HPC Seminar

Parallelization on a Multiple Nodes – Part I

Sergey Yakubov - Maxwell Team - DESY IT
Hamburg, 11.06.2018

HELMHOLTZ ASSOCIATION

DESY

Agenda

- Take away from the OpenMP section
- MPI overview
- Basic functions
- Point-to-point communications
- Summary

Take away from previous seminars

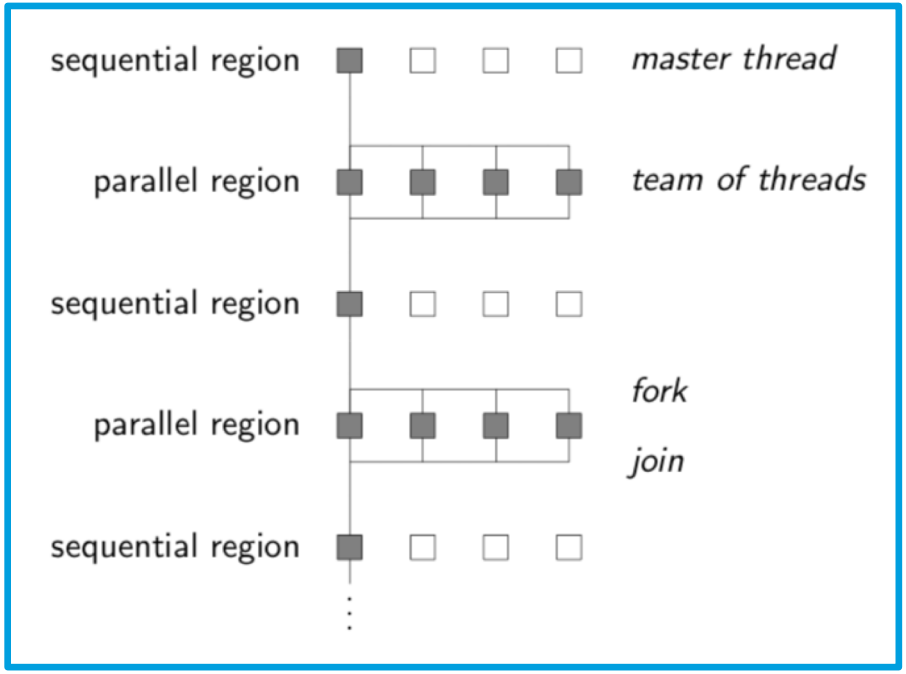
OpenMP

- OpenMP is the most recognised model for parallelization on shared memory architectures
  - Actively developed and supported by many vendors. Latest specifications - version 4.5, November 2015, working at OpenMP 5.0
  - Integrated in C/C++ and Fortran compilers
  - Portable
  - Hardly any alternatives (maybe openACC in the future)

