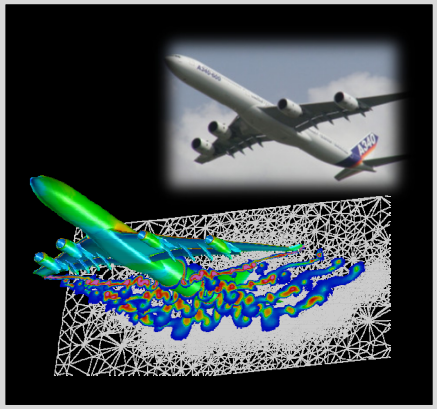


High Performance Computing

ADVANCED SCIENTIFIC COMPUTING

Prof. Dr. – Ing. Morris Riedel
Adjunct Associated Professor
School of Engineering and Natural Sciences, University of Iceland, Reykjavik, Iceland
Research Group Leader, Juelich Supercomputing Centre, Forschungszentrum Juelich, Germany

@Morris Riedel @MorrisRiedel @MorrisRiedel

PRACTICAL LECTURE 3.1

Understanding MPI Messages & Collectives

September 16, 2019
Webinar

UNIVERSITY OF ICELAND
SCHOOL OF ENGINEERING AND NATURAL SCIENCES
FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL ENGINEERING AND COMPUTER SCIENCE

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

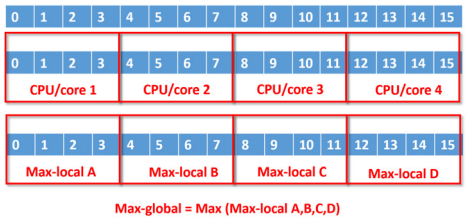
DEEP Projects

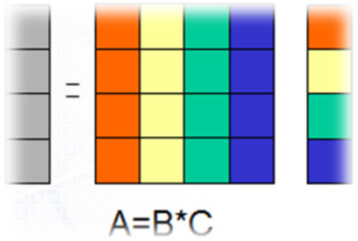
HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

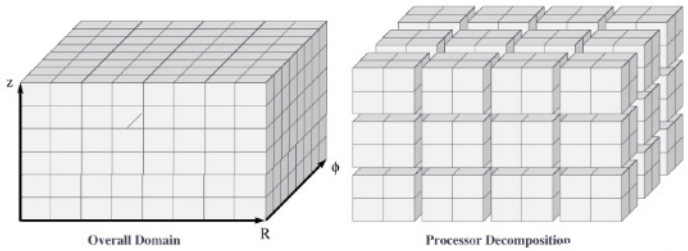
HAICU HELMHOLTZ ARTIFICIAL INTELLIGENCE COOPERATION UNIT

Review of Lecture 3 – Parallelization Fundamentals

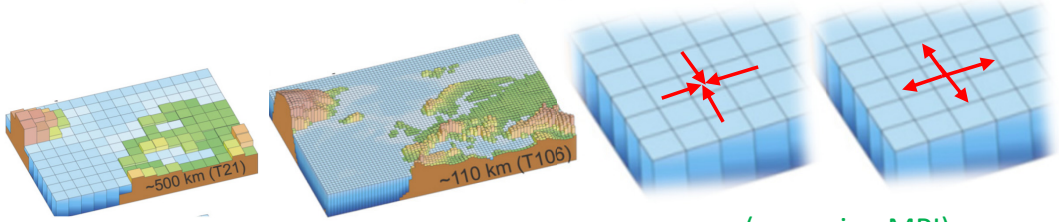
- Parallelization & Domain Decomposition





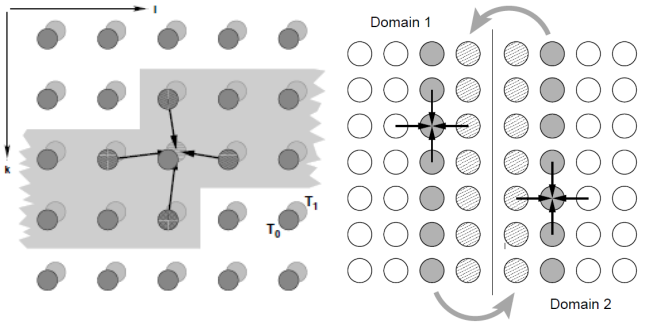


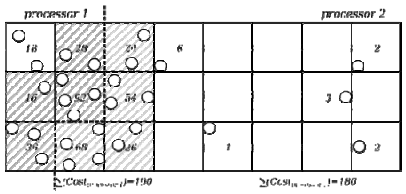
(not always mapping 1 CPU/core to 1 decomposed domain tile)



(e.g. using MPI)

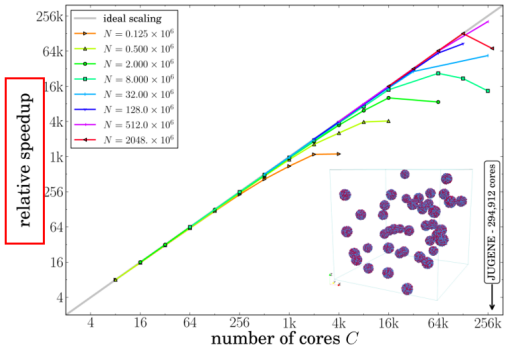
- Halo/Ghost Layers/Cells & Load Imbalance

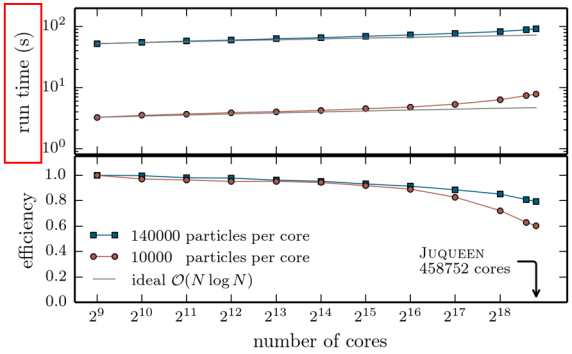




[5] M. Goetz & M. Riedel et al, Proceedings IEEE SC 2015

- Speed-Up & Strong/Weak Scalability Metrics





[1] 2013 SMU HPC Summer Workshop *[2] Parallel Computing Tutorial* *[3] Introduction to High Performance Computing for Scientists and Engineers* *[4] PEPC Webpage*

Outline of the Course

1. High Performance Computing
2. Parallel Programming with MPI
3. Parallelization Fundamentals
4. Advanced MPI Techniques
5. Parallel Algorithms & Data Structures
6. Parallel Programming with OpenMP
7. Graphical Processing Units (GPUs)
8. Parallel & Scalable Machine & Deep Learning
9. Debugging & Profiling & Performance Toolsets
10. Hybrid Programming & Patterns
11. Scientific Visualization & Scalable Infrastructures
12. Terrestrial Systems & Climate
13. Systems Biology & Bioinformatics
14. Molecular Systems & Libraries
15. Computational Fluid Dynamics & Finite Elements
16. Epilogue

+ additional practical lectures & Webinars for our hands-on assignments in context

- Practical Topics
- Theoretical / Conceptual Topics

Outline

- Programming & Compiling C-based MPI Programs
  - Distributed Memory & Parallel Programming – Revisited
  - Step-Wise Walkthrough for Programming a Simple C & MPI Program
  - Parallel Environment & Message Passing with MPI – Revisited
  - Simple Application Example with MPI Send/Receive
  - Fine-grained Job Script Request & Allocation of Compute Resources
- Understanding MPI Collectives & Message Exchange Options
  - MPI Collective Functions – Revisited
  - Simple Application Example with MPI Broadcast
  - Differences between MPI Point-to-Point vs. Collective Operations
  - Exploring the Walltime – What happens if a job runs against the wall?
  - Simple Hellosleep.c Example to understand Walltime

