High Performance Computing
Term 4 2018/2019

Lecture 4

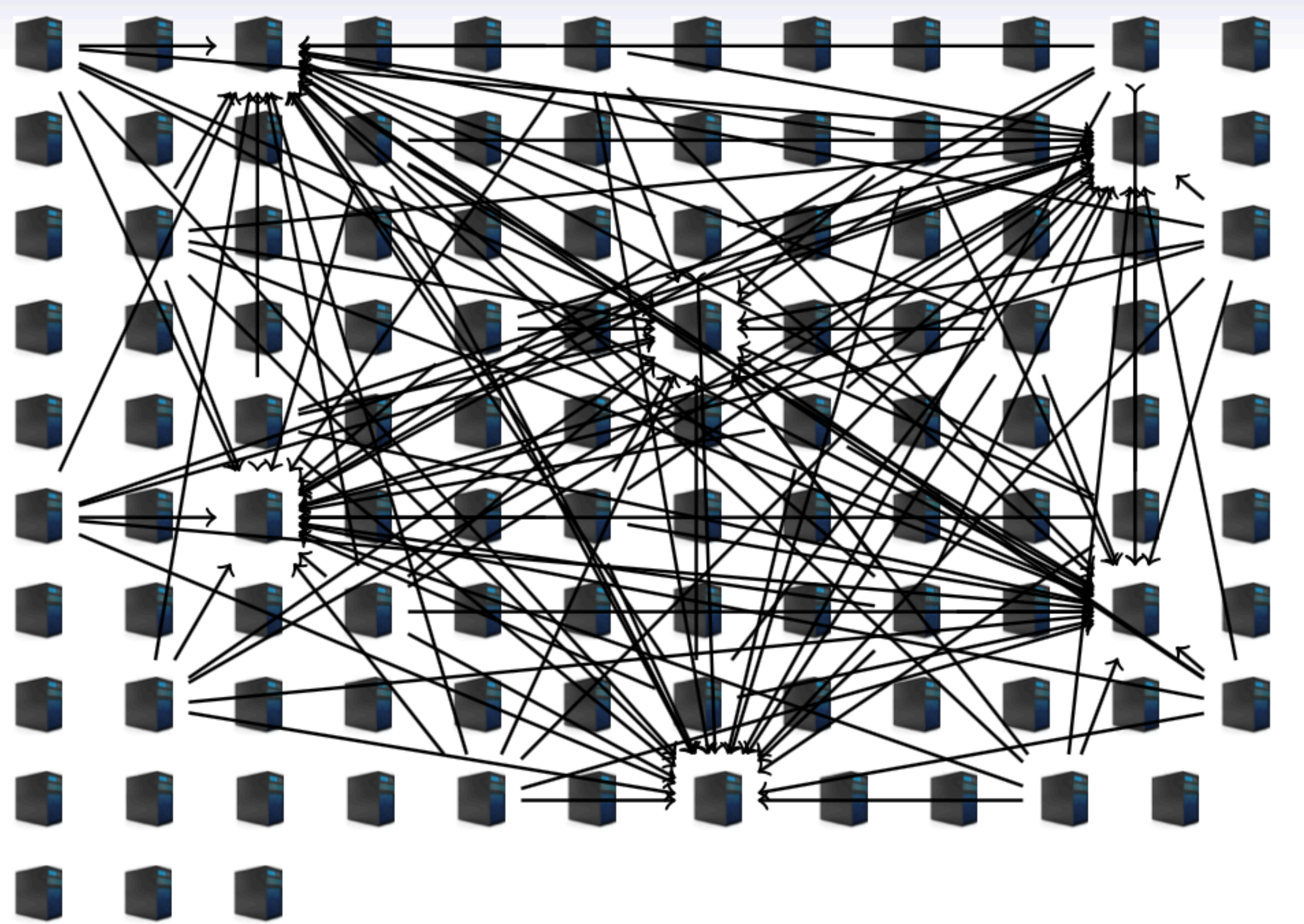
MPI

Message Passing Interface:

http://mpitutorial.com/

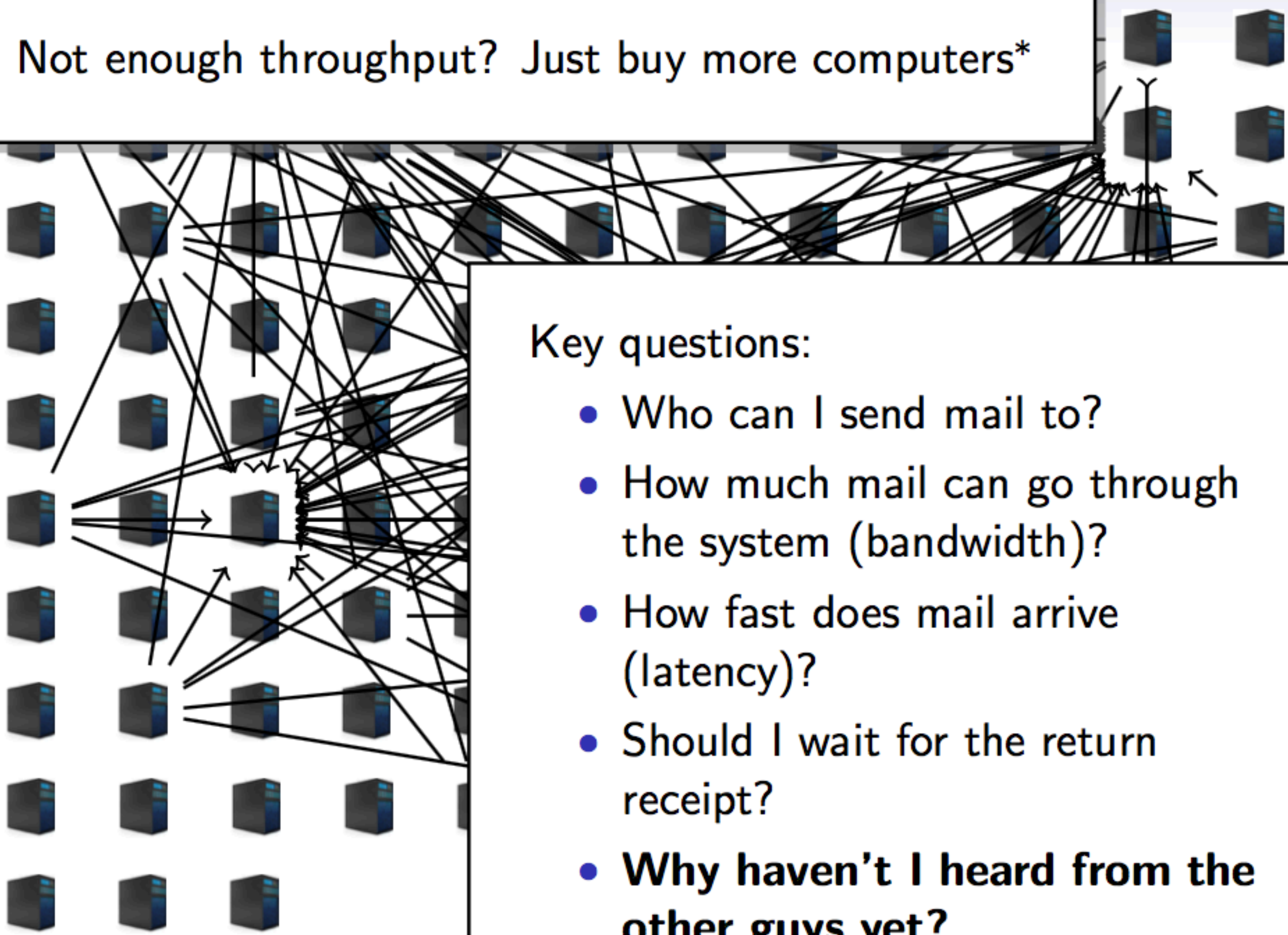
MPI

MPI



MPI

Not enough throughput? Just buy more computers*



Key questions:

- Who can I send mail to?
- How much mail can go through the system (bandwidth)?
- How fast does mail arrive (latency)?
- Should I wait for the return receipt?
- **Why haven't I heard from the other guys yet?**

MPI

Since 1992, when there were several unstandardized Message Passing frameworks.