Take away from previous seminars

OpenMP

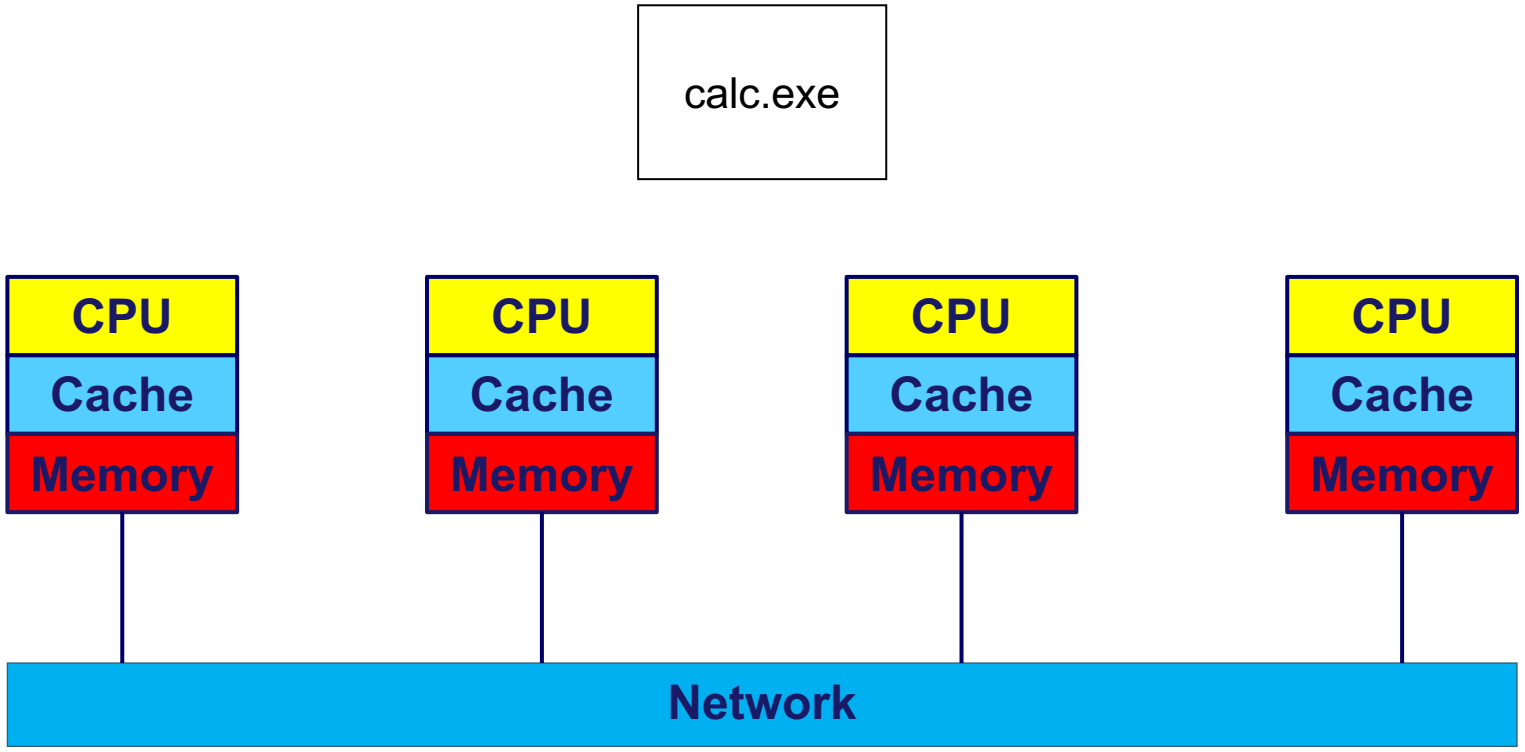
- Programming is based on compiler directives (pragmas)
  - Seen as comments/ignored for non-parallel program
  - Define regions of the code to be executed in parallel by threads
- Loop parallelization is a foundation stone of OpenMP
- Parallel performance is limited
  - Number of cores is limited
  - **Usually cannot go beyond a single node**