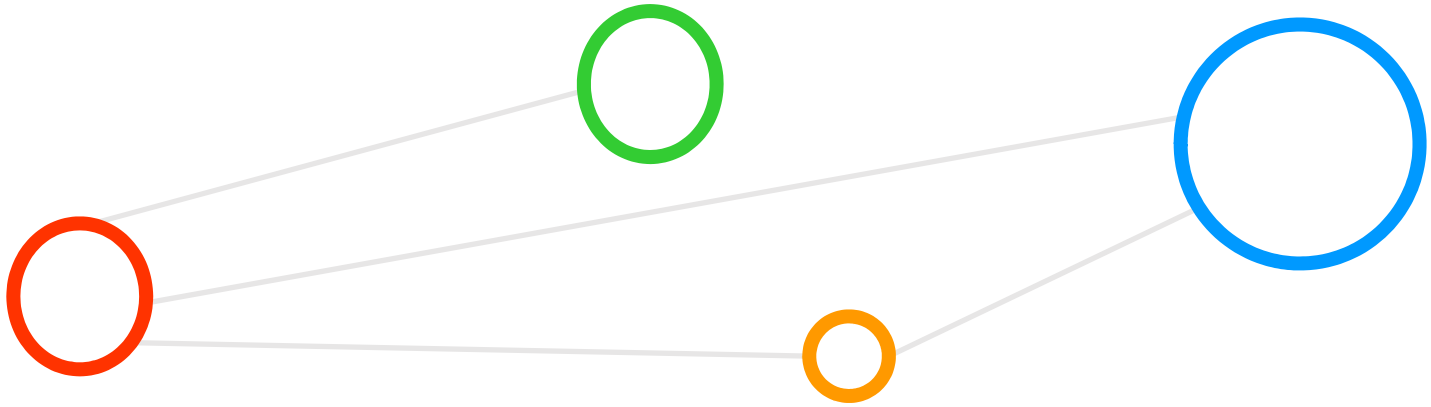
- **This lecture is not considered to be a full introduction to MPI programming and the overall MPI functions library and rather focusses on selected commands and concepts particularly relevant for our assignments, e.g. simple MPI send/receive and selected MPI collective functions**
- **The goal of this practical lecture is to make course participants aware of the process of compiling simple C & MPI programs and the use of MPI message exchanges that enable many scientific & engineering applications in data sciences & simulation sciences today**

Selected Learning Outcomes – Revisited

- Students understand…
  - Latest developments in parallel processing & high performance computing (HPC)
  - How to create and use high-performance clusters
  - What are scalable networks & data-intensive workloads
  - The importance of domain decomposition
  - Complex aspects of parallel programming → e.g., scheduling(!)
  - HPC environment tools that support programming or analyze behaviour
  - Different abstractions of parallel computing on various levels
  - Foundations and approaches of scientific domain-specific applications

- Students are able to …
  - Programm and use HPC programming paradigms
  - Take advantage of innovative scientific computing simulations & technology
  - Work with technologies and tools to handle parallelism complexity

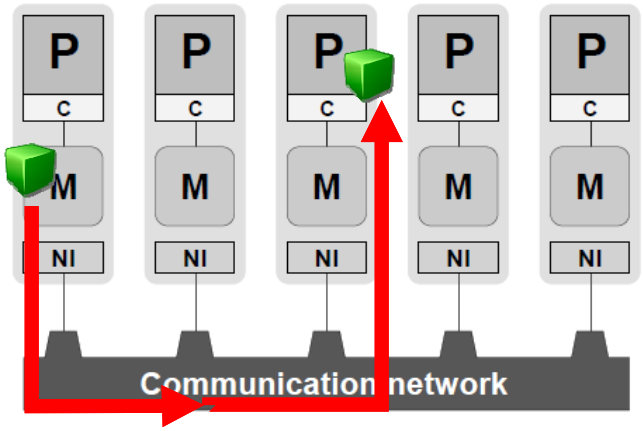
Programming & Compiling C-based MPI Programs

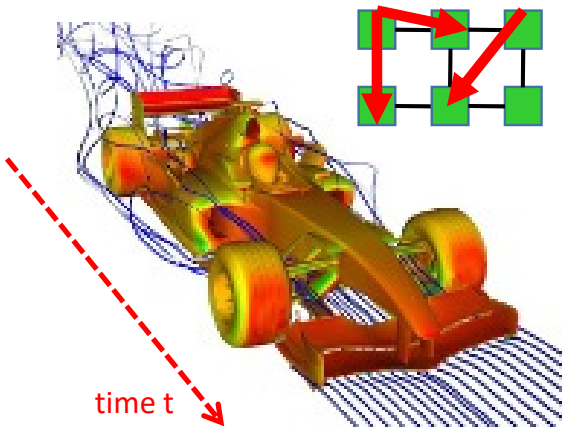


Distributed-Memory Computers – Revisited (cf. Lecture 1)

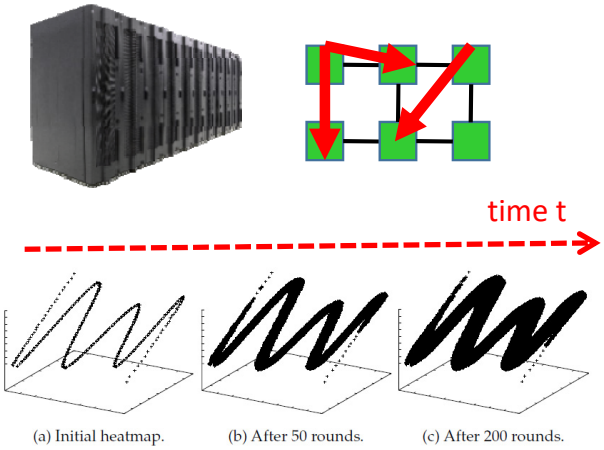
- **A distributed-memory parallel computer establishes a 'system view' where no process can access another process' memory directly**

dominant programming model Message Passing Interface (MPI)





[8] Modified from Caterham F1 team

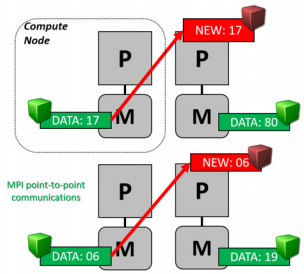


[3] Introduction to High Performance Computing for Scientists and Engineers

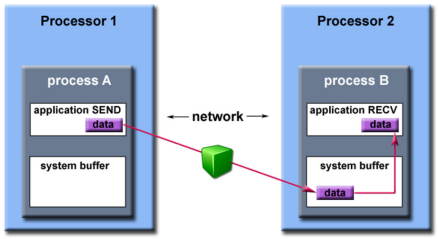
- Features
  - Processors communicate via Network Interfaces (NI)
  - NI mediates the connection to a Communication network
  - This setup is rarely used → a programming model view today

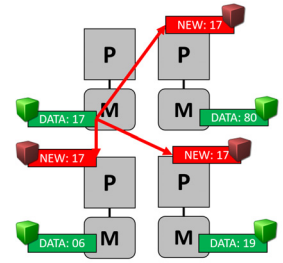
Parallel Programming with MPI & Basic Building Blocks (cf. Lecture 2)

