

HPC Seminar

Parallelization on a Multiple Nodes – Part II

Sergey Yakubov - Maxwell Team - DESY IT
Hamburg, 25.06.2018

Agenda

- Recap from the previous seminar
- Parallelization of a computational problem
- Non-blocking communications
- Collective communications
- Summary

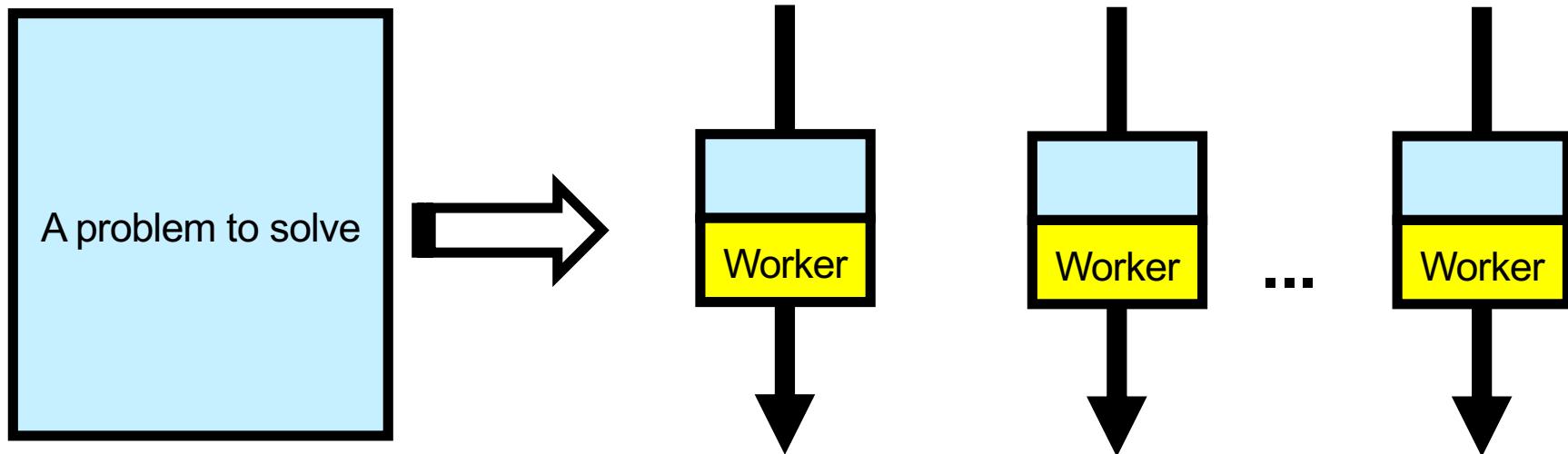
Last time...

- MPI library was created for parallel processing on distributed memory systems
 - available for various languages (C/C++, Python, ...)
 - architecture and hardware independent
 - implements interprocess communications via messages
 - can also be used on shared memory systems
- Starts specified amount of processes and runs an independent program instance on each of them
- A set of send/receive routines provides point-to-point message exchange
 - blocking communication calls return only when send/recieve buffer can be reused
- Deadlocks may occur with use of blocking MPI calls

Parallelization

Approaches – Embarrassingly Parallel

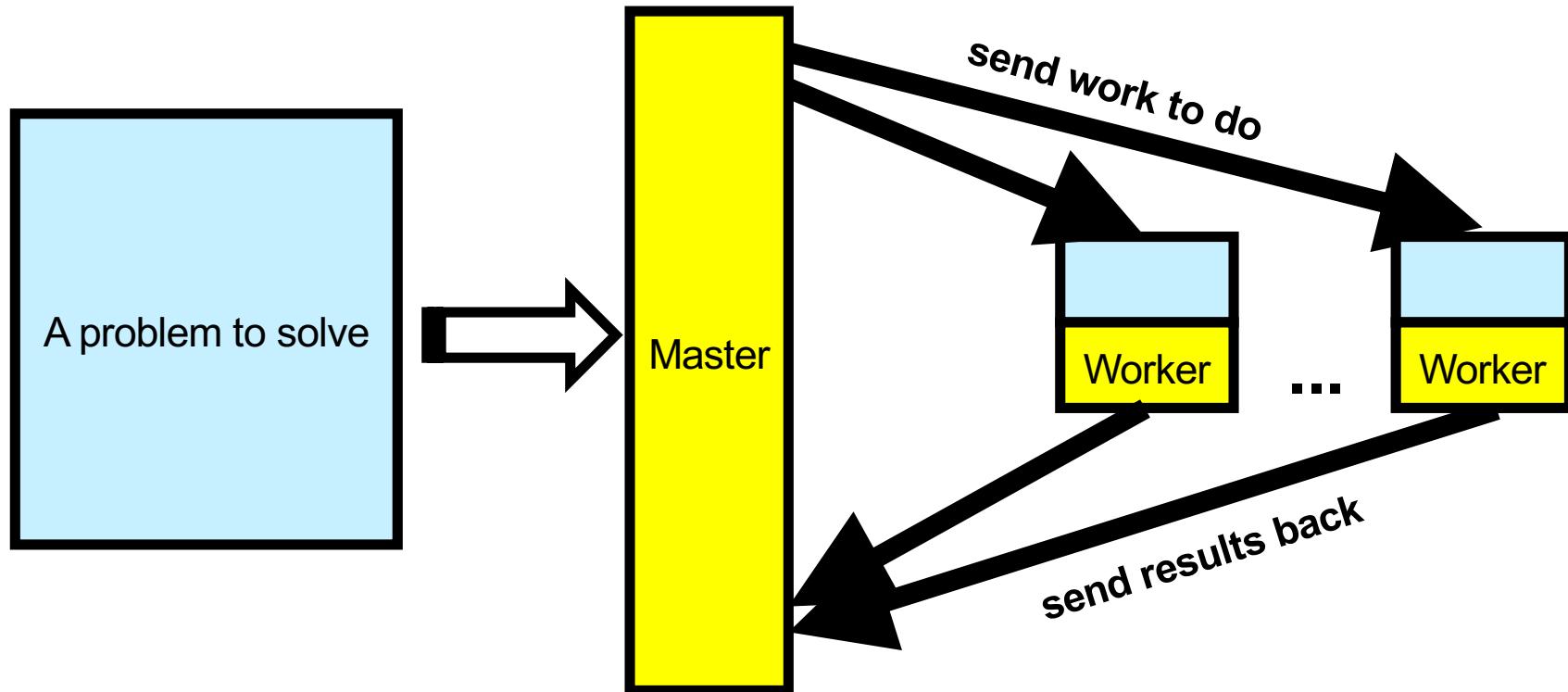
- “Embarrassingly parallel”
 - no effort is needed to separate the problem into a number of parallel tasks