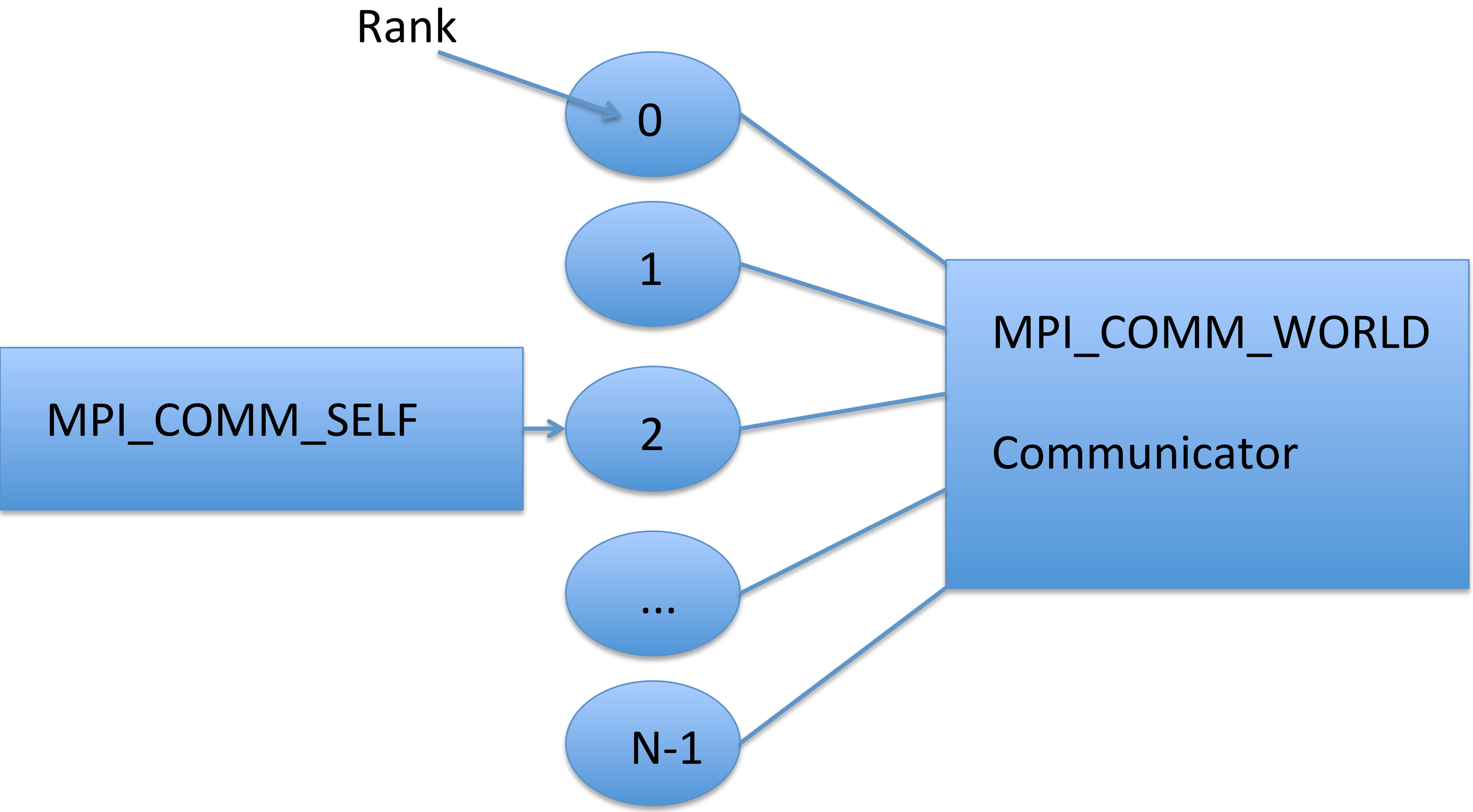
more than 40 companies envolved in the development.

Goal: make sure programs run on many different architectures without losing productivity.

Different versions (more or less compatible):
MPICH, **OpenMPI**, IntelMPI (commercial)

More than 120 commands, only 10 are mostly used.

MPI



Independent processes – no shared resources.
Communication only via sending messages.

MPI

all identifiers and names of → procedures, functions, data types, constants → MPI_

Minimal code

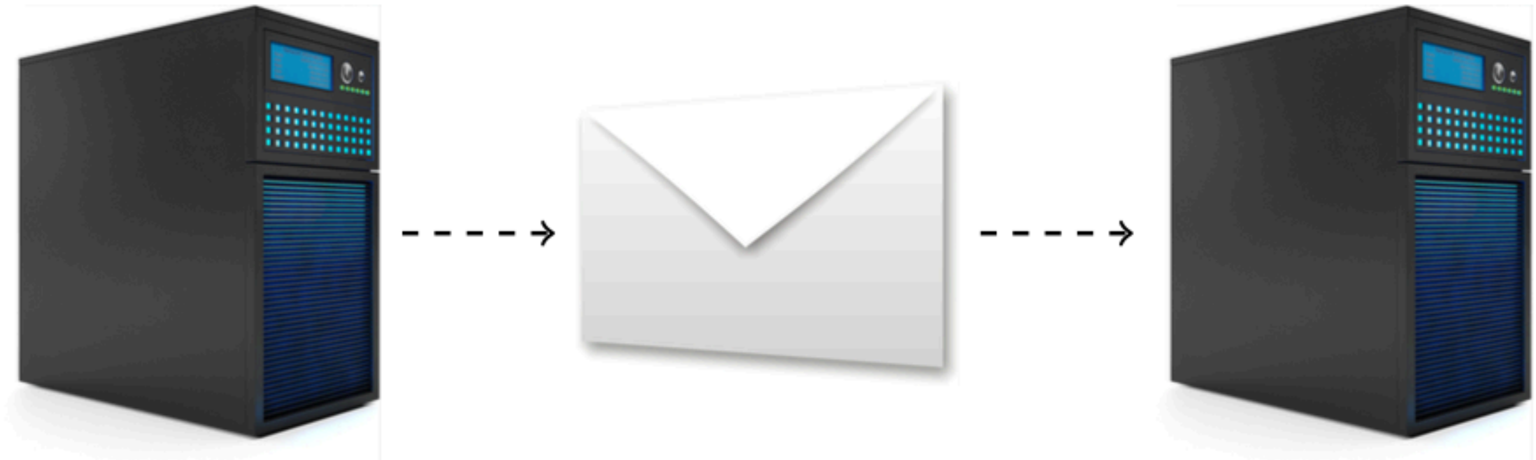
```
#include “mpi.h“
....

MPI_Init(&argc, &argv);
....
MPI_Comm_rank(MPI_COMM_WORLD, &prank);
MPI_Comm_size(MPI_COMM_WORLD, &psize);
....
MPI_Finalize();
```