Message Passing Interface (MPI) Concepts

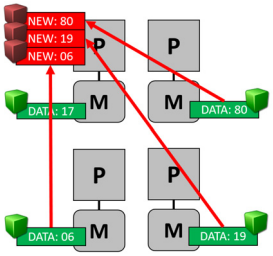


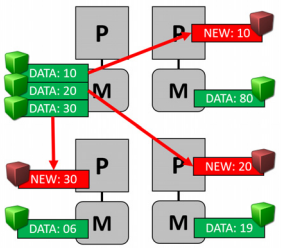
MPI Point to Point Communication

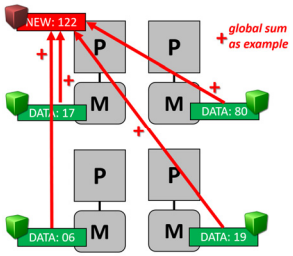




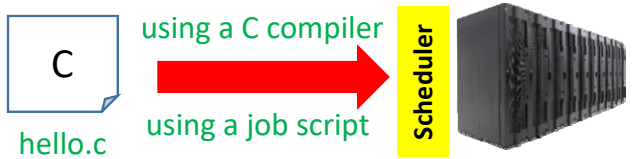
MPI Collective Communication

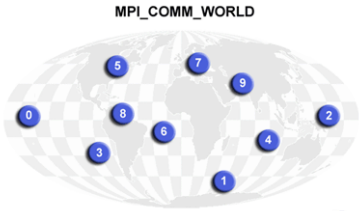


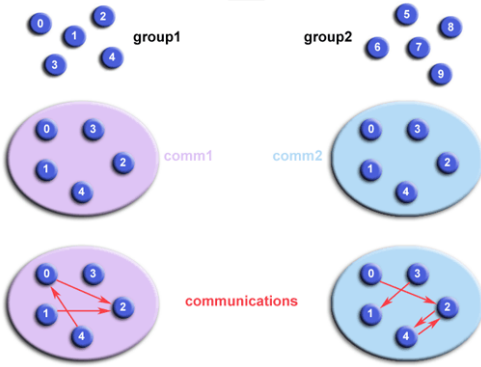




MPI Parallel Programming Basics



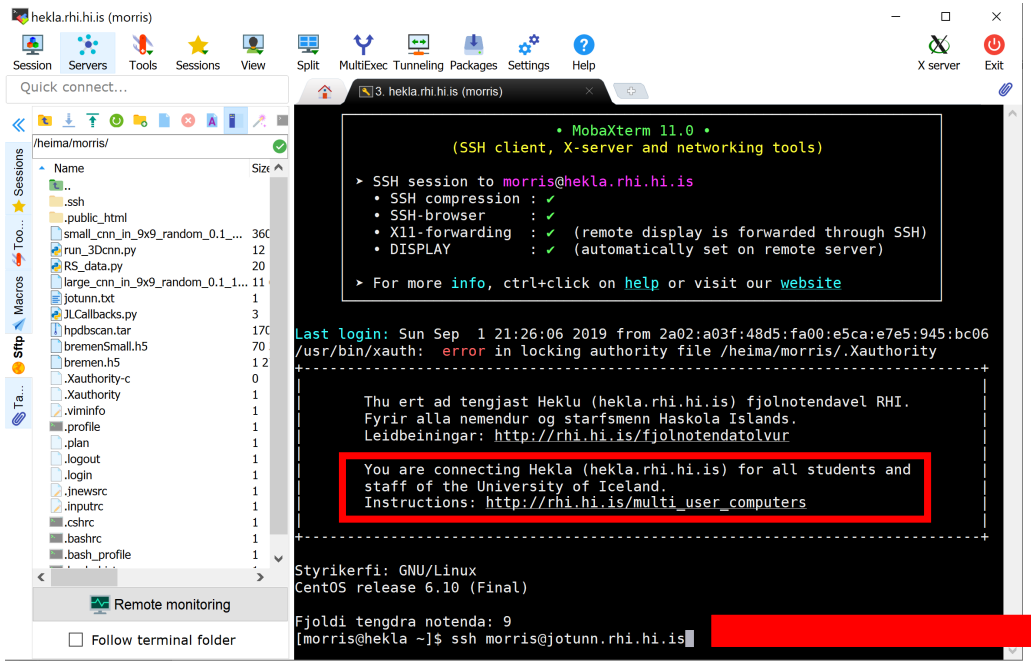





```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello World, I am %d out of %d\n",
         rank, size);
  MPI_Finalize();
  return 0;
}
```

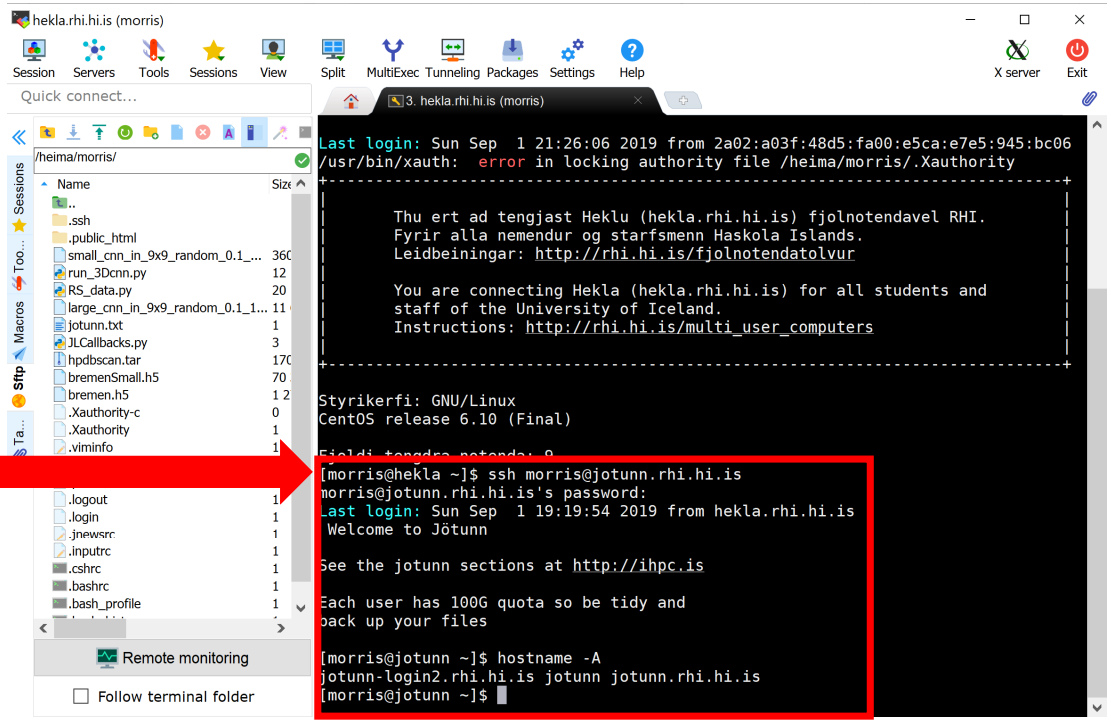
[6] LLNL MPI Tutorial

Step 1: SSH Access to HPC System – Jötunn HPC System Example



Hekla System

Jötunn HPC System

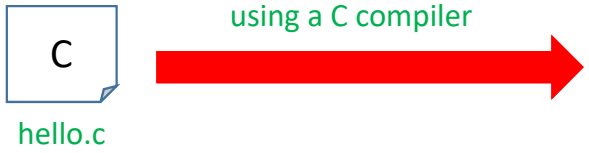


Step 2 & 3: Edit a Text File – (MPI) Basic Building Blocks: Variables & Output

```
#include <stdio.h>

int main(int argc, char** argv)
{
  int rank, size;
  printf("Hello World, I am %d out of %d\n",
         rank, size);

  return 0;
}
```

- The main function is 'called' by the operating system when a user runs the C program – but essentially a usual c function with optional parameters that we added here to be used later in the initialization of the MPI environment
- Two integer variables that are later useful for working with specific data obtained from the specific MPI library that we need to add in the next step too in order to fill information into the integer variables about rank and sizes
- The printf() function sends formatted text as output to stdout and is often used for simple debugging of C programs
- Thinking in parallel in parallel programming is to understand that different processes have an identity and work on different elements of the program
- In the example we want to give an output that shows the identity of each MPI process by using the rank and size information

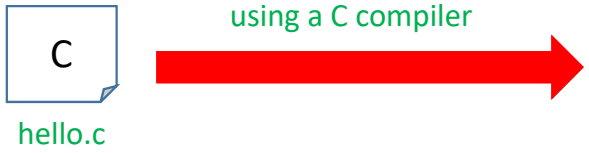
- Extended Simple C Program (still C only)
  - Above file content is stored in file hello.c
  - Selected changes to the basic c program structure to prepare for MPI
  - hello.c is not executable as C programm → it needs a compilation