Distributed memory cluster

Parallelization approach

- We have a program calc.exe
  - want to do our simulations faster

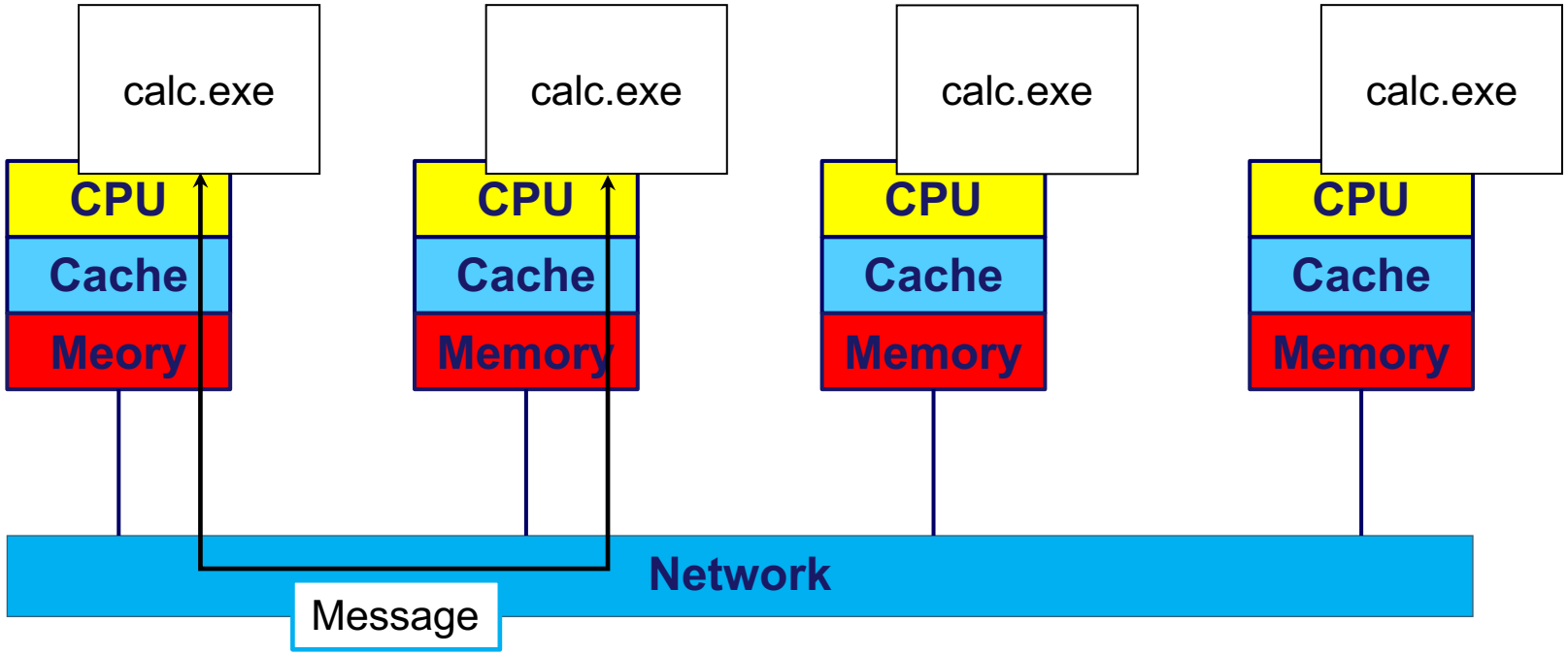


Distributed memory cluster

Parallelization approach

- we have to somehow split the job
- each nodes starts its own instance of the program
- instances exchange information via messages