Parallelization

Approaches – Embarrassingly Parallel

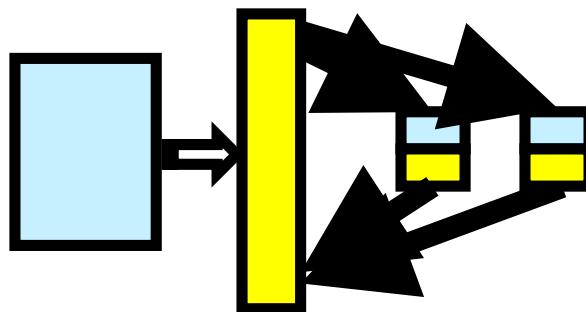
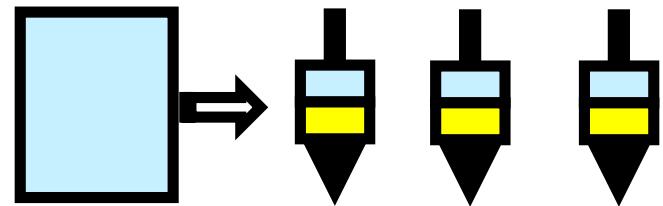
- “Nearly embarrassingly parallel”
 - little effort is needed to separate the problem into a number of parallel tasks



Parallelization

Embarrassingly Parallel - Examples

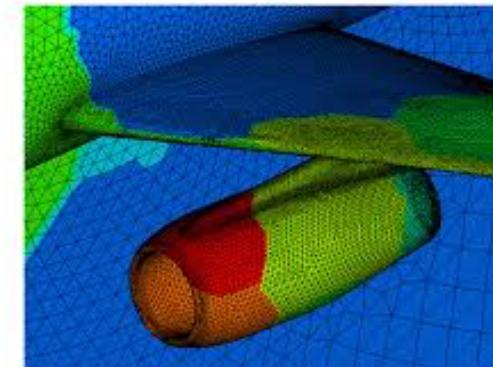
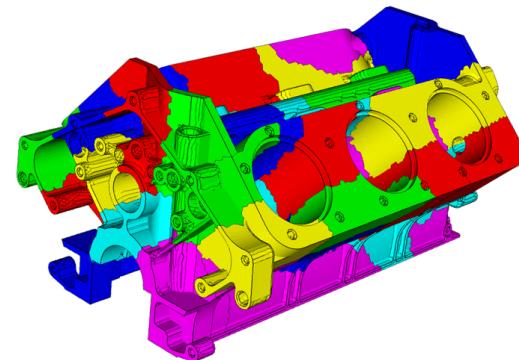
- HEP – event simulation and reconstruction
- Photon science – image processing (independent)
- Rendering in computer graphics (pixels are independent)
- Face recognition system
- Simulations comparing independent scenarios
 - e.g. climate models
- Discrete Fourier transform
-



Parallelization

Approaches – Domain Decomposition

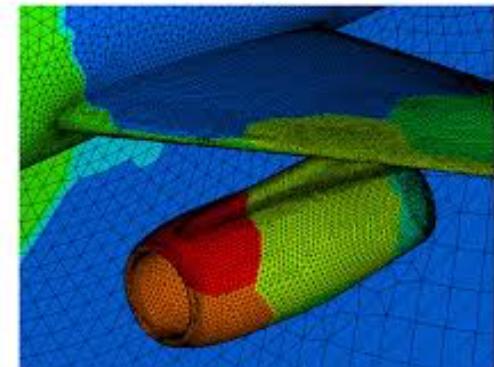
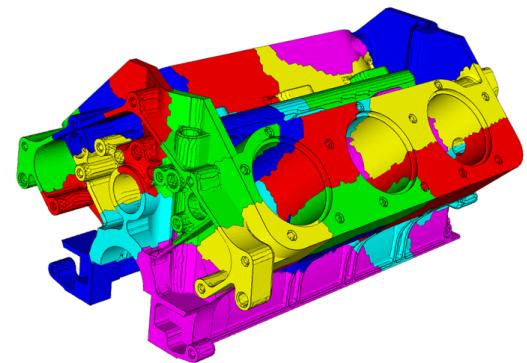
- Domain decomposition methods
 - split your geometrical/mathematical domain on multiple subdomains, each CPU take care of a single one
- Challenges
 - partition your domain most efficiently
 - subdomain sizes are equal
 - single process allocates memory only for its own subdomain
 - minimize boundaries
 - keyword – ParMETIS
 - exchange information between subdomains
 - may become a bottleneck
 - I/O should be parallelized as well
 - Result post-processing
 - Paraview, Visit, Tecplot, ...



Parallelization

Examples – Domain Decomposition

- Computational Fluid Dynamics
 - heat transfer, mass transfer, phase change, chemical reactions
- Structural mechanics
 - deformations, deflections, forces/stresses
- Climate studies
- Everything where computations with matrices are needed



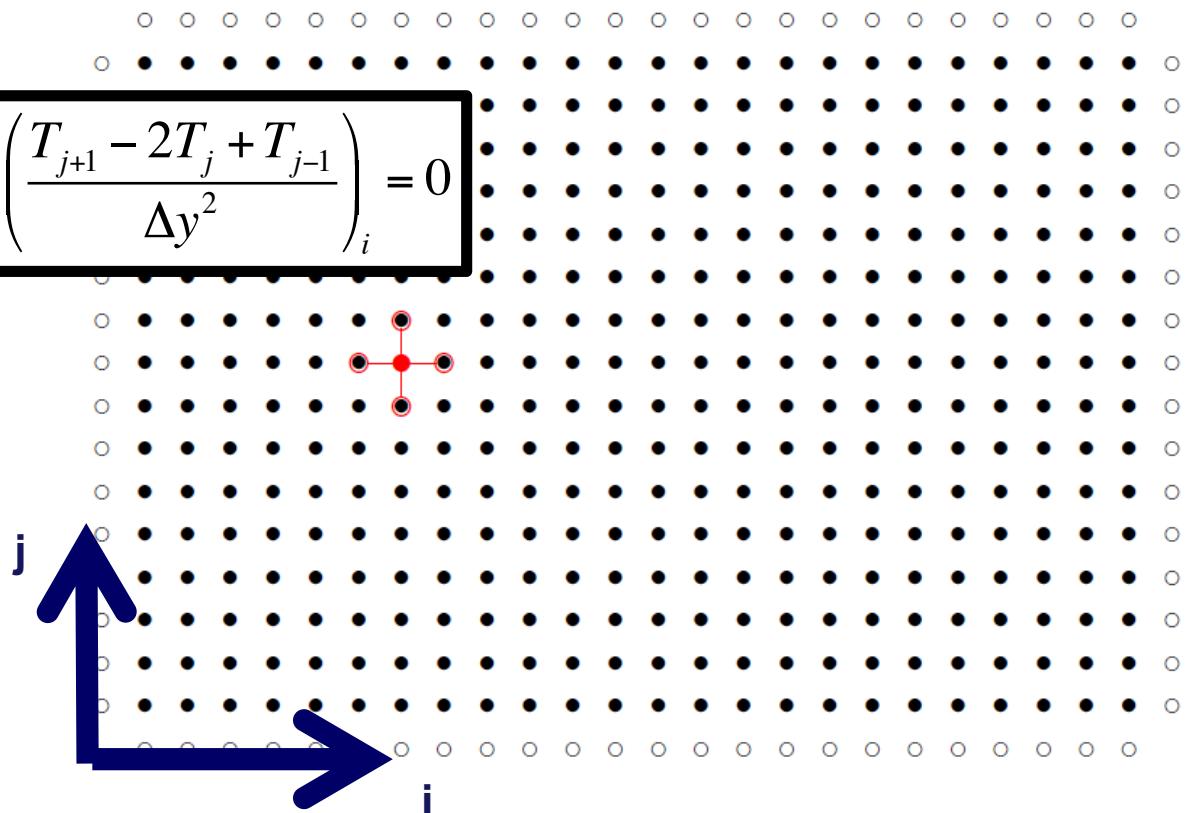
Parallelization

Example – Laplace equation (from introduction seminar)