Step 4: Edit a Text File – MPI Basic Building Blocks: Header & Init/Finalize

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

int main(int argc, char** argv)
{
  int rank, size;
  MPI_Init(&argc, &argv);
  printf("Hello World, I am %d out of %d\n",
         rank, size);
  MPI_Finalize();
  return 0;
}
```

- Libraries can be used by including C header files, here the library for MPI is included in order to use several MPI functions in our extended C program
- The MPI_INIT() function initializes the MPI environment and can take inputs via the main() function arguments
- MPI_Finalize() shuts down the MPI environment
- After MPI_Finalize() no parallel execution of the code can take place)

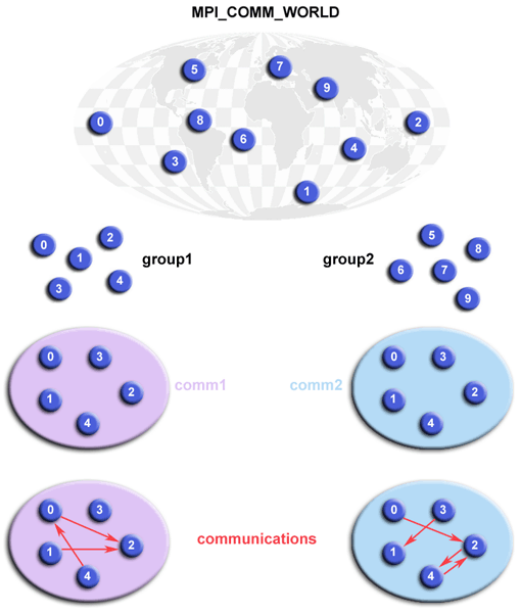


- Extended Simple C Program
  - hello.c is not executable as C programm → it needs a compilation

Step 4: Edit a Text File – MPI Basic Building Blocks: Rank & Size Variables

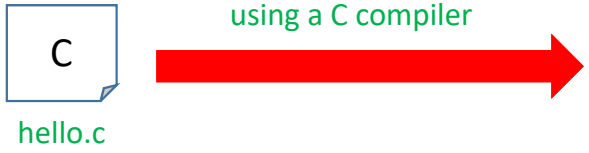
```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv)
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello World, I am %d out of %d\n",
         rank, size);
  MPI_Finalize();
  return 0;
}
```

- **The MPI_Comm_size() function determines the overall number of n processes in the parallel program: stores it in variable size**
- **The MPI_Comm_rank() function determines the unique identifier for each processor: stores it in variable rank with values (0 … n-1)**
- **MPI_COMM_WORLD communicator constant denotes the 'region of communication', here all processes**



[8] LLNL MPI Tutorial

- Extended Simple C Program with MPI functionality
  - hello.c is not executable as C programm → it needs a compilation

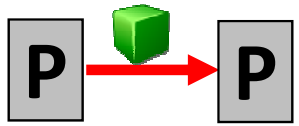


New Steps Required: Start 'Thinking' Parallel – Revisited (cf. Lecture 2)

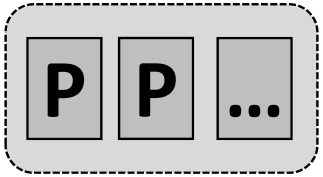
- Parallel Processing Approach
  - Parallel MPI programs know about the existence of other processes of it and what their own role is in the bigger picture
  - MPI programs are written in a sequential programming language, but executed in parallel
  - Same MPI program runs on all processes (SPMD)

- SPMD stands for Single Program Multiple Data

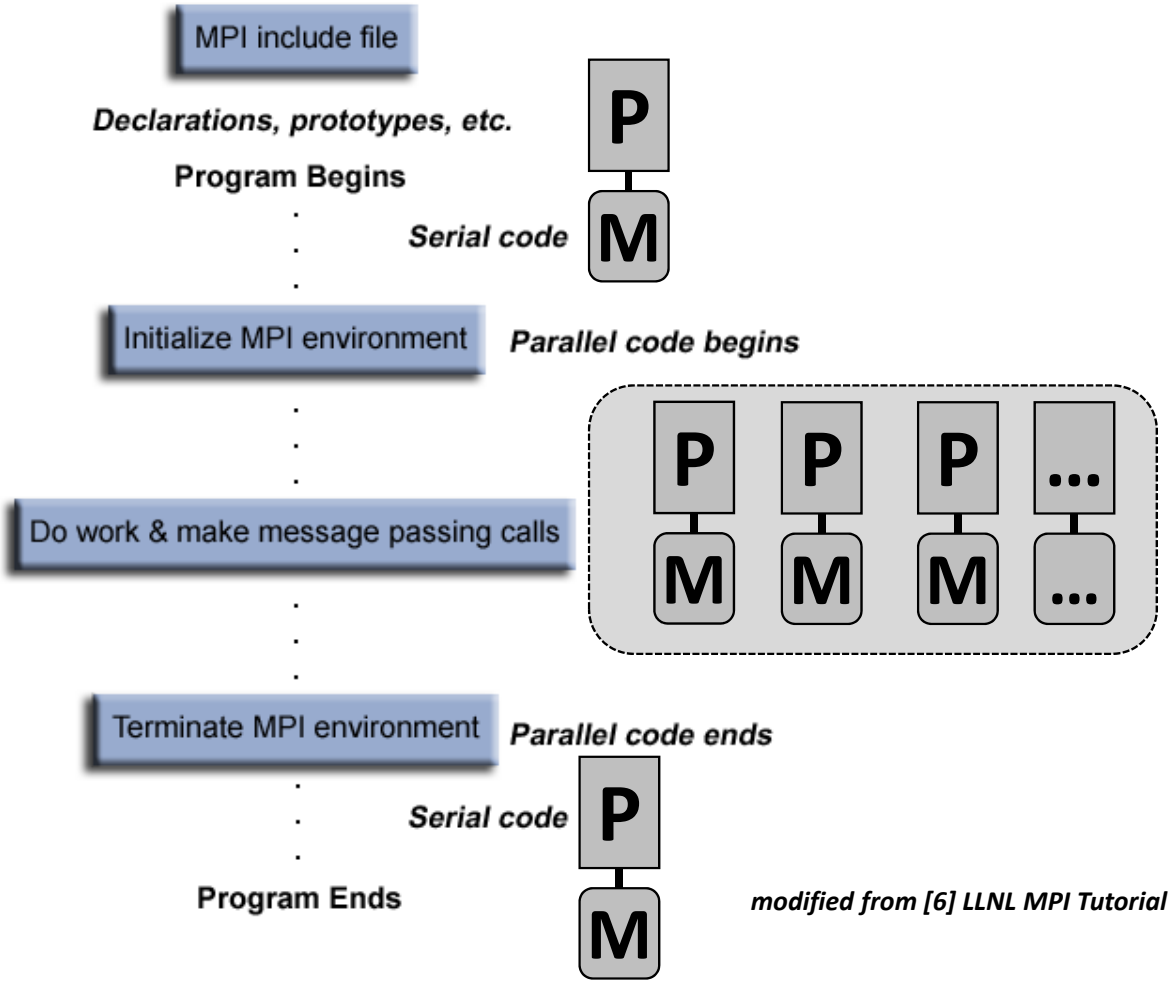
- Data exchange is key for design of applications
  - Sending/receiving data at specific times in the program
  - No shared memory for sharing variables with other remote processes
  - Messages can be simple variables (e.g. a word) or complex structures



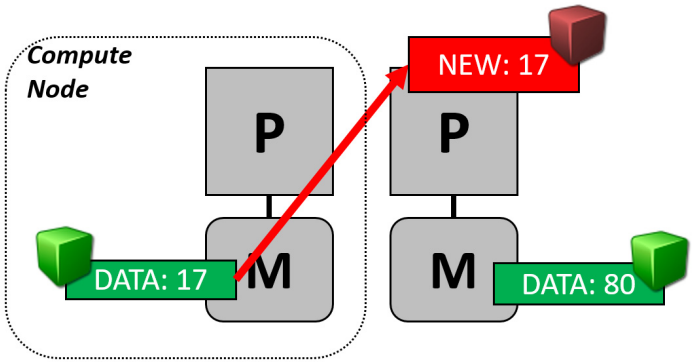
- Start with the basic building blocks using MPI
  - Building up the 'parallel computing environment'

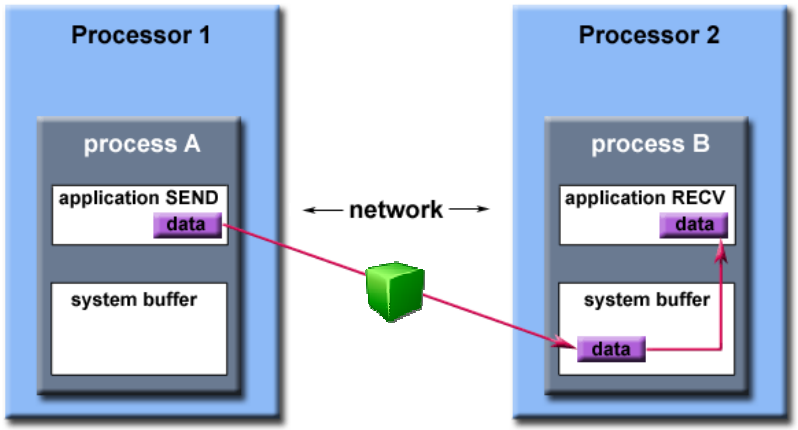


Parallel Environment & Message Passing – PingPong Application Example



modified from [6] LLNL MPI Tutorial





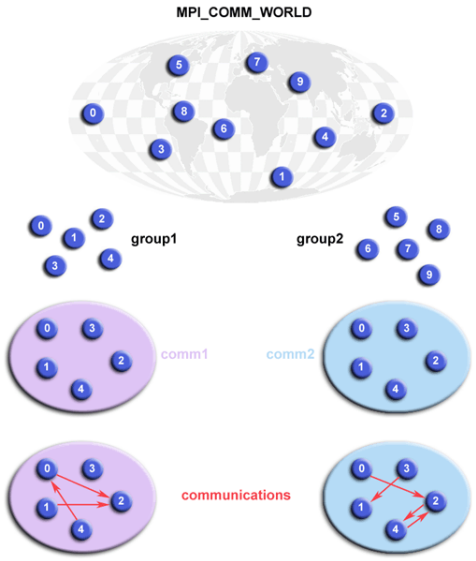
Path of a message buffered at the receiving process

Add to Step 4: Edit a Text File – Defining Variables & Using Rank for Identity

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
  int numtasks, rank, dest, source, rc, count, tag=1;
  char inmsg, outmsg='x';


  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    dest = 1; source = 1;


  }
  else if (rank ==1 ) {
    dest = 0; source = 0;


  }









  MPI_Finalize();
  return 0;
}
```