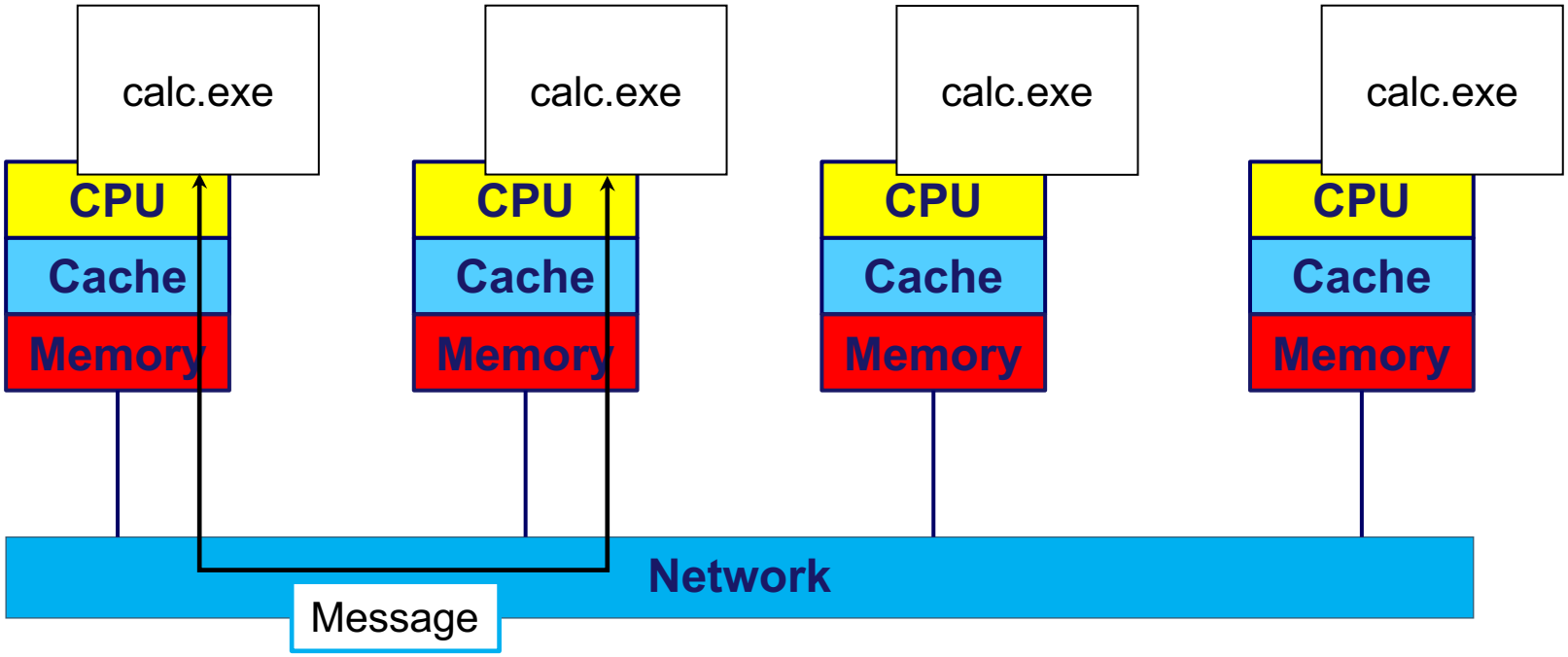
⇒ MPI



MPI

Overview

- Standardized interface to a communication library
- Hides hardware/software communication mechanisms from a user
- Implements internode communications via messages
- SPMD – each process starts same program. All variables are local for each process.



MPI

Overview

- Beginning (1992-1994) – introduction of the first message passing standard MPI 1.0
- MPI forum (www.mpi-forum.org) defines the standards
  - MPI-1 (1994) , MPI-1.3 (2008)
  - MPI-2 (1997), MPI2-2 (2009)
  - MPI-3 (2012), MPI-3.1 (2015), now working at 4.0
- Contributions from approx. 60 members
  - Academical organizations
  - HPC centres
  - Computer vendors

MPI

Overview

- Independent implementations
  - OpenMPI, MPICH
- Vendor‘s implementations
  - Cray, HP, Intel, Microsoft
- All implementations are portable
- Architecture and hardware independent
- Can also be used on shared memory machines
- Language support
  - Basic languages - C/C++ and Fortran
  - Bindings to Perl, Python, Java, etc.

MPI

Example

```
#include <stdio.h>
#include <mpi.h>
main (int argc, char* argv[]) {
    int myrank; int size;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello World. I'm process %d of %d
    processes.", myrank, size);
    MPI_Finalize();
    exit(0);
}
```

MPI

Compilation

- There are two ways to compile an MPI program
  - Using wrappers (mpicc, mpif77, mpif90)
    - $ mpicc prog.c
    - $ mpif90 prog.f90
  - Using regular compiler and setting compiler/linker flags
    - $ gcc prog.c –I<path to mpi.h> -L<path to libmpi> -lmpi
- CMake is as usual recommended
- On Maxwell many MPI implementations/versions and several compilers are installed – one should be careful