MPI hello world

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

int main(int argc, char ** argv)
{
    int psize;
    int prank;
    MPI_Status status;

    int ierr;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &prank);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &psize);

    if (prank == 0)
    {
        printf("The number of processes available is %d\n", psize);
    }

    printf("Hello world from process[%d]\n", prank);

    ierr = MPI_Finalize();
    return 0;
}
```

MPI. Point-to-point communication

Message Passing Interface:



Blocking communication

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

MPI Datatypes

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

MPI Send&Recv example

```
// Find out rank, size
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int number;
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
           number);
}
```

MPI Send&Recv example

```
if(rank==0)
{
    MPI_Send(x to process 1)
    MPI_Recv(y from process 1)
}
if(rank==1)
{
    MPI_Send(y to process 0);
    MPI_Recv(x from process 0);
}
```

MPI. Point-to-point communication

Non-Blocking communication

```
MPI_Status status;

MPI_Request request;

MPI_Isend(
    &count, 1, MPI_INT, dest, prank, MPI_COMM_WORLD, &request
);

MPI_Irecv(
    &count, 1, MPI_INT, source, source, MPI_COMM_WORLD, &request
);
```

Testing whether the message has arrived:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)


int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

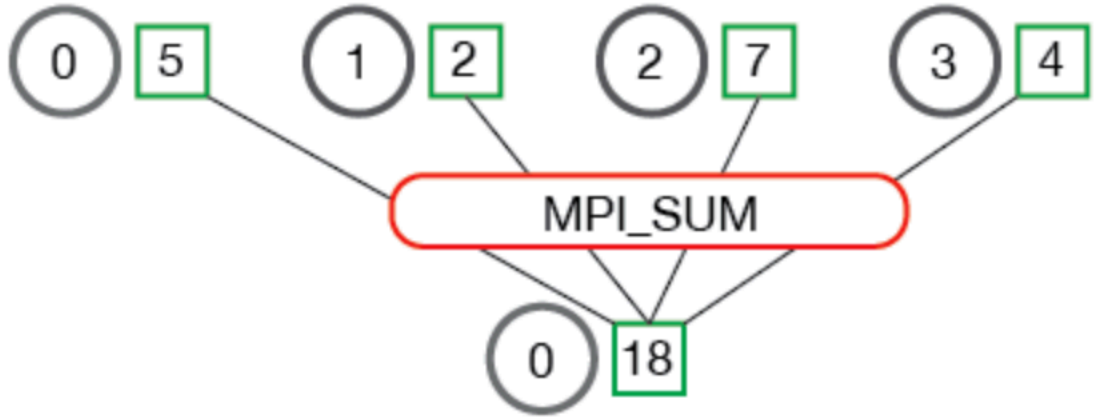
MPI. Collective communications

```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

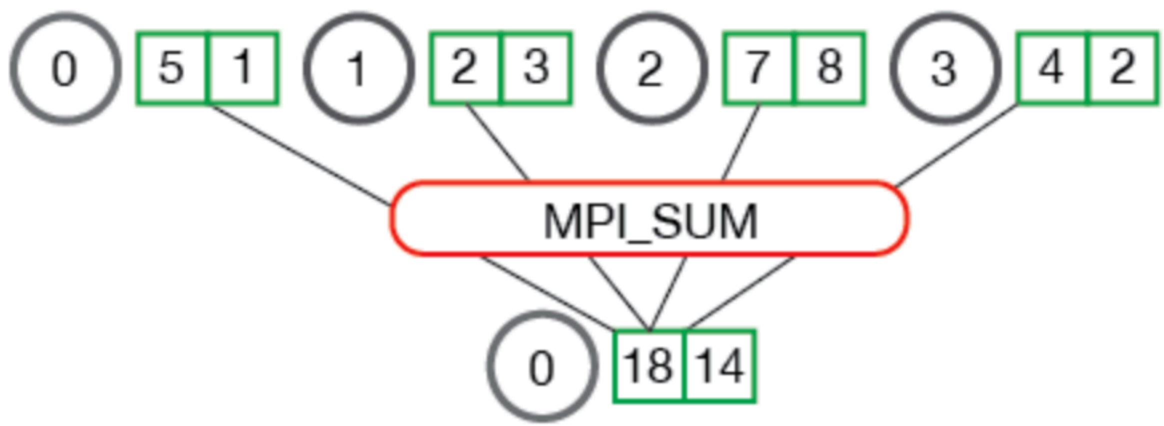
- `MPI_MAX` - Returns the maximum element.
- `MPI_MIN` - Returns the minimum element.
- `MPI_SUM` - Sums the elements.
- `MPI_PROD` - Multiplies all elements.
- `MPI_LAND` - Performs a logical *and* across the elements.
- `MPI_LOR` - Performs a logical *or* across the elements.
- `MPI_BAND` - Performs a bitwise *and* across the bits of the elements.
- `MPI_BOR` - Performs a bitwise *or* across the bits of the elements.
- `MPI_MAXLOC` - Returns the maximum value and the rank of the process that owns it.
- `MPI_MINLOC` - Returns the minimum value and the rank of the process that owns it.