- **Defining variables: dest and source are required to identify the ranks between the messages should be passed**
- **Defining variables: count indicates how many characters are transferred in a message passing command**
- **Defining variables: tag indicates which message transfer we use**
- **Defining variables: inmsg & outmsg indicates what char we put as message**

- **Our obtained unique rank can be used to influence the execution of the program depending on the unique process identity**
- **We can use the rank information to give processes specific roles like master / workers (cf. our Lecture 3 and/or domain decompositions with specific functions to execute)**
- **For our ping-pong example we want to send a message 'ping' from one rank to another and also want to receive a 'pong' back from another rank: defining dest and source using rank identities**



[8] LLNL MPI Tutorial

➢ **Lecture 4 will offer more insights about using different types of MPI communicators with different rank identities in MPI applications**

Add to Step 4: Edit a Text File – MPI Send/Recv Functions

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
  int numtasks, rank, dest, source, rc, count, tag=1;
  char inmsg, outmsg='x';


  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    dest = 1; source = 1;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }
  else if (rank ==1 ) {
    dest = 0; source = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }


  printf("Task %d: Received %d char(s) from task %d with tag %d \n",
    rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
  MPI_Finalize();
  return 0;
}
```

- **MPI_Send() function is used to send a certain number of elements of some datatype to another MPI rank; this routine blocks until the message is received by the destination process**
- **MPI_Recv() function is used to receive a certain number of elements of some datatype from another MPI rank; this routine blocks until the message is received and thus send by the source process**
- **This form of MPI communication is called 'blocking' while there are also possibilities to have 'non-blocking' communication in MPI applications**

```
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)
```

[9] DEINO MPI & Examples

➢ **Lecture 4 will offer more insights about using blocking communication vs. non-blocking communication functions when using MPI**

Add to Step 4: Edit a Text File – MPI Status & MPI_Get_count

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
  int numtasks, rank, dest, source, rc, count, tag=1;
  char inmsg, outmsg='x';

  MPI_Status Stat;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    dest = 1; source = 1;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }
  else if (rank ==1 ) {
    dest = 0; source = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

  rc = MPI_Get_count(&Stat, MPI_CHAR, &count);

  printf("Task %d: Received %d char(s) from task %d with tag %d \n",
    rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
  MPI_Finalize();
  return 0;
}
```

- **MPI_Status is a variable that includes a lot of information about the corresponding MPI function call**
- **We use the MPI_Status in our example to check how much chars we really transferred by using the MPI_Get_count() function**
- **As a simple debug possibility we can check whether the MPI_Status information about source and tag of the messages are corresponding to our idea of programming**

```
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)
```

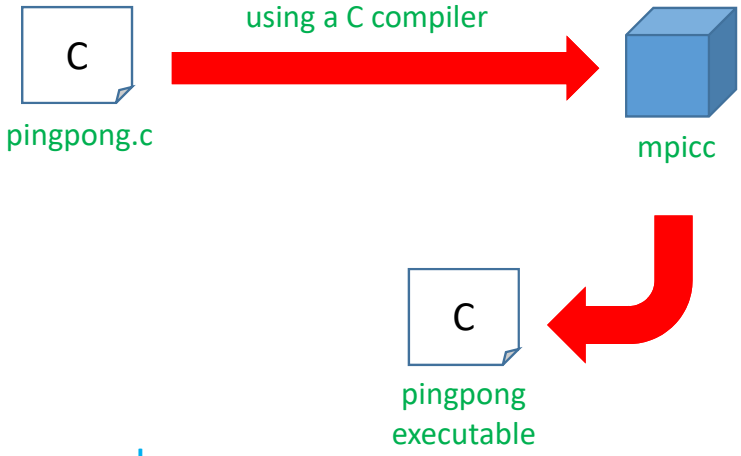
[9] DEINO MPI & Examples

➢ **Lecture 4 will offer more insights about using the MPI status for different purposes and to obtain a better understanding what happens**

Step 5: Load the right Modules for Compilers & Compile C & MPI Program

- Using modules to get the right C compiler for compiling pingpong.c
  - ‘module load gnu openmpi‘
  - Note: there are many C compilers available, we here pick one for our particular HPC course that works with the Message Passing Interface (MPI)
  - Note: If there are no errors, the file pingpong is now a full C program executable that can be started by an OS
  - New: C program with MPI message exchanges (cf. Lecture 2 – Parallel Programming with MPI)

```
[morris@jotunn pingpong]$ pwd
/home/morris/2019-HPC-Course/pingpong
[morris@jotunn pingpong]$ module load gnu openmpi
[morris@jotunn pingpong]$ mpicc pingpong.c -o pingpong
[morris@jotunn pingpong]$ ls -al
total 20
drwxrwxr-x 2 morris morris   63 sep 16 07:36 .
drwxrwxr-x 5 morris morris   49 sep 16 07:21 ..
-rwxrwxr-x 1 morris morris 8816 sep 16 07:36 pingpong
-rwxr-xr-x 1 morris morris  904 sep 16 07:26 pingpong.c
-rwxr-xr-x 1 morris morris  180 sep 16 07:26 submit-pingpong.sh
```