MPI

Execution

- Depending on MPI version execution command may differ
  - Standard commands
    - $ mpirun –np 4 prog_name
    - $ mpiexec -n 4 prog_name
  - Other versions
    - $ poe -procs 4 prog_name # IBM
    - $ aprun -n 4 prog_name # Cray
- See man pages for more info
- On Maxwell we use SLURM
  - one can omit number of processes

MPI

First Program

- Let's have a look at the first example:

/data/netapp/hpc-seminars/MultiNodeParallelization/1_hello_world

  - copy to your folder
  - compile

        cmake .

        make
  - run it via SLURM – sbatch job.sh
  - check output
  - play with SLURM parameters and see what changes
    - --nodes, --tasks, --cpus-per-task
  - Change the program to print "Bye" at the end.
    - should be printed only once

Finished? https://goo.gl/forms/Drru18bf33aqcq1q2

MPI

First Program - Solution

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char* argv[]) {
    int myrank; int size;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello World. I'm process %d of %d
    processes.\n", myrank, size);
    if (myrank == 0) printf("Bye\n");
    MPI_Finalize();
}
```

MPI

First Program - Solution

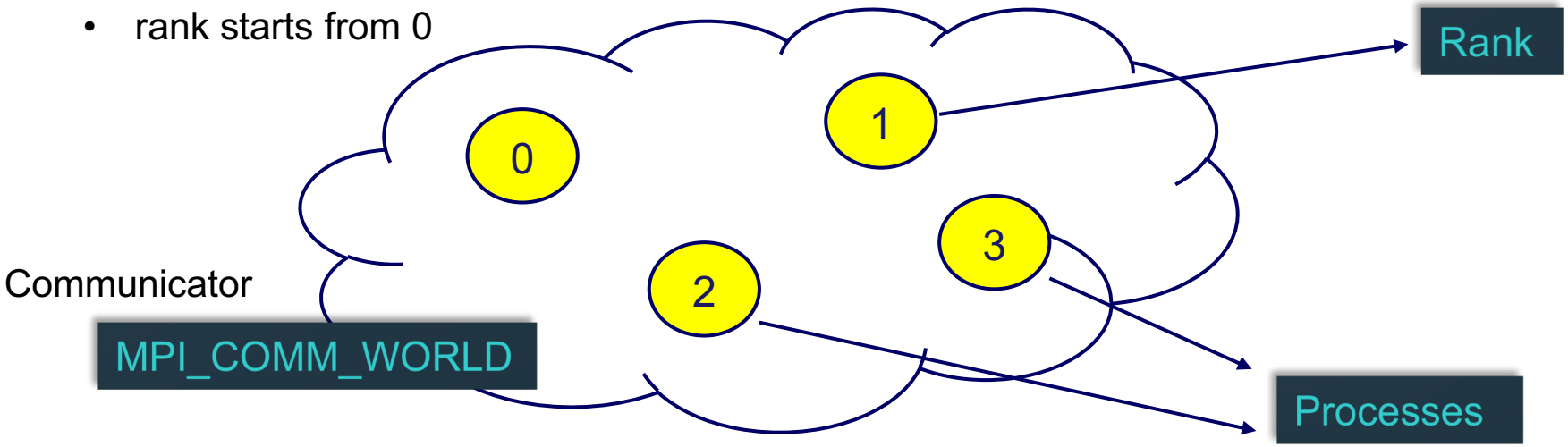
```
#!/bin/bash
##SBATCH --ntasks=2
#SBATCH --nodes=1
##SBATCH --cpus-per-task=8
#SBATCH --partition=all
#SBATCH -t 00:01:00