Data values in neighbour points are needed to compute local value

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

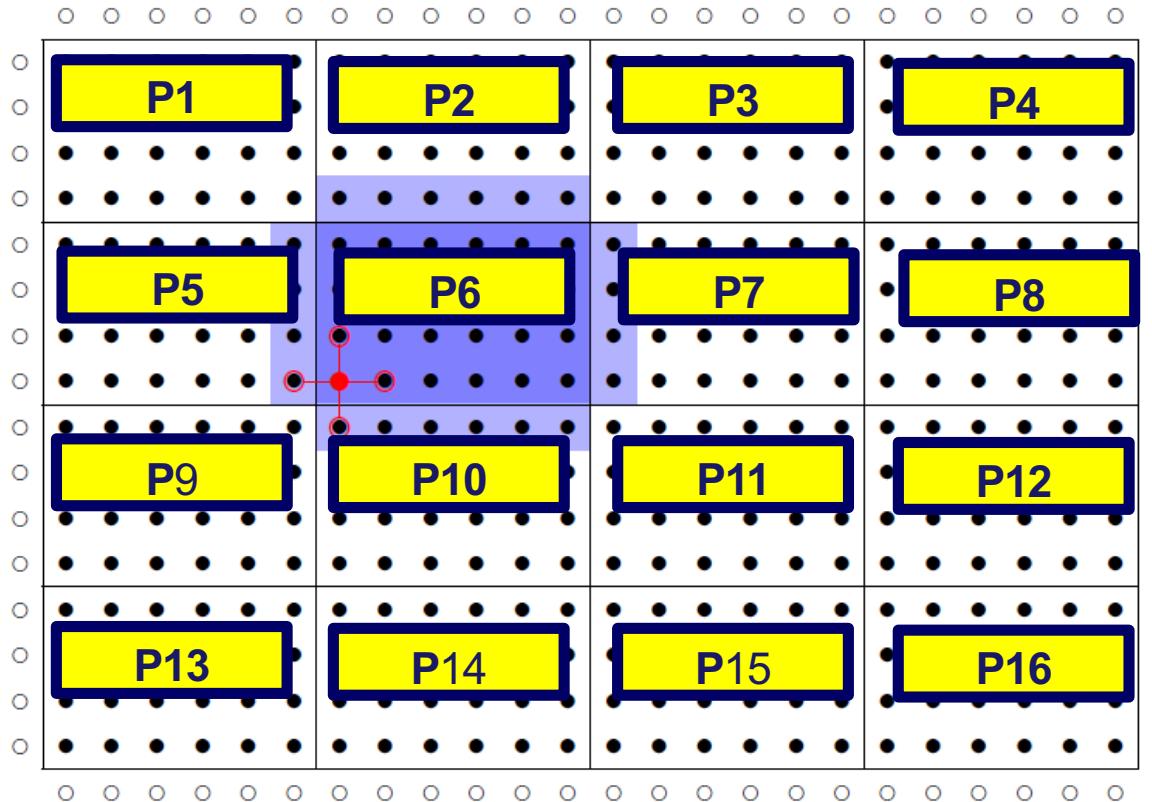
$$\left(\frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} \right)_j + \left(\frac{T_{j+1} - 2T_j + T_{j-1}}{\Delta y^2} \right)_i = 0$$



Parallelization

Example – Laplace equation

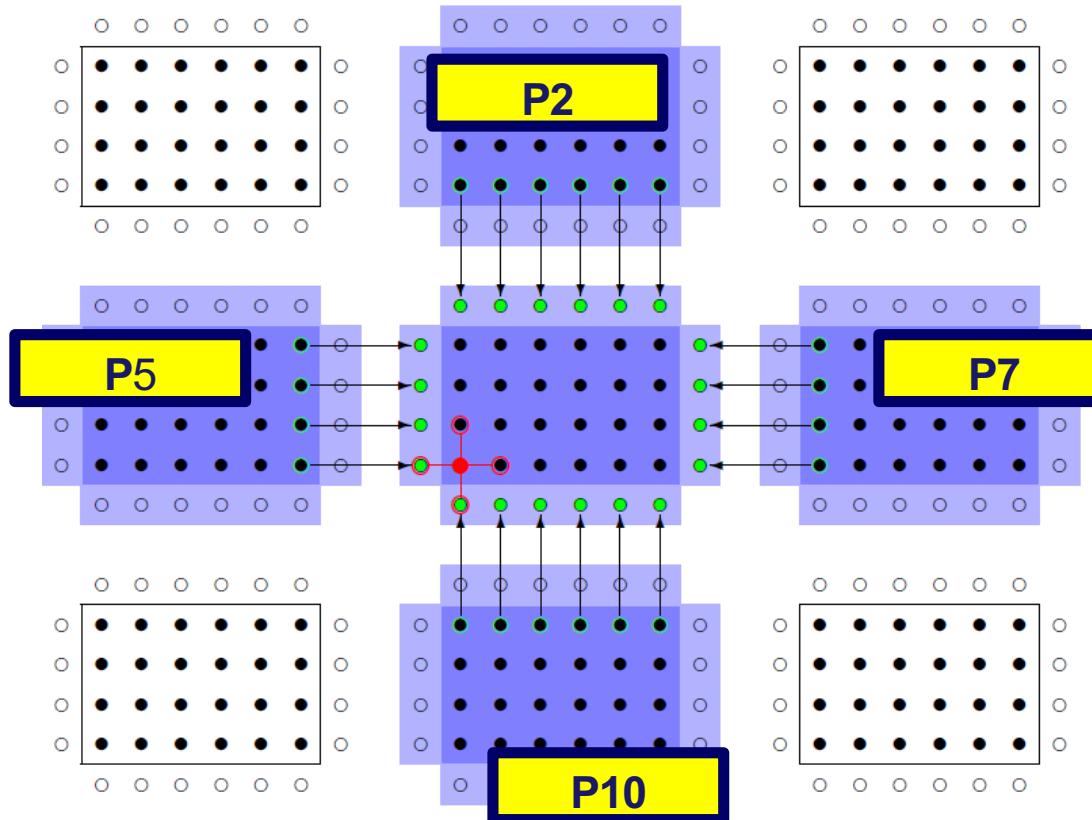
Communications needed when points are distributed over processes



Parallelization

Communication Overhead

Here the overhead would be large - 16 communicated points / 24 computing points per process