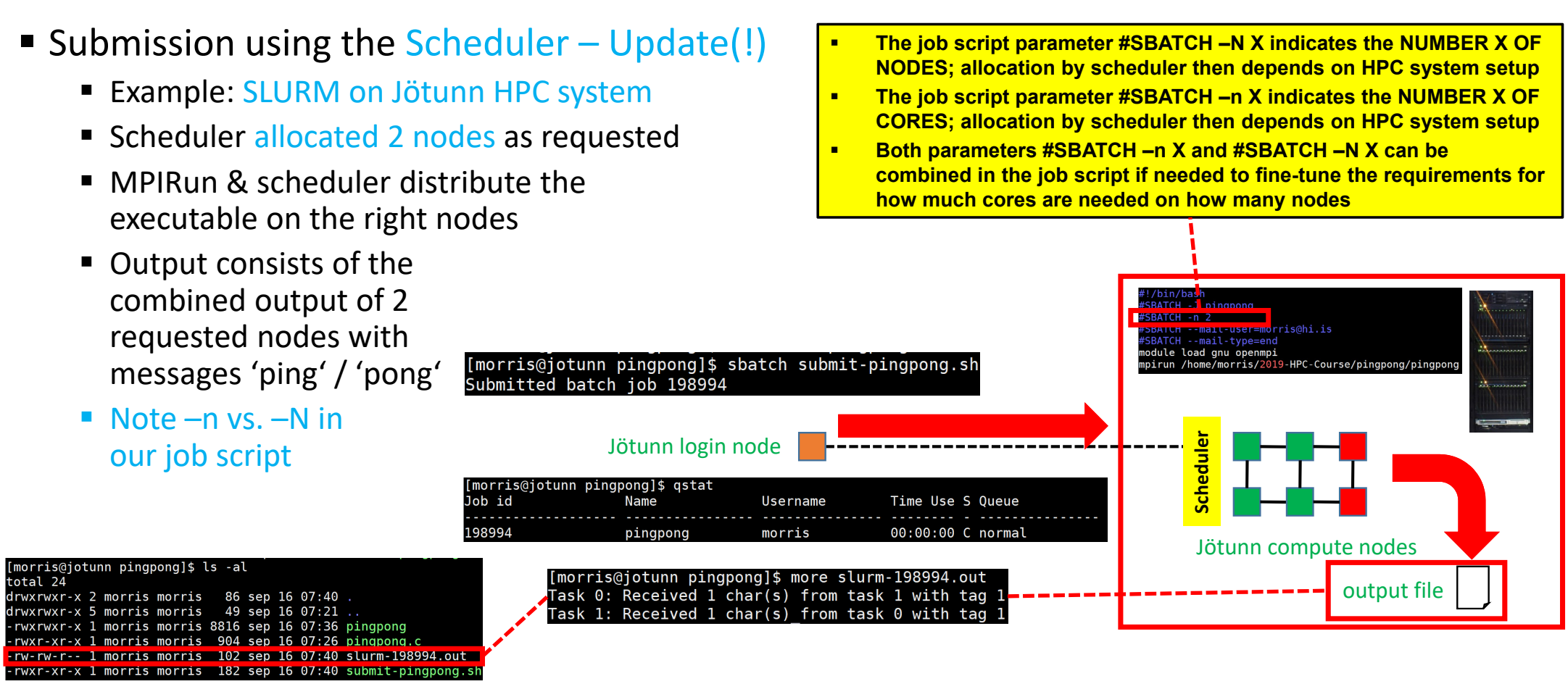


[7] Icelandic HPC Machines & Community

Step 6: Parallel Processing – Executing an MPI Program with MPIRun & Script

- Submission using the Scheduler – Update(!)
  - Example: SLURM on Jötunn HPC system
  - Scheduler allocated 2 nodes as requested
  - MPIRun & scheduler distribute the executable on the right nodes
  - Output consists of the combined output of 2 requested nodes with messages ‘ping‘ / ‘pong‘
  - Note –n vs. –N in our job script

- **The job script parameter #SBATCH –N X indicates the NUMBER X OF NODES; allocation by scheduler then depends on HPC system setup**
- **The job script parameter #SBATCH –n X indicates the NUMBER X OF CORES; allocation by scheduler then depends on HPC system setup**
- **Both parameters #SBATCH –n X and #SBATCH –N X can be combined in the job script if needed to fine-tune the requirements for how much cores are needed on how many nodes**



Step 6: SLURM – Scontrol & Job Script Parameters Fine Tuning

```
#!/bin/bash
#SBATCH -J pingpong
#SBATCH -N 2
#SBATCH --mail-user=morris@hi.is
#SBATCH --mail-type=end
module load gnu openmpi
mpirun /home/morris/2019-HPC-Course/pingpong/pingpong
```

```
#!/bin/bash
#SBATCH -J pingpong
#SBATCH -n 2
#SBATCH --mail-user=morris@hi.is
#SBATCH --mail-type=end
module load gnu openmpi
mpirun /home/morris/2019-HPC-Course/pingpong/pingpong
```

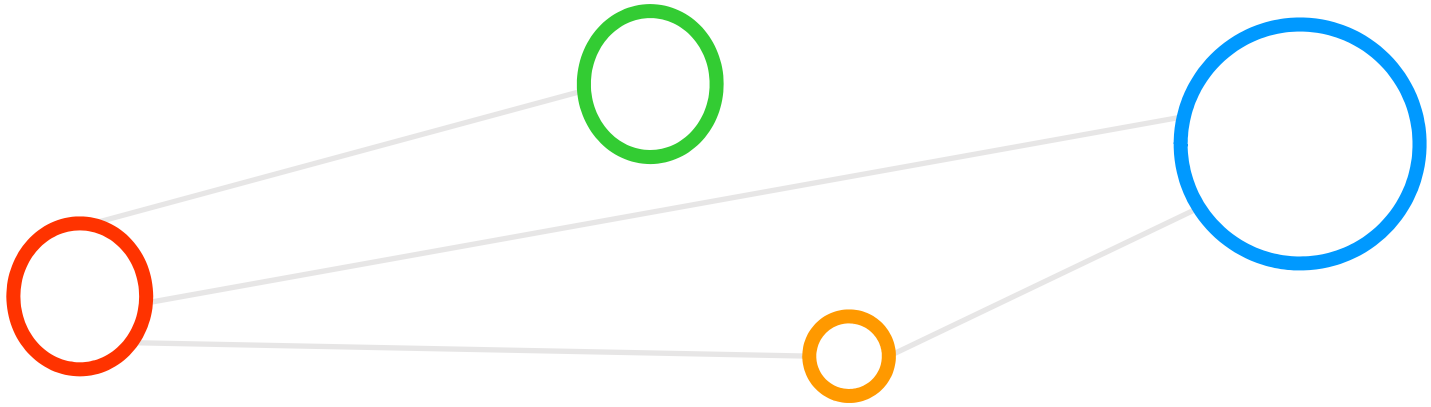
```
[morris@jotunn pingpong]$ scontrol show jobid -dd 198996
JobId=198996 JobName=pingpong
   UserId=morris(30017) GroupId=morris(30017)
   Priority=4294895170 Nice=0 Account=(null) QOS=normal
   JobState=COMPLETED Reason=None Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=0:0
   DerivedExitCode=0:0
   RunTime=00:00:01 TimeLimit=02:00:00 TimeMin=N/A
   SubmitTime=2019-09-16T08:02:55 EligibleTime=2019-09-16T08:02:55
   StartTime=2019-09-16T08:02:55 EndTime=2019-09-16T08:02:56
   PreemptTime=None SuspendTime=None SecsPreSuspend=0
   Partition=normal AllocNode:Sid=jotunn:27823
   ReqNodeList=(null) ExcNodeList=(null)
   NodeList=compute-2-[0-1]
   BatchHost=compute-2-0
   NumNodes=2 NumCPUs=2 CPUs/Task=1 ReqB:S:C:T=0:0:*:*
   TRES=cpu=2,node=2
   Socks/Node=* NtasksPerN:B:S:C=0:0:*:* CoreSpec=*
     Nodes=compute-2-[0-1] CPU_IDs=0 Mem=0
   MinCPUsNode=1 MinMemoryNode=0 MinTmpDiskNode=0
   Features=(null) Gres=(null) Reservation=(null)
   Shared=OK Contiguous=0 Licenses=(null) Network=(null)
   Command=/home/morris/2019-HPC-Course/pingpong/submit-pingpong.sh
   WorkDir=/home/morris/2019-HPC-Course/pingpong
   StdErr=/home/morris/2019-HPC-Course/pingpong/slurm-198996.out
   StdIn=/dev/null
   StdOut=/home/morris/2019-HPC-Course/pingpong/slurm-198996.out
   Power= SICP=0
```

Scheduler

(enables a more fine-grained request & allocation of required cores/nodes)

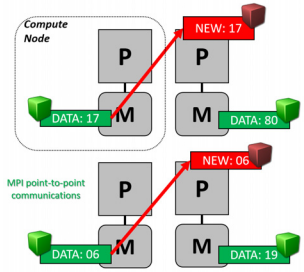
```
[morris@jotunn pingpong]$ scontrol show jobid -dd 198995
JobId=198995 JobName=pingpong
   UserId=morris(30017) GroupId=morris(30017)
   Priority=4294895171 Nice=0 Account=(null) QOS=normal
   JobState=COMPLETED Reason=None Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=0:0
   DerivedExitCode=0:0
   RunTime=00:00:00 TimeLimit=02:00:00 TimeMin=N/A
   SubmitTime=2019-09-16T07:59:55 EligibleTime=2019-09-16T07:59:55
   StartTime=2019-09-16T07:59:55 EndTime=2019-09-16T07:59:55
   PreemptTime=None SuspendTime=None SecsPreSuspend=0
   Partition=normal AllocNode:Sid=jotunn:27823
   ReqNodeList=(null) ExcNodeList=(null)
   NodeList=compute-2-0
   BatchHost=compute-2-0
   NumNodes=1 NumCPUs=2 CPUs/Task=1 ReqB:S:C:T=0:0:*:*
   TRES=cpu=2,node=1
   Socks/Node=* NtasksPerN:B:S:C=0:0:*:* CoreSpec=*
     Nodes=compute-2-0 CPU_IDs=0,12 Mem=0
   MinCPUsNode=1 MinMemoryNode=0 MinTmpDiskNode=0
   Features=(null) Gres=(null) Reservation=(null)
   Shared=OK Contiguous=0 Licenses=(null) Network=(null)
   Command=/home/morris/2019-HPC-Course/pingpong/submit-pingpong.sh
   WorkDir=/home/morris/2019-HPC-Course/pingpong
   StdErr=/home/morris/2019-HPC-Course/pingpong/slurm-198995.out
   StdIn=/dev/null
   StdOut=/home/morris/2019-HPC-Course/pingpong/slurm-198995.out
   Power= SICP=0
```