MPI. Collective communications

MPI_Reduce

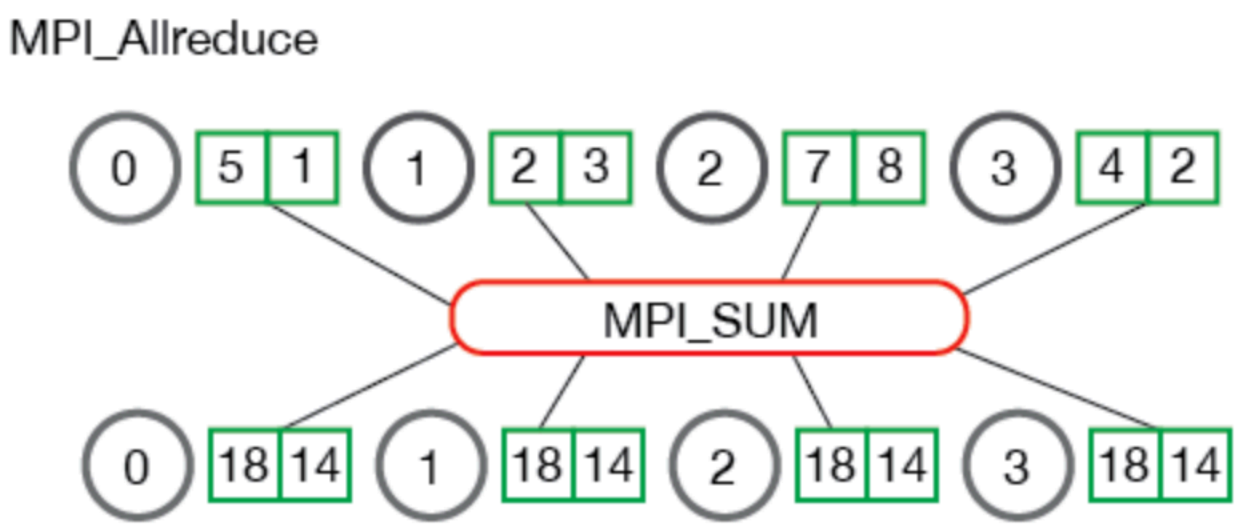


MPI_Reduce



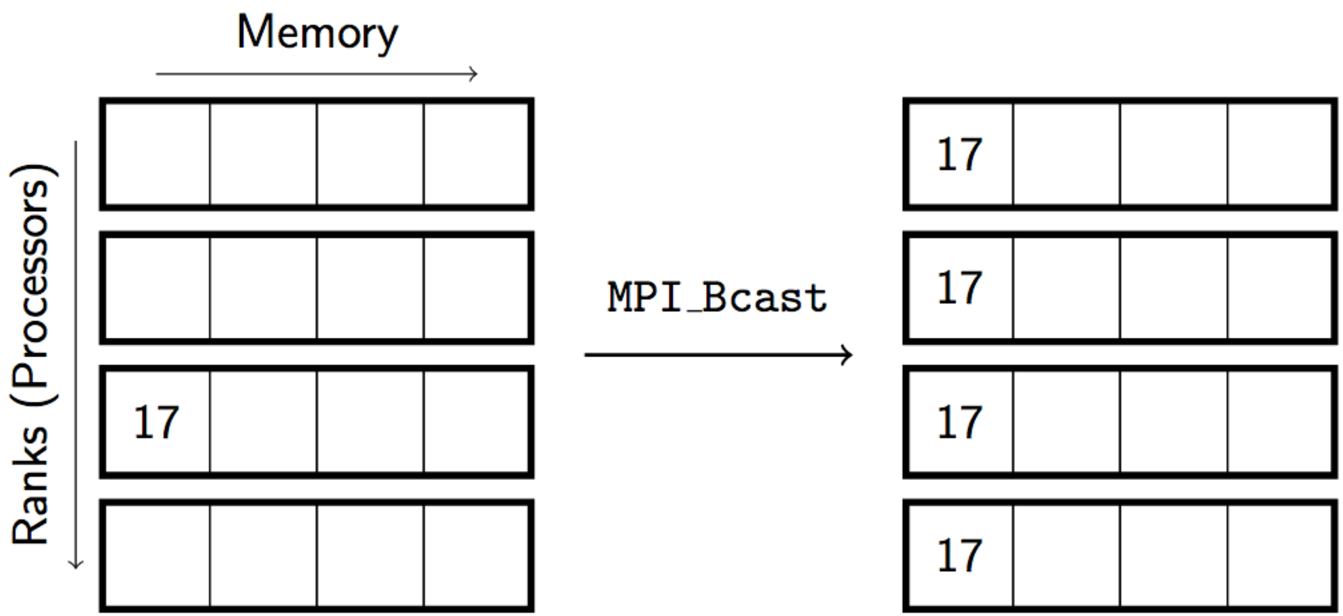
MPI. Collective communications

```
MPI_Allreduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm communicator)
```