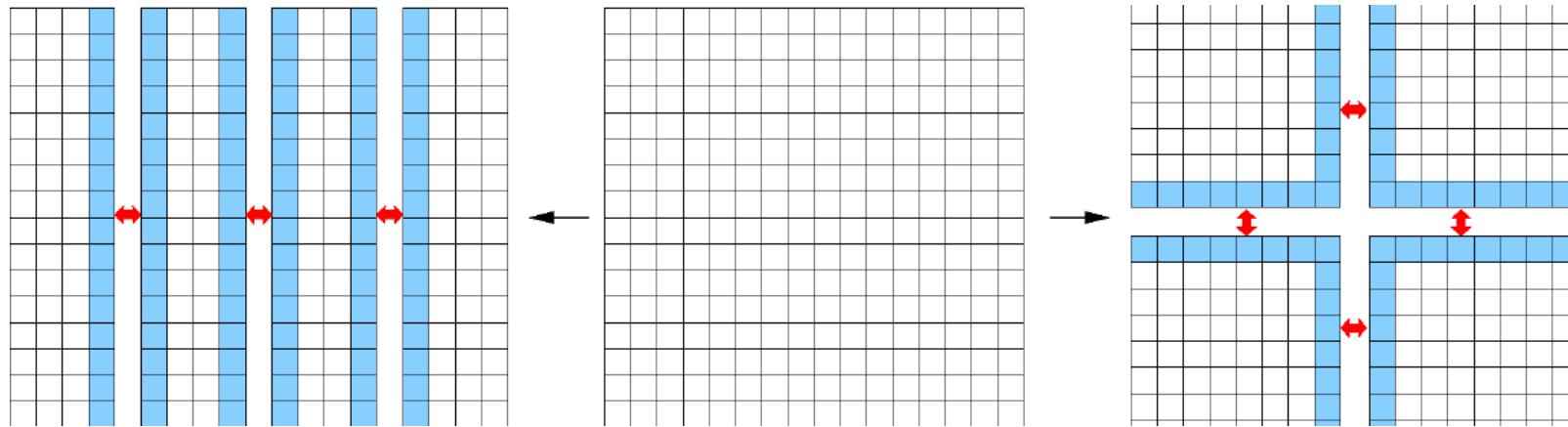


Parallelization

Communication Overhead

Example: 2D regular grid with $M \times M$ nodes

Single job time $T_{work} = a \cdot M^2$



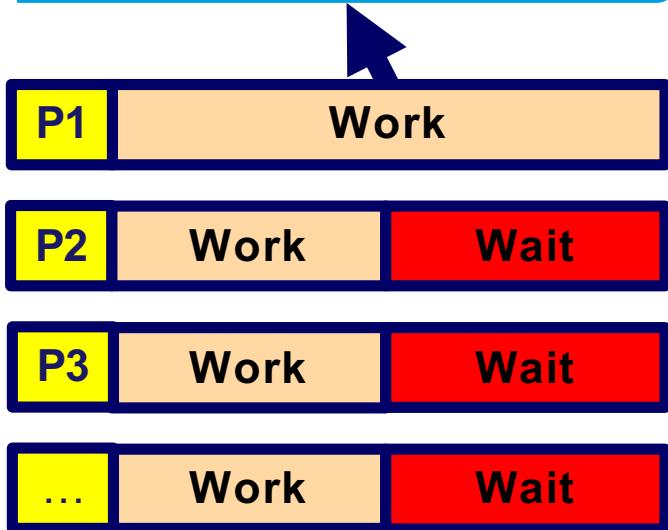
$$T_{comm_1}(N) = c \cdot 2 \cdot M$$

$$T_{comm_2}(N) = \frac{c \cdot 4M}{\sqrt{N}}$$

Parallelization

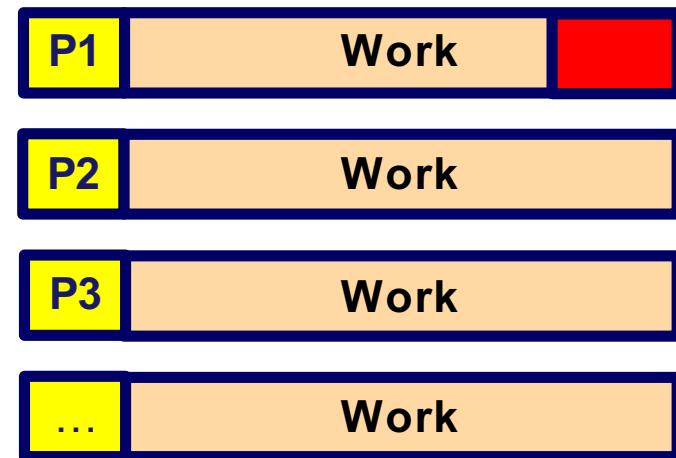
Load balance

Laggers waste a lot of resources



Work imbalance

Problem rebalancing may improve performance

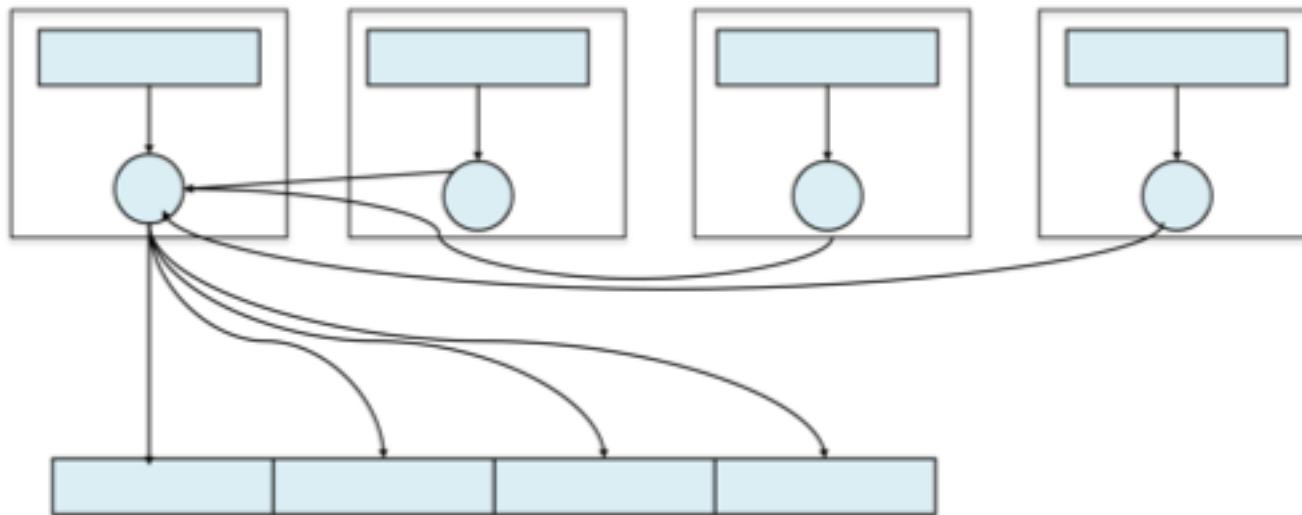


Work imbalance

Parallelization

I/O - Approaches

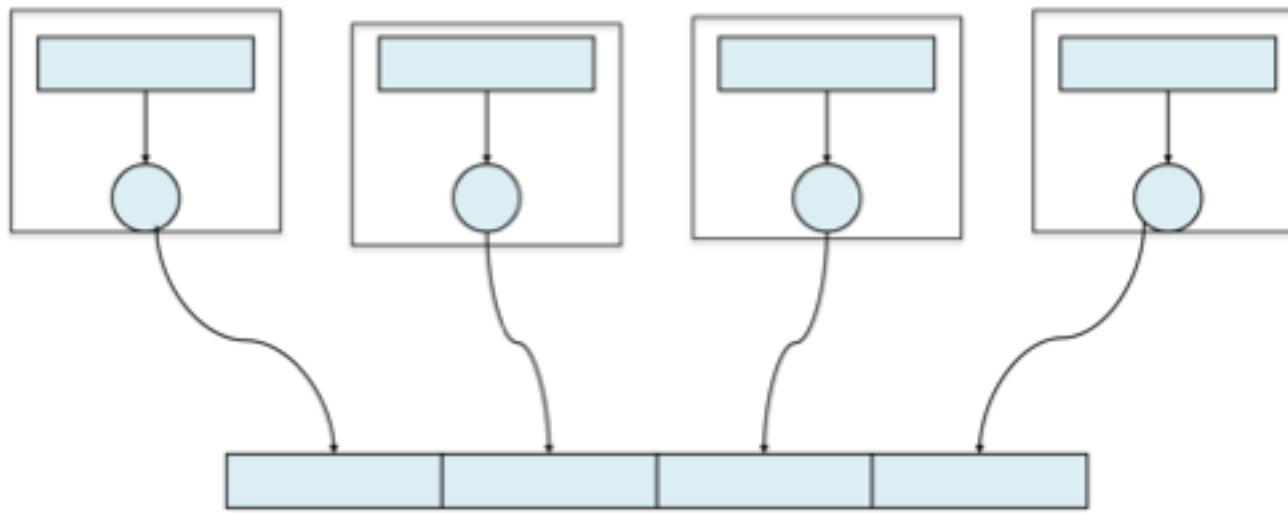
- One process performs I/O
 - data aggregation or duplication
 - I/O does not scale
 - Communication does not scale