Understanding MPI Collectives & Message Exchange Options

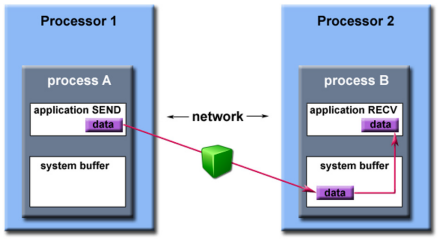


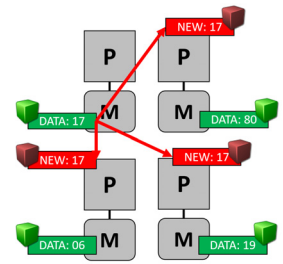
Parallel Programming with MPI & MPI Collective Functions (cf. Lecture 2)

- Message Passing Interface (MPI) Concepts

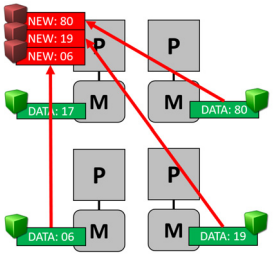


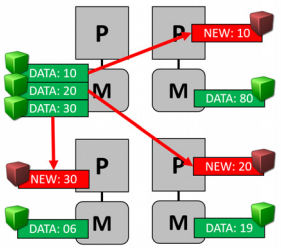
MPI Point to Point Communication

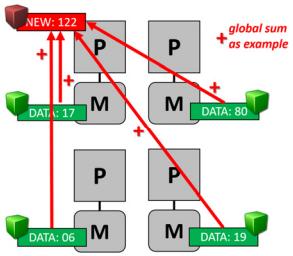




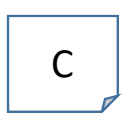
MPI Collective Communication







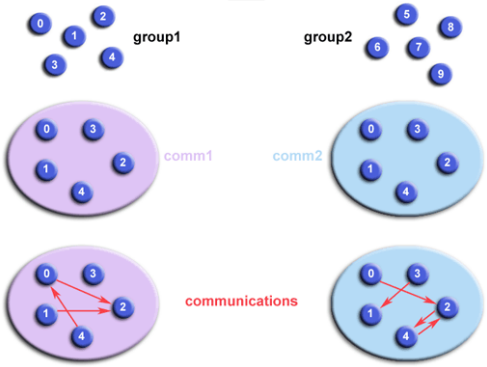
- MPI Parallel Programming Basics



using a C compiler

using a job script


```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello World, I am %d out of %d\n",
         rank, size);
  MPI_Finalize();
  return 0;
}
```

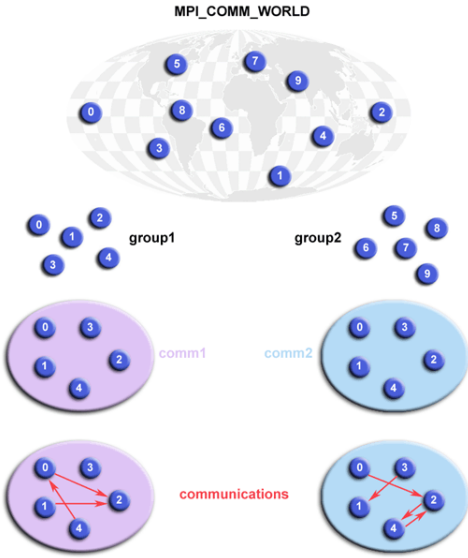


[6] LLNL MPI Tutorial

Add to Step 4: Edit a Text File – Defining Variables & Using Rank for Identity

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
  int i,rank, numprocs;
  int source,count;
  int buffer[4];

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  source=0;
  count=4;

  if(rank == source){
    for(i=0;i<count;i++)
      buffer[i]=i;
  }

  for(i=0;i<count;i++)
    printf("%d \n",buffer[i]);

  MPI_Finalize();
  return 0;
}
```

- Defining variables: source indicates who initiates the broadcast MPI collective function; no receiving identity is required because we will broadcast to all processes (i.e., here in the same communicator space MPI_COMM_WORLD)
- Defining variables: We will send four integers in the broadcast and need to define a buffer for sending and a buffer for receiving four integers (!)
- Here we use the different unique job identity rank to fill the buffer with four integer values
- All processes should print out their buffer, also for those ranks that did not initialize the buffer



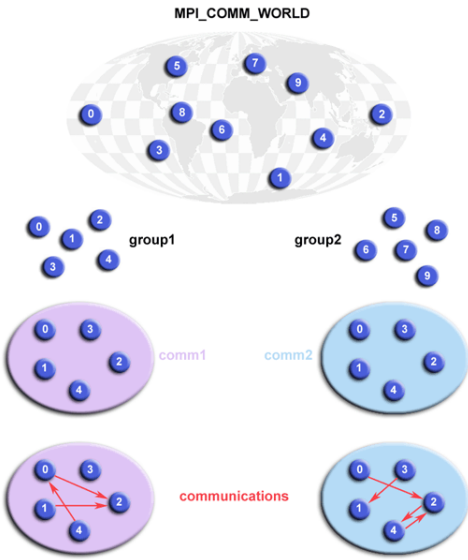
[8] LLNL MPI Tutorial

➢ **Assignment #1 includes the use of MPI collective functions and will enable you to explore different type of MPI collective operations**

Add to Step 4: Edit a Text File – Defining Variables & Using Rank for Identity

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
  int i,rank, numprocs;
  int source,count;
  int buffer[4];

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  source=0;
  count=4;

  if(rank == source){
    for(i=0;i<count;i++)
      buffer[i]=i;
  }

  MPI_Bcast(buffer,count,MPI_INT,source,MPI_COMM_WORLD);

  for(i=0;i<count;i++)
    printf("%d \n",buffer[i]);

  MPI_Finalize();
  return 0;
}
```

- The MPI_Bcast() function broadcasts a message from the process with rank root to all processes of the group, itself included (!)
- The function if called by all members of the selected communicator group (i.e., here MPI_COMM_WORLD) using all the same arguments)
- Note: only for rank 0 is the buffer properly initialized in this example



[8] LLNL MPI Tutorial

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

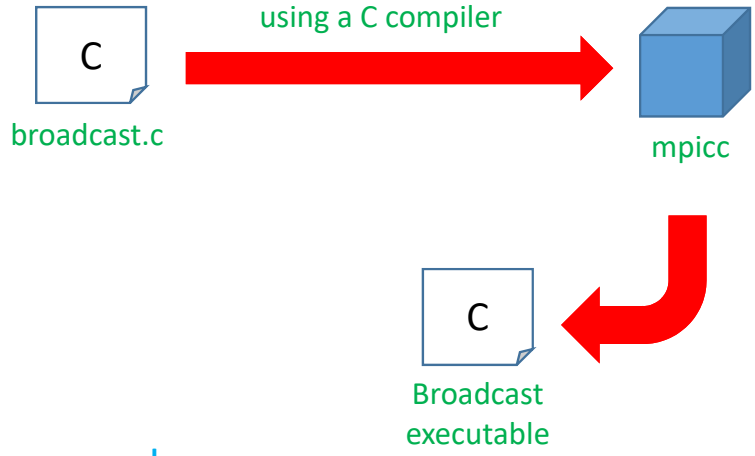
[9] DEINO MPI & Examples

➢ **Assignment #1 includes the use of MPI collective functions and will enable you to explore different type of MPI collective operations**

Step 5: Load the right Modules for Compilers & Compile C & MPI Program

- Using modules to get the right C compiler for compiling broadcast.c
  - ‘module load gnu openmpi‘
  - Note: there are many C compilers available, we here pick one for our particular HPC course that works with the Message Passing Interface (MPI)
  - Note: If there are no errors, the file broadcast is now a full C program executable that can be started by an OS
  - New: C program with MPI message exchanges (cf. Lecture 2 – Parallel Programming with MPI)

```
[morris@jotunn broadcast]$ pwd
/home/morris/2019-HPC-Course/broadcast
[morris@jotunn broadcast]$ module load gnu openmpi
[morris@jotunn broadcast]$ mpicc broadcast.c -o broadcast
[morris@jotunn broadcast]$ ls -al
total 20
drwxrwxr-x 2 morris morris   66 sep 16 09:37 .
drwxrwxr-x 6 morris morris   65 sep 16 09:11 ..
-rwxrwxr-x 1 morris morris 8730 sep 16 09:37 broadcast
-rwxr-xr-x 1 morris morris  579 sep 16 09:30 broadcast.c
-rwxr-xr-x 1 morris morris  183 sep 16 09:17 submit-broadcast.sh
```