mpirun ./hello
```

MPI

Basic functions

- int MPI_Init(int *argc, char ***argv)
  - Initialization of parallel program
  - Must be first MPI call, must be called only once
- int MPI_Finalize()
  - Clean shut-down of parallel program
  - Must be last MPI call, must be called by each proces
- int MPI_Comm_size(MPI_Comm comm, int *size);
- int MPI_Comm_rank(MPI_Comm comm, int *rank);
  - rank starts from 0



MPI

Basic functions

- int MPI_Abort(MPI_Comm comm, int errorcode)
  - correctly terminates all processes of group *comm*
  - *errorcode* will be returned
  - no MPI_Finalize() is needed after MPI_Abort(MPI_COMM_WORLD)
- double MPI_Wtime();
  - Elapsed wallclock time. Call two times (before, after), calculate difference.
- int MPI_Barrier(MPI_Comm comm)
  - Blocks the caller until all processes in the communicator have called it

MPI

Parallelization concept

- Second example:

/data/netapp/hpc-seminars/MultiNodeParallelization/2_parallelization_concept

  - copy to your folder
  - change it so that
    - process 0 prints “Hello, I’m the master here!”
    - process 1 prints “Hello, I’m the worker!”
    - other processes do nothing
  - add measuring time and print it at the end of the program
  - run it via SLURM – sbatch job.sh

Finished? https://goo.gl/forms/Drru18bf33aqcq1q2

MPI

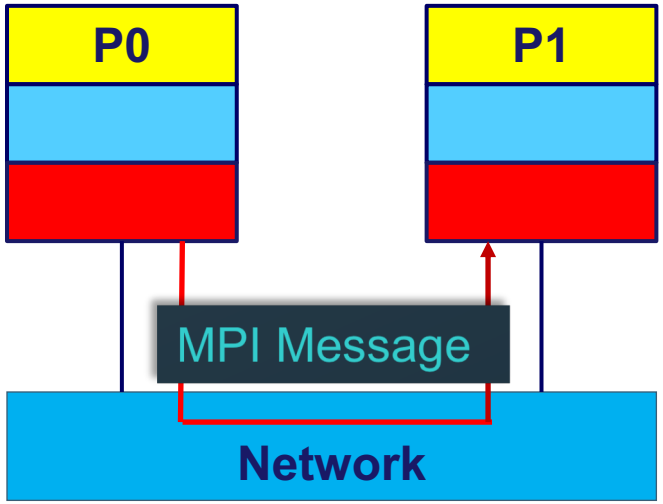
Parallelization concept - Solution

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char* argv[]) {
    int myrank; int size;
    MPI_Init(NULL, NULL);
    double start = MPI_Wtime();
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0) printf("I'm the master here")
    else if (rank == 1) printf("I'm the worker  here");
    MPI_Barrier(MPI_COMM_WORLD);
    double end = MPI_Wtime()
    if (myrank == 0) printf("Elapsed: %f s\n",end-start);
    MPI_Finalize();
}
```

MPI

Point-to-point communications

- MPI implements interprocess communications via messages
  - Who receives the message
  - Data type of the message
  - Message size
  - Address of the message to send
  - Message Tag
  - Communicator
  - Who sent the message
  - Where to store the received message
  - Maximum size of the received message



MPI

Point-to-point communications

```
int MPI_Send(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
```

- The function is blocking
  - Buffer may be reused after return
  - That doesn‘t mean that message was recieved!
  - May block until the message is received by the destination process – implementation/message size depending

MPI

Point-to-point communications

```
MPI_Receive(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_Status* status); // receive status
```

- The function is blocking
  - After return message has been successfully received
  - Buffer size can be larger than message size
  - Number of elements in the message can be less than count