Parallelization

I/O - Approaches

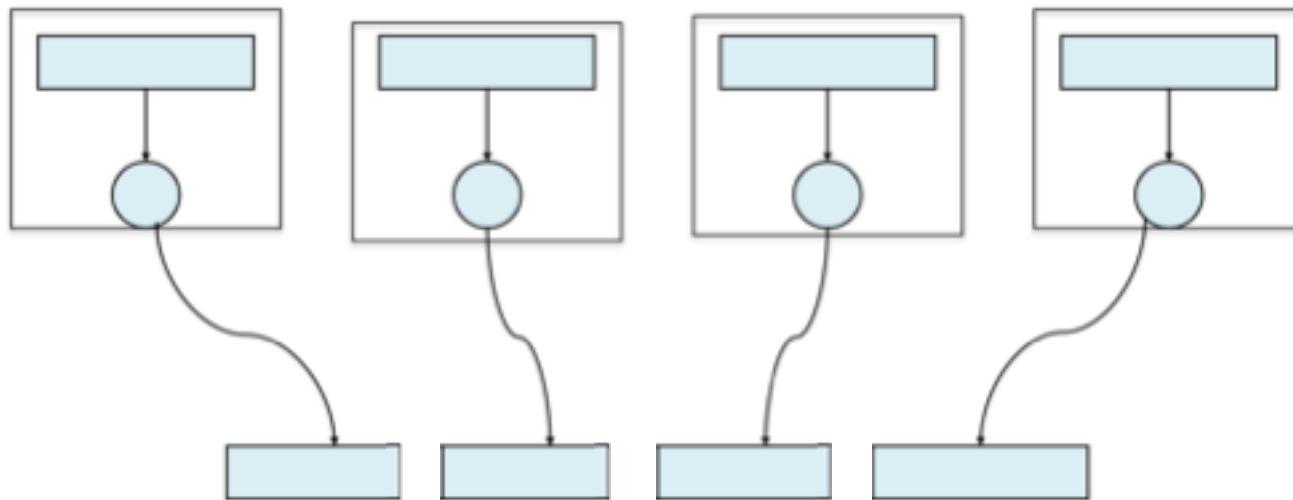
- Collective I/O to a single file
 - each process performs I/O to a single file which is shared.
 - data layout within the shared file is very important



Parallelization

I/O - Approaches

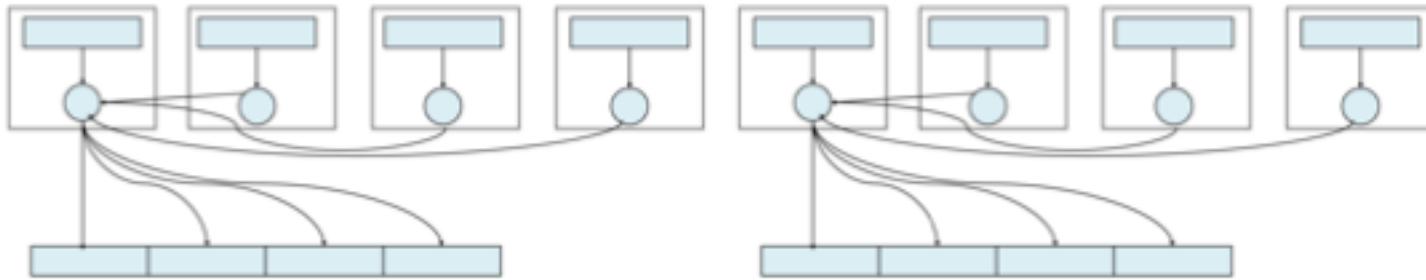
- All processes perform I/O to individual files
 - easy to program
 - enough for small/medium problems
 - number of files can be a bottleneck
 - number of simultaneous disk accesses creates contention for file system resources



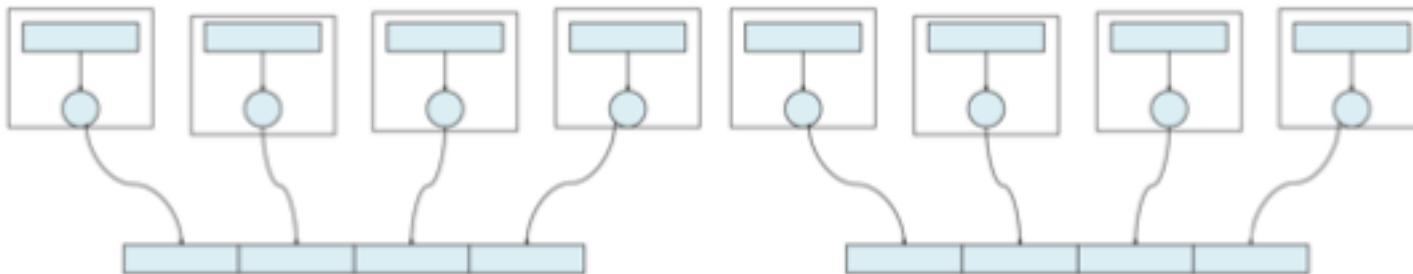
Parallelization

I/O - Approaches

- Hybrid
 - aggregation to a processor in a group which processes the data
 - group of processes perform parallel I/O to a file