[7] Icelandic HPC Machines & Community

Step 6: Parallel Processing – Executing an MPI Program with MPIRun & Script

- Submission using the Scheduler – Update(!)
  - Example: SLURM on Jötunn HPC system
  - Scheduler allocated 4 nodes as requested
  - MPIRun & scheduler distribute the executable on the right nodes
  - Output consists of the buffer from all the involved processes that was filled by rank 0 with content
  - Note –n vs. –N in our job script

- **The job script parameter #SBATCH –N X indicates the NUMBER X OF NODES; allocation by scheduler then depends on HPC system setup**
- **The job script parameter #SBATCH –n X indicates the NUMBER X OF CORES; allocation by scheduler then depends on HPC system setup**
- **Both parameters #SBATCH –n X and #SBATCH –N X can be combined in the job script if needed to fine-tune the requirements for how much cores are needed on how many nodes**

```
[morris@jotunn broadcast]$ sbatch submit-broadcast.sh
Submitted batch job 199002
```

```
[morris@jotunn broadcast]$ qstat
Job id              Name             Username        Time Use S Queue
------------------- ---------------- --------------- -------- - ---------------
199001              hello-mpi-exampl gem9            00:00:00 C normal
199002              broadcast        morris          00:00:00 C normal
```

```
#!/bin/bash
#SBATCH -J broadcast
#SBATCH -n 4
#SBATCH --mail-user=morris@hi.is
#SBATCH --mail-type=end
module load gnu openmpi
mpirun /home/morris/2019-HPC-Course/broadcast/broadcast
```

Jötunn login node

Scheduler

Jötunn compute nodes

output file

```
[morris@jotunn broadcast]$ more slurm-199002.out
0
1
2
3
0
1
2
3
0
1
2
3
0
1
2
3
```

```
[morris@jotunn broadcast]$ ls -al
total 24
drwxrwxr-x 2 morris morris   89 sep 16 09:41 .
drwxrwxr-x 6 morris morris   65 sep 16 09:11 ..
-rwxrwxr-x 1 morris morris 8730 sep 16 09:37 broadcast
-rwxr-xr-x 1 morris morris  579 sep 16 09:30 broadcast.c
-rw-rw-r-- 1 morris morris   48 sep 16 09:39 slurm-199002.out
-rwxr-xr-x 1 morris morris  185 sep 16 09:41 submit-broadcast.sh
```

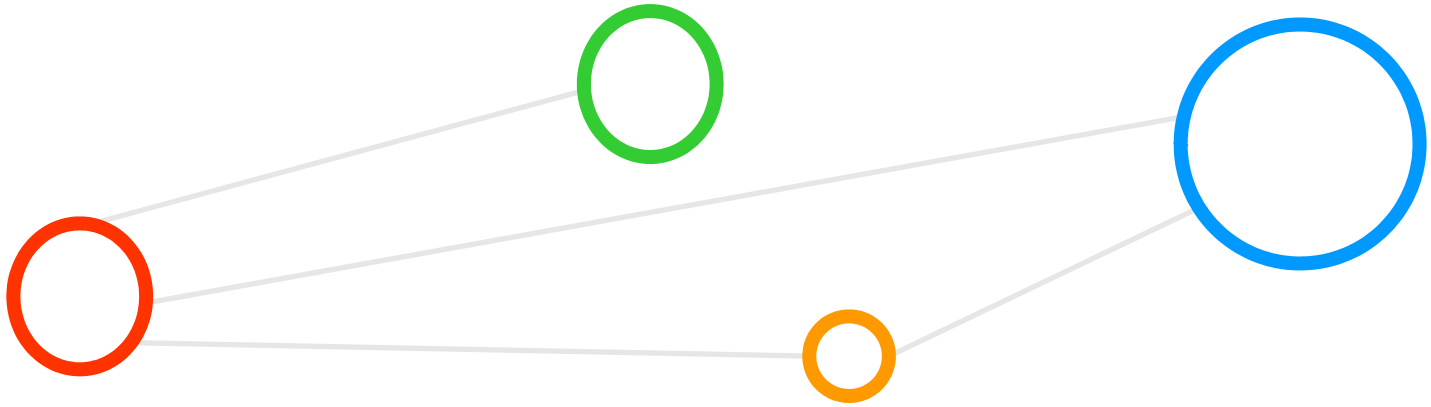
Exploring the Walltime – What happens when the job runs against the wall?

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

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

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("Hello World, I am %d of %d and going to sleep now.\n", rank, size);

  sleep(5);

  printf("Hello World, I am %d of %d and just awake now.\n", rank, size);

  MPI_Finalize();

  return 0;
}
```

```
#!/bin/bash
#SBATCH -J hello-example
#SBATCH -n 2
#SBATCH --time=00:01:00
#SBATCH --mail-user=morris@hi.is
#SBATCH --mail-type=end
module load gnu openmpi
mpirun /home/morris/2019-HPC-Course/hellosleep/hellosleep
```

- **Assignment #1 includes the use of the sleep command and the use of walltime with the SLURM scheduler using the –time option**

Lecture Bibliography



Lecture Bibliography

- [1] 2013 SMU HPC Summer Workshop, Session 8: Introduction to Parallel Computing, Online: http://dreynolds.math.smu.edu/SMUHPC_workshop/session_8.html
- [2] Introduction to Parallel Computing Tutorial, Online: https://computing.llnl.gov/tutorials/parallel_comp/
- [3] Introduction to High Performance Computing for Scientists and Engineers, Georg Hager & Gerhard Wellein, Chapman & Hall/CRC Computational Science, ISBN 143981192X
- [4] PEPC Webpage, FZ Juelich, Online: http://www.fz-juelich.de/ias/jsc/EN/AboutUs/Organisation/ComputationalScience/Simlabs/slpp/SoftwarePEPC/_node.html
- [5] M. Goetz, C. Bodenstein, M. Riedel, 'HPDBSCAN – Highly Parallel DBSCAN', in proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC2015), Machine Learning in HPC Environments (MLHPC) Workshop, 2015, Online: https://www.researchgate.net/publication/301463871_HPDBSCAN_highly_parallel_DBSCAN
- [6] LLNL MPI Tutorial, Online: https://computing.llnl.gov/tutorials/mpi/
- [7] Icelandic HPC Machines & Community, Online: http://ihpc.is
- [8] Caterham F1 Team Races Past Competition with HPC, Online: http://insidehpc.com/2013/08/15/caterham-f1-team-races-past-competition-with-hpc
- [9] DEINO MPI & Examples, Online: https://mpi.deino.net/


Data
Big
Science
Research
science
simulations
methods
Computational
E.g
Centre
Computing
big
nodes
centers
PB
community
Environment
storage
ENES
technologies
processing
computing
TB
European
Supercomputing
Scientific
images
performance
datasets
management
analysis
databases
computational
research
HPC
Juelich
data-intensive
Service
Climate
modelling
access
structures
System
DLCLs
Germany
Provide
NASA
Energy
systems
day
Health
Processing
distributed
large-scale
cores
project
General
current
Structure
Hardware
AR5
Technologies
Earth
directory
Simulation
Understanding
compute
hundreds
Nutshell
Services
network
Landscape
system
Concrete
Matters
Summary
Large
Resources
stored
analyze
DLCL
Level
use
JSC
e.g
acquisition
Project
MapReduce
create
GB
Work
Using
Common
Methods
IPCC
physical
EUDAT
via
increasing
brain
formats
References
engineering
SMAQ
international
model
codes
Cross-Disciplinary
EU
like
year
forms
cross-disciplinary
resources
services
measurement
Policy-based
within
fusion
new
funding
concepts
device
Analysis
climate
Computer
expertise
Test
High
etc
DKRZ
Use
disciplines
results
Approaches
construction
Fusion
Enable
Modelling
Key
scientific
important
Infrastructure
basis
files
different
external
levels
often