MPI

Point-to-point communications

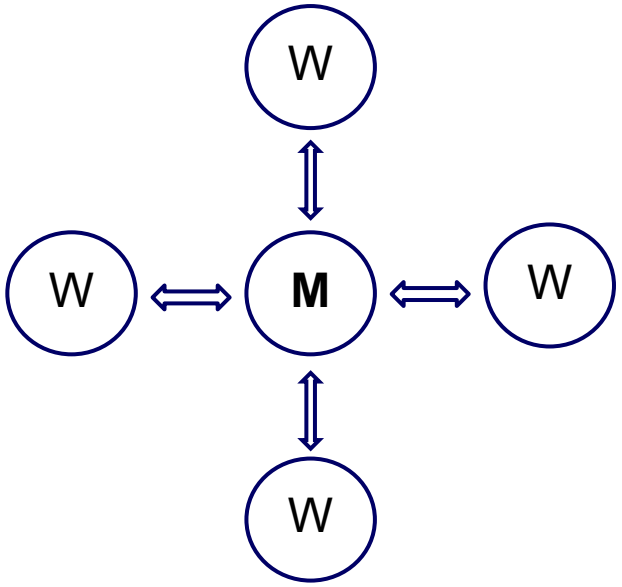
MPI_Receive(buffer, count, datatype, source, tag, comm, status)

- Source can be MPI_ANY_SOURCE
- Tag can be MPI_ANY_TAG
- MPI_Status* status (integer status(MPI_STATUS_SIZE)) has info
  - status(MPI_SOURCE) – rank of source process
  - status(MPI_TAG) – tag of the message
  - can be MPI_STATUS_IGNORE
- Number of received elements can be obtained via
  - MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype, int *count)

MPI

Master-Worker Approach

- Classical approach to parallel programming
  - One process is a master
  - The other processes are workers
  - Master collects results from workers
- Uses only MPI_SEND and MPI_RECEIVE
- Point-to-point communication pattern



MPI

Integration in parallel

```
integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
a=0.d0 ; b=2.d0 ; res=0.d0 ; f_x = 10.d0;
mya = a + rank*(b-a)/size
myb = mya + (b-a)/size
psum = (myb-mya)*h
if(rank.eq.0) then
 res=psum
 do i=1,size-1
  call MPI_Recv(tmp,1,MPI_DOUBLE_PRECISION,i,
0,MPI_COMM_WORLD,status,ierror)
  res=res+tmp
 enddo
 write(*,*) 'Result: ',res
else
 call MPI_Send(psum,1,MPI_DOUBLE_PRECISION,0,0,
MPI_COMM_WORLD,ierror)
endif
```

$$\int_a^b f(x)\,dx$$

do it in C, measure execution time

Finished? https://goo.gl/forms/Drru18bf33aqcq1q2

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; double b=2.0; double 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();
    if (rank == 0) {
         res = psum;
         for (int i=1;i<size;i++) {
          MPI_Recv(&psum,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
          res+=psum;
         }
         printf("Integral = %f\n",res);
    } else
      MPI_Send(&psum,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    double end = MPI_Wtime();
    if (rank == 0) printf("Elapsed: %f s\n",end-start);
    MPI_Finalize();
}
```

MPI

Deadlocks

- As with OpenMP, synchronization of the processes can result in deadlock

```
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);
```

MPI

Deadlock

- /data/netapp/hpc-seminars/MultiNodeParallelization/4_deadlock
  - copy to your folder, compile, run (without SLURM or via an interactive job)
  - do you get deadlock?
  - correct the program

Finished? https://goo.gl/forms/Drru18bf33aqcq1q2

MPI

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;
    numbers_send[0]=rank;
    if (rank == 0) {
        MPI_Send(numbers_send, size, MPI_INT, other, 0, MPI_COMM_WORLD);
        MPI_Recv(numbers_recv, size, MPI_INT, other, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else {
        MPI_Recv(numbers_recv, size, MPI_INT, other, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(numbers_send, size, MPI_INT, other, 0, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    printf("%d got %d from %d\n",rank,numbers_recv[0],other);
    free(numbers_send);
    free(numbers_recv);
    return 0;
}
```

Summary

- 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