OR



Parallelization

I/O – General Recommendations

- All STDIN, STDOUT, and STDERR I/O streams serialize your program
 - Disable debugging messages when running in production mode
 - “Hello from process 1200!”
 - “Process 128000, a=3.0”
- Reduce I/O
 - reduce number of saving points/checkpoints
 - process results during simulations (output integral quantities, not row data)
 - produce pictures instead of row data (coprocessing)
- Try to hide I/O (asynchronous I/O)
- Use parallel I/O
 - high-level libraries (HDF5)
 - MPI I/O

MPI – Non-blocking Communications

Deadlocks

- Using blocking communications can result in deadlocks

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int other = rank == 0?1:0;
MPI_Send(data, size, MPI_INT, other, 0, MPI_COMM_WORLD);
MPI_Recv(data, size, MPI_INT, other, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

Non-blocking Communications

- Motivation
 - Avoids deadlocks
 - Simplifies programming
 - Reduces synchronization
 - May allow to overlap communication and computation
- Requires additional request handle
 - Created for each nonblocking communication call
 - Used to check the communication state (wait/test operation)

MPI_Isend

```
int MPI_Isend(void* buffer, // message sending buffer
              int count,    // number of elements to send
              MPI_Datatype datatype, // element type
              int destination, // rank of destination process
              int tag,        // message tag
              MPI_Comm comm, // communicator
              MPI_Request *request); // request handle
```

- The function is non-blocking
 - Buffer cannot be reused after return
 - Request handle is unique for each operation
 - Test/wait operation must follow the non-blocking call

MPI_IReceive

```
int MPI_Ireceive(void* buffer, // message receive buffer
                 int count, // max. number of elements to receive
                 MPI_Datatype datatype, // element type
                 int source, // rank of source process
                 int tag, // message tag
                 MPI_Comm comm, // communicator
                 MPI_Request* request); // request handle
```

- The function is non-blocking
 - Message has not been received yet
 - No status argument
 - Test/wait operation must follow (status will be provided there)

MPI_Wait, Waitall

```
int MPI_Waitall(int count,           // arrays length
                MPI_Request *requests, // array of request handles
                MPI_Status *statuses); // array of status objects
```

- Blocks until all communication operations associated with active handles in the list complete
- Returns the statuses of all these operations

MPI – Non-blocking Communications

Deadlock

- /data/netapp/hpc-seminars/MultiNodeParallelization/5_deadlock_nonblock
 - check solution from the previous seminar
 - rewrite the program using non-blocking calls

Finished? <https://goo.gl/forms/DwuExWiMfkKlqvDm1>

MPI – Non-blocking Communications

Deadlock - solution

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

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int size = 100000;
    int* numbers_send = malloc(sizeof(int)*size);
    int* numbers_recv = malloc(sizeof(int)*size);
    int to, from;
    int other = rank == 0?1:0;
    MPI_Request requests[2];
    MPI_Status statuses[2];
    numbers_send[0]=rank;
    MPI_Isend(numbers_send, size, MPI_INT, other, 0, MPI_COMM_WORLD,&requests[0]);
    MPI_Irecv(numbers_recv, size, MPI_INT, other, 0, MPI_COMM_WORLD,&requests[1]);
    MPI_Waitall(2, requests, statuses);
    MPI_Finalize();
    printf("%d got %d from %d\n",rank,numbers_recv[0],other);
    free(numbers_send);
    free(numbers_recv);
    return 0;
}
```

MPI - Collective Communications

- Three types:
 - Synchronization (Barrier)
 - Data Movement (Scatter, Gather, Alltoall, Allgather)
 - Reductions (Reduce, Allreduce, Reduce_scatter)
- Usually blocking (MPI 3.0+ allows non-blocking)
- Receive buffers size must be exactly the size of the message
- Can be implemented with point-to-point calls
- Collective implementation is usually better optimized (tree-based algorithms, etc)

MPI_Barrier

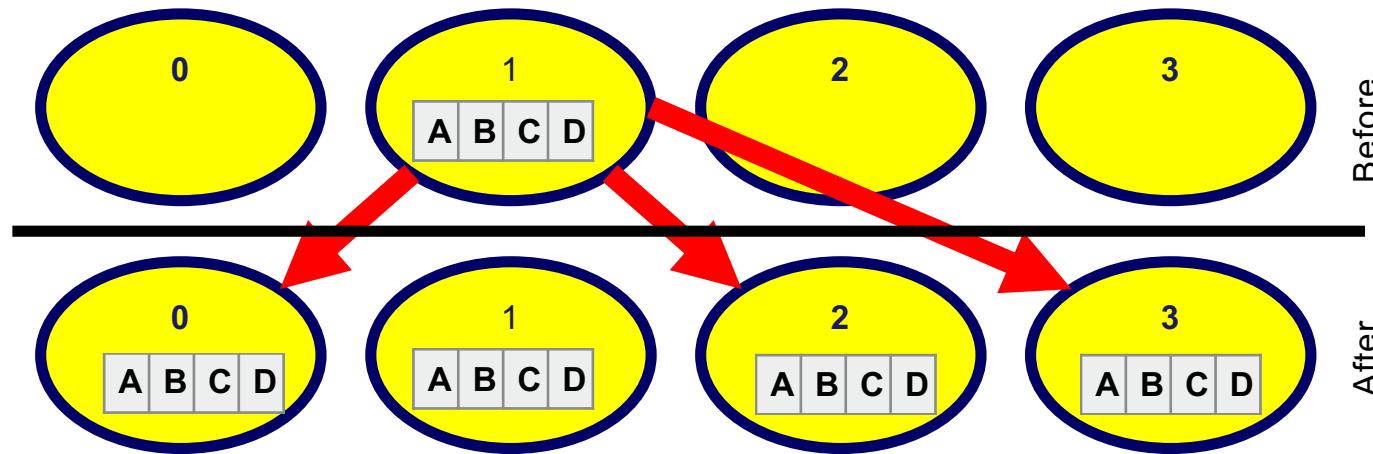
```
int MPI_Barrier(MPI_Comm comm);
```

- Explicit synchronization
- Block the process until all processes in the communicator called it
- Usually not needed
 - Synchronization is done implicitly by other communication calls
 - Can be used for debugging, profiling, etc.

MPI_Bcast

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,  
               MPI_Comm comm);
```