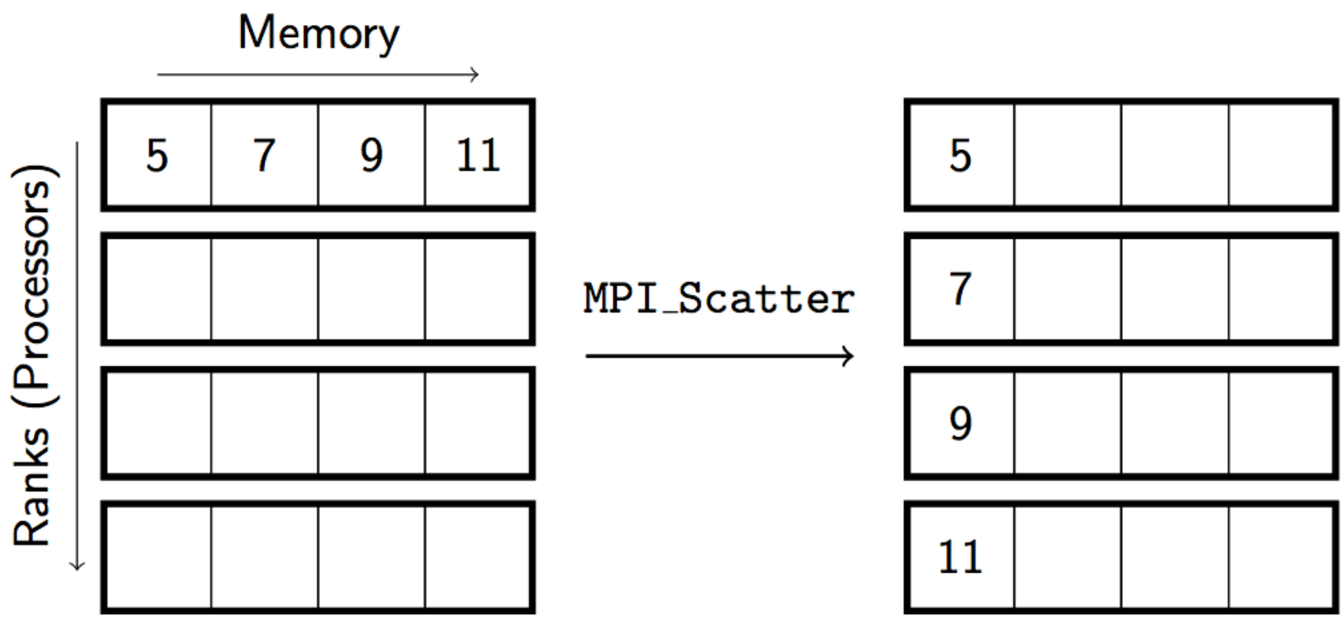
MPI. Collective communications

Broadcast



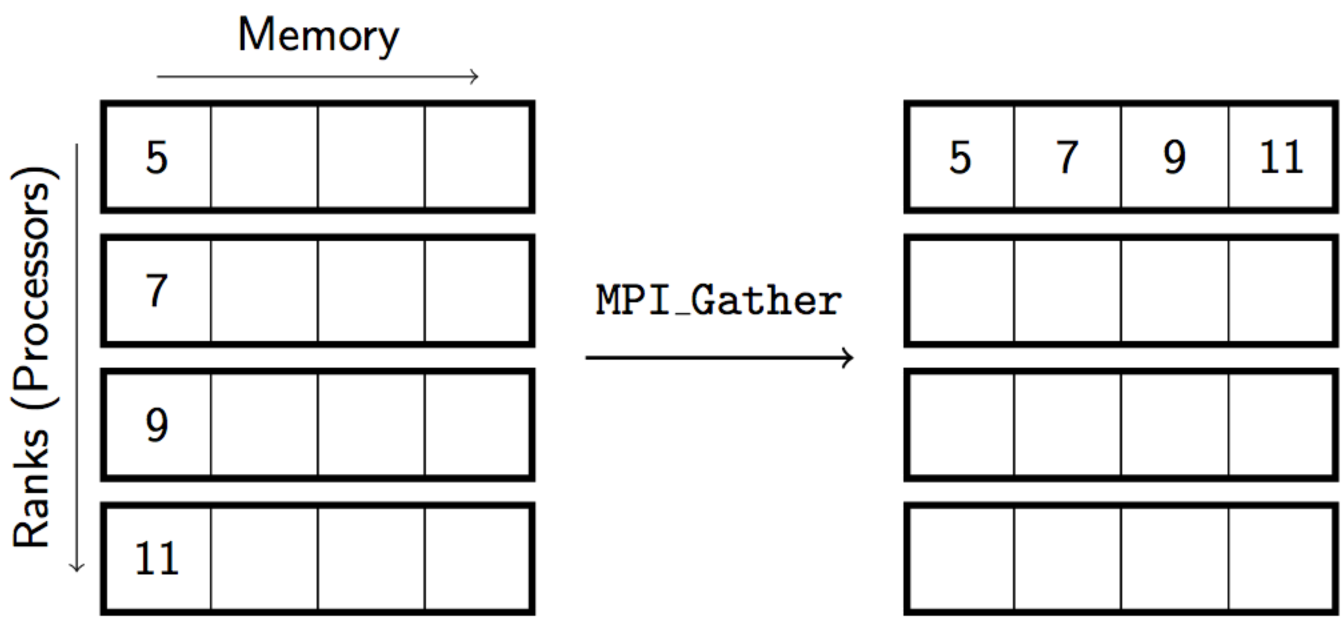
MPI. Collective communications

Scatter



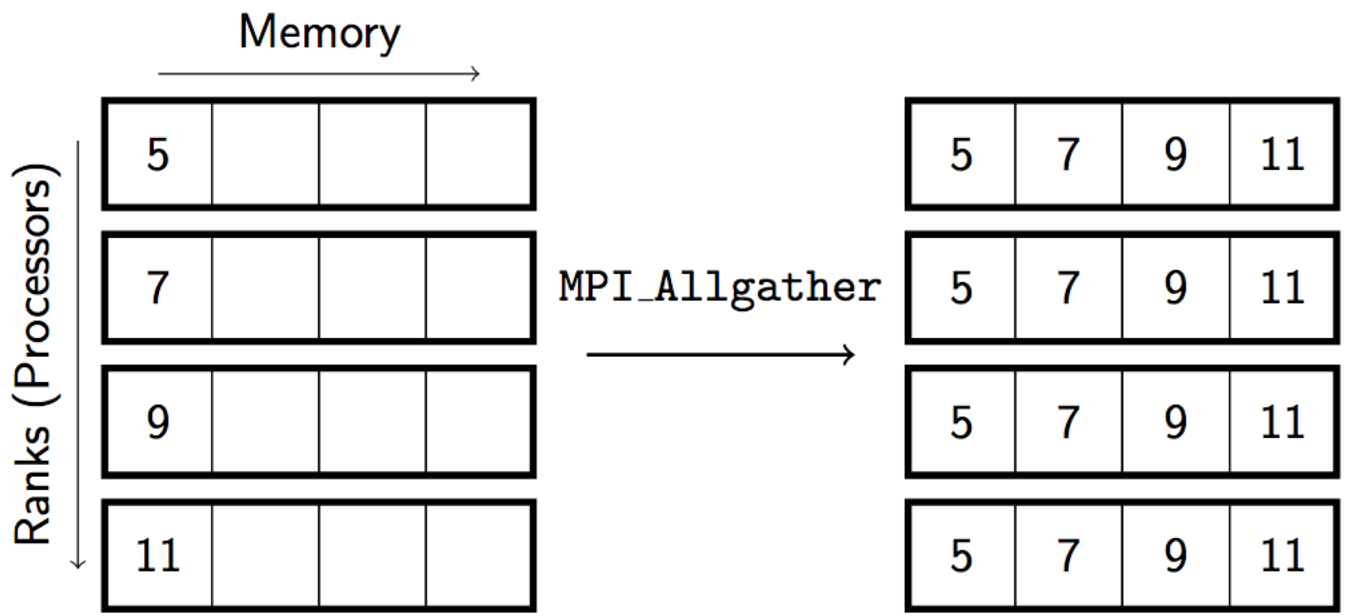
MPI. Collective communications

Gather



MPI. Collective communications

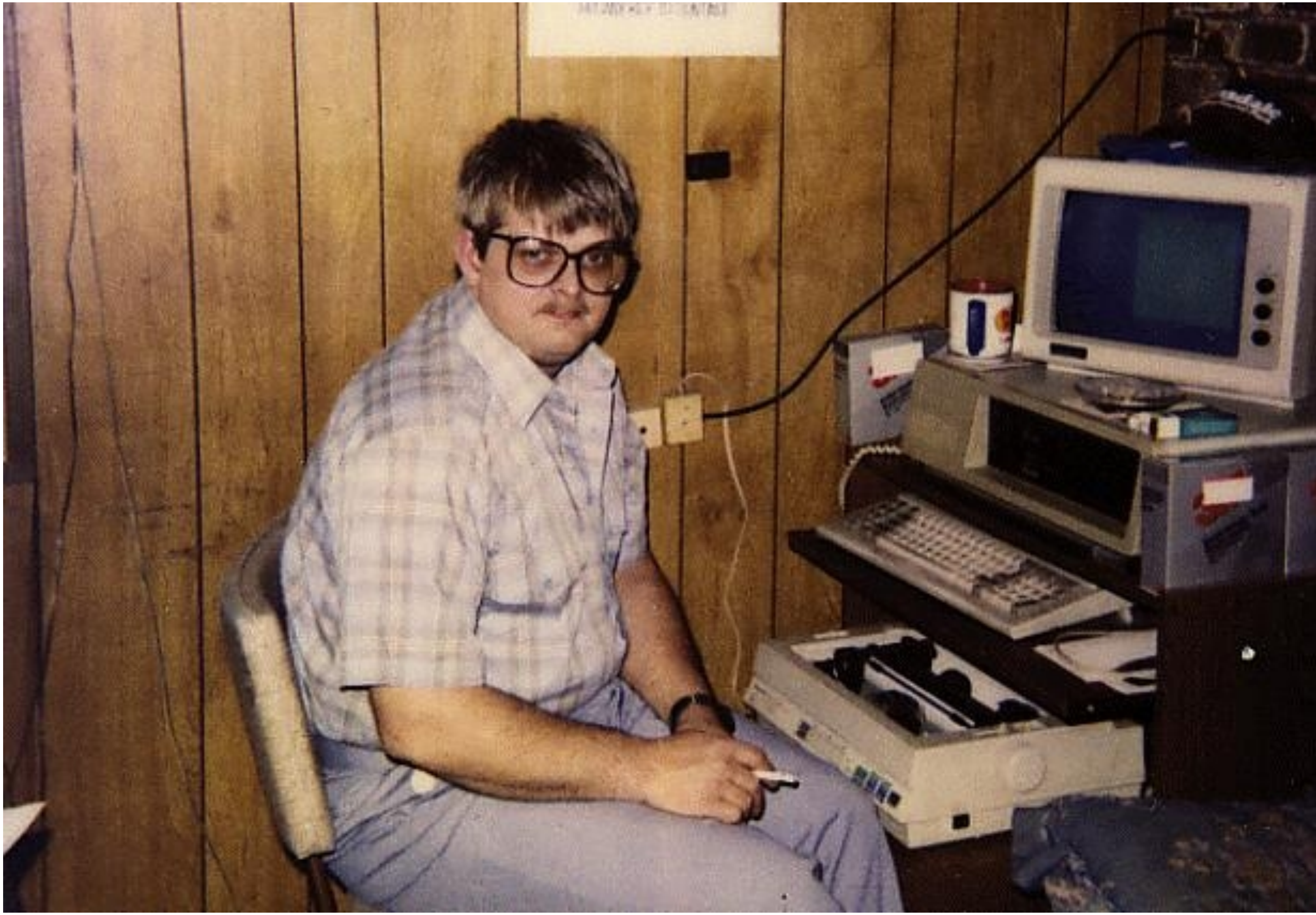
All-Gather



MPI measuring wall time

```
#include “mpi.h“
....

MPI_Init(&argc, &argv);
....
MPI_Comm_rank(MPI_COMM_WORLD, &prank);
MPI_Comm_size(MPI_COMM_WORLD, &psize);
....

double time_elapsed = MPI_Wtime();
// Computations
// more computations

time_elapsed = MPI_Wtime() – time_elapsed;

MPI_Finalize();
```

You know what this means...



Superproblem for today/tomorrow

Journal of Mathematical Sociology
1971, Vol. 1, pp 143–186

DYNAMIC MODELS OF SEGREGATION†

THOMAS C. SCHELLING
Harvard University

City map: each square is a household

0 and 1 − different status (rich/poor, angry/fun etc)

each square: if more than (for example) half of neighbours have the same status − stay,
if not − randomly change location (move)

process repeats over and over

Journal of Mathematical Sociology
1971, Vol. 1, pp 143–186

DYNAMIC MODELS OF SEGREGATION†

THOMAS C. SCHELLING
Harvard University