1,0,MPI_COMM_WORLD);
    }
    else if (rank == size - 1){
      MPI_Send(segment[1],N+2,MPI_INT,rank-1,0,MPI_COMM_WORLD);
      MPI_Recv(segment[0],N+2,MPI_INT,rank-1,0,MPI_COMM_WORLD,&status);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Gather(segment,sendbuf,MPI_INT,test,sendbuf,MPI_INT,0,MPI_COMM_WORLD);

    if (rank == 0){
      printf("---------------------------------------------------\n");
      for (i = 0; i < N + size * 2; i++){
        for (j = 0; j < N + 2; j++){
          printf("%d ",test[i][j]);
        }
        printf("\n");
      }
    }
  }

  MPI_Finalize();
  return 0;
}
```



```
  ---------------------------------------------------
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
7 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
5 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
6 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
7 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
4 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
5 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
6 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
7 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
5 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
7 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```