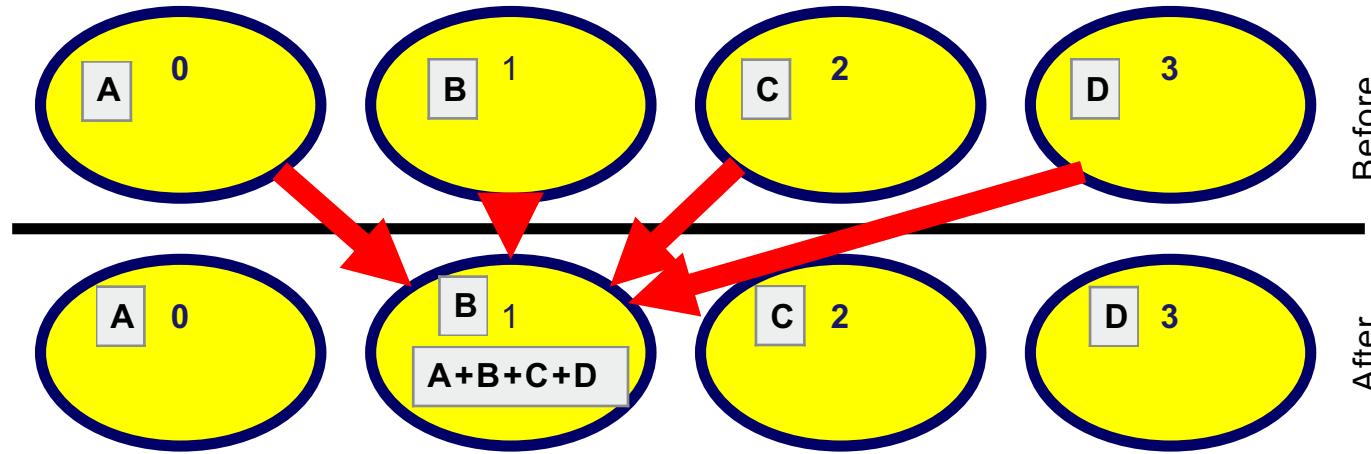
- One process (root) sends data to all others



MPI_Reduce

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,  
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

- MPI_Op (MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD, or user defined)



MPI

Integration in parallel

- /data/netapp/hpc-seminars/MultiNodeParallelization/6_integration_collective
 - check solution from the previous seminar
 - rewrite the program using MPI_Reduce

Finished? <https://goo.gl/forms/DwuExWiMfkKlqvDm1>

MPI

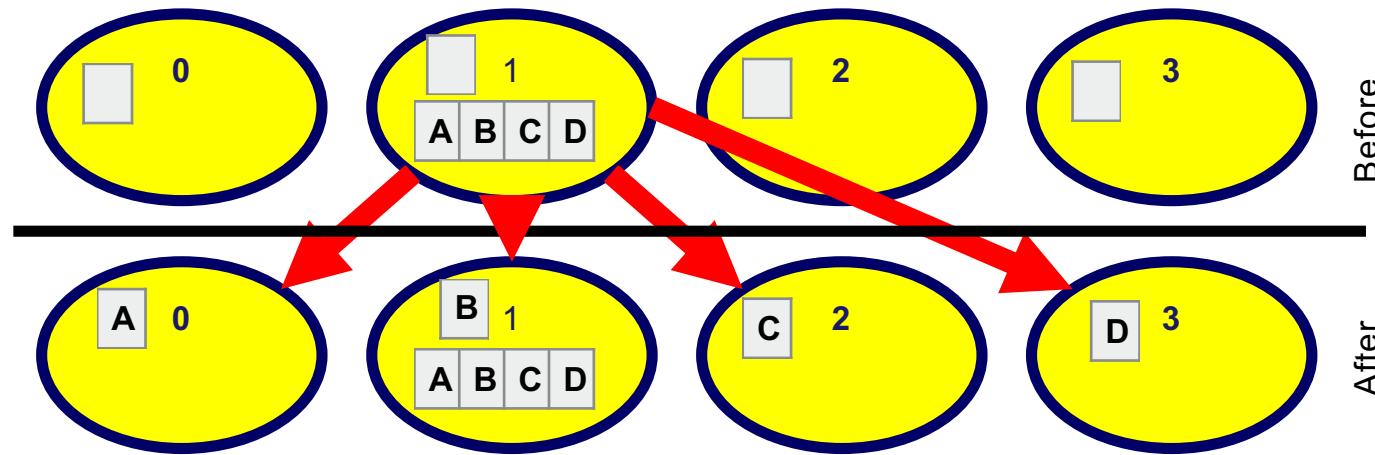
Integration in parallel - solution

```
#include <stdio.h>
#include <mpi.h>
double integral(double a,double b) {
    return (b-a)*10.0;
}
int main (int argc, char* argv[]) {
    int rank; int size;
    MPI_Status status;
    double a=0.0, b=2.0, res=0.0;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    double my_interval=(b-a)/size;
    double mya=a+rank*my_interval;
    double myb=mya+my_interval;
    double psum=integral(mya,myb);
    double start = MPI_Wtime();
MPI_Reduce(&psum,&res,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
    double end = MPI_Wtime();
    if (rank == 0) {
        printf("Integral = %f\n",res);
        printf("Elapsed: %f s\n",end-start);
    }
    MPI_Finalize();
}
```

MPI_Scatter

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm  
comm)
```

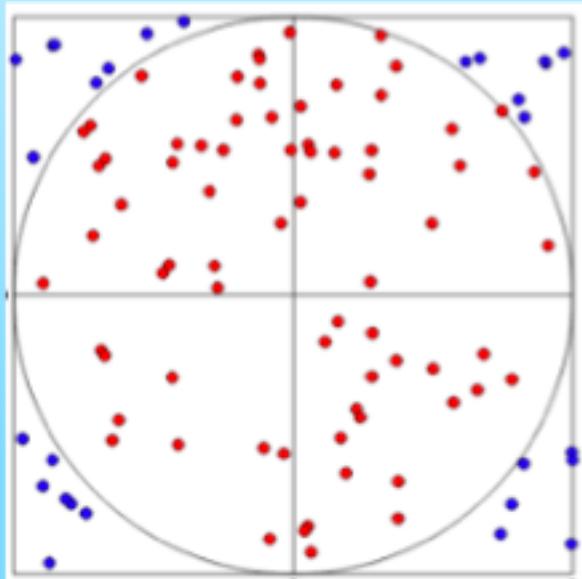
- One process scatters data to all others (including itself)



MPI – Collective Communications

Computation of Pi

- /data/netapp/hpc-seminars/MultiNodeParallelization/7_Pi
 - Create random points in process 0
 - send them to other processes (use MPI_Scatter)
 - compute part of Pi on every process, use MPI_Reduce to get Pi.



$$\pi = 4 \frac{A_{circle}}{A_{square}} = 4 \frac{N_{circle}}{N_{square}}$$

Finished? <https://goo.gl/forms/DwuExWiMfkKIqvDm1>

MPI – Collective Communications

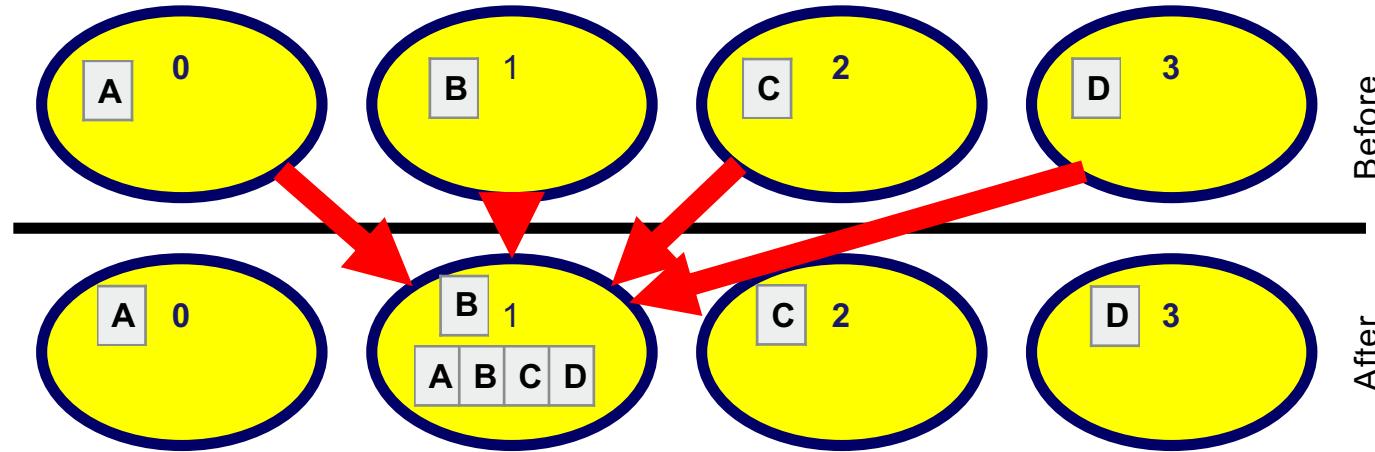
Computation of PI - Solution

```
int main (int argc, char* argv[]) {
MPI_Init(NULL, NULL);
int rank,size;int i;
MPI_Comm_rank(MPI_COMM_WORLD, &rank); MPI_Comm_size(MPI_COMM_WORLD, &size);
int npoints = strtol(argv[1],NULL,10);
int n_localpoints = npoints/size;
npoints = n_localpoints*size;
double* local_random_points=malloc(sizeof(double)*2*n_localpoints);
double* random_points;
if (rank == 0) random_points=CreateRandomPoints(npoints);
MPI_Scatter(random_points,n_localpoints*2,MPI_DOUBLE,local_random_points,n_local
points*2,MPI_DOUBLE,0,MPI_COMM_WORLD);
int local_count = 0;
for (i=0; i<=n_localpoints; ++i) {
double x = local_random_points[i];
double y = local_random_points[n_localpoints + i];
double r = ((x*x)+(y*y));
if (r<=1) local_count++;
int count;
MPI_Reduce(&local_count,&count,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
if (rank == 0) {
    double pi = 4.0*((double)count/npoints);
    printf("Pi: %f\n", pi);
    free (random_points);}
free (local_random_points);
MPI_Finalize();}
```

MPI_Gather

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,  
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm  
comm)
```

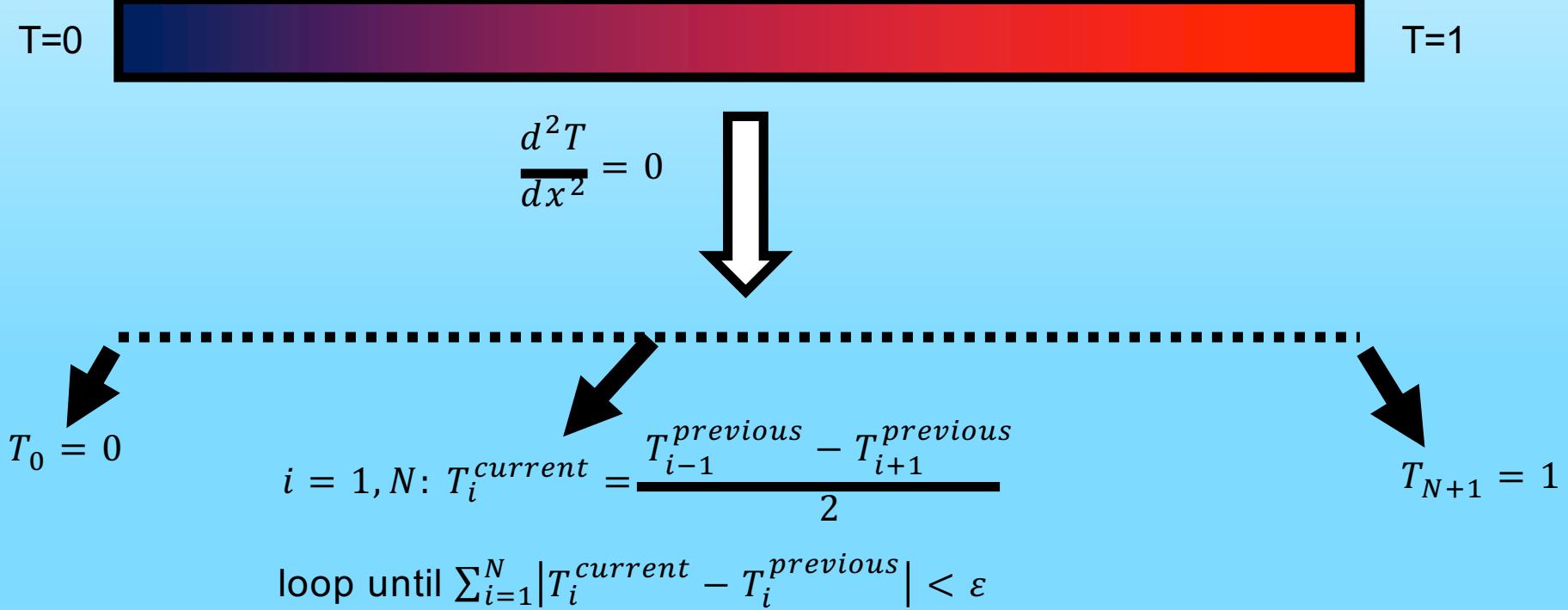
- One process gathers data from all others (including itself)



MPI – Collective Communications

Heat Equation

- /data/netapp/hpc-seminars/MultiNodeParallelization/8_heat
 - make it work in parallel, output from process 0 (use MPI_Scatter)



Finished? <https://goo.gl/forms/DwuExWiMfkKlqvDm1>

MPI – Collective Communications

Heat Equation - Solution

- /data/netapp/hpc-seminars/MultiNodeParallelization/8_heat/main_parallel.c

Summary

- Splitting a problem into multiple parts is most challenging part of the parallelization process
 - the rest is just applying appropriate MPI calls
 - embarrassingly parallel is easy, but it is not “real HPC”
 - domain decomposition is a most commonly used approach
- I/O might become a bottleneck, so it should be also parallelized
- Non-blocking communications may help to avoid deadlocks and be more efficient
- Use collective communications to distribute/collect data within many processes
 - if not all of the processes involved in communications, point-to-point pattern might be still more efficient