Software solutions in High Performance Computing

Matthias Rauter

Universität Innsbruck, fall 2024

Agenda

1. About me, about this.
2. Basics of HPC: Hardware, history, architecture: Distributed memory
3. Introduction to Message Passing Interface (MPI)
4. Coding with MPI

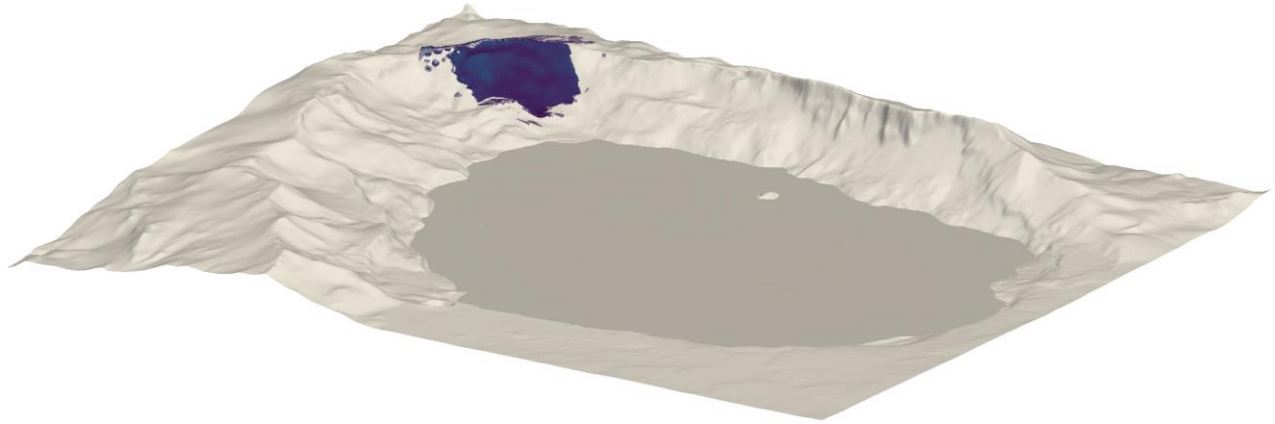
5. Exercise: Calculate Pi with MPI
6. Building and running on a cluster
7. Questions.

8. Extra: Sheard memory: Pthread, Mutex, Race conditions

About me

Matthias Rauter: Education

- MSc from UIBK
- Started C++ as hobby (lol)
- 2 PhDs in Engineering / Math
  - Avalanches (UIBK)
  - Tsunamis (UiO)
  - Both with HPC
  - Both with CFD



Nat Commun 13, 678 (2022), https://doi.org/10.1038/s41467-022-28296-7

Matthias Rauter: Work

- Worked for Government in Natural Hazard Mgmt
- Short Postdoc at BOKU
- Now: The Qt Company
  - Senior/Staff Dev Graphics & UI
  - Team Lead Core & Network

Matthias Rauter: Contact

- **Mail**: matthias@rauter.it
- **Code**: https://github.com/mattisnowman
- Papers: https://www.researchgate.net/profile/Matthias-Rauter
- LinkedIn: https://www.linkedin.com/in/matti-rauter/

About this

https://github.com/mattisnowman/mpiplayground

Sources:

- **Standards & library documentation:** Hard to read but trustable.
  - Standard: https://www.mpi-forum.org/docs/
  - C docs: https://www.open-mpi.org/doc/
  - Python docs: https://mpi4py.readthedocs.io/en/stable/
  - Extra: Unix threads: https://man7.org/linux/man-pages/man7/pthreads.7.html
  - Extra: QThreads: https://doc.qt.io/qt-6/qthread.html
- **Books**: Didn't find any outstanding books. Happy for tips.
- **Wikipedia:** Good for general information/history. Quite reliable but check with standards if important.
- **ChatGPT:** Well informed about topic, produces correct code for simple stuff, fails for complex stuff.
- **Try & Error:** Just to make sure…

If no source is given, it is from Wikipedia.

Basics of High Performance Computing

Goals:

1. Understand the basics and how we got here.
2. Make sense out of HPC error messages & issues.

Methods:

1. Quick look at history and architecture
2. Write an example from scratch (C/C++ because I was told python-mpi does not work here)

Grading:

1. Finish / change the example.
2. Answer two questions via mail.

History, Hardware, Architecture

https://en.wikipedia.org/wiki/History_of_supercomputing

1960-1990:

- Learning to walk
- Very specific "CPU" designs: One-offs
- Not much concurrency

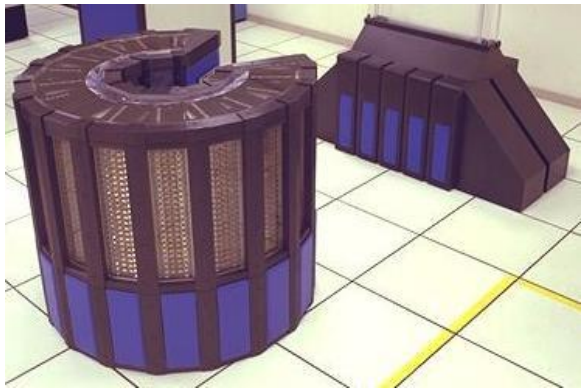
1963: Atlas

pioneered virtual memory and paging

1964: CDC6000

3 MFLOPS, it was dubbed a supercomputer

1985: *Cray 2*



Four-processor fluorinert cooled computer.

from 1990:

- Architecture with massive amounts of CPUs
- Parallelization and distributed computing becomes important
- Using mainstream hardware in huge amounts

1994: Fujitsu NWT

166 vector processors

Picture: blog.de.fujitsu.com

1996: Hitachi SR2201

2048 RISC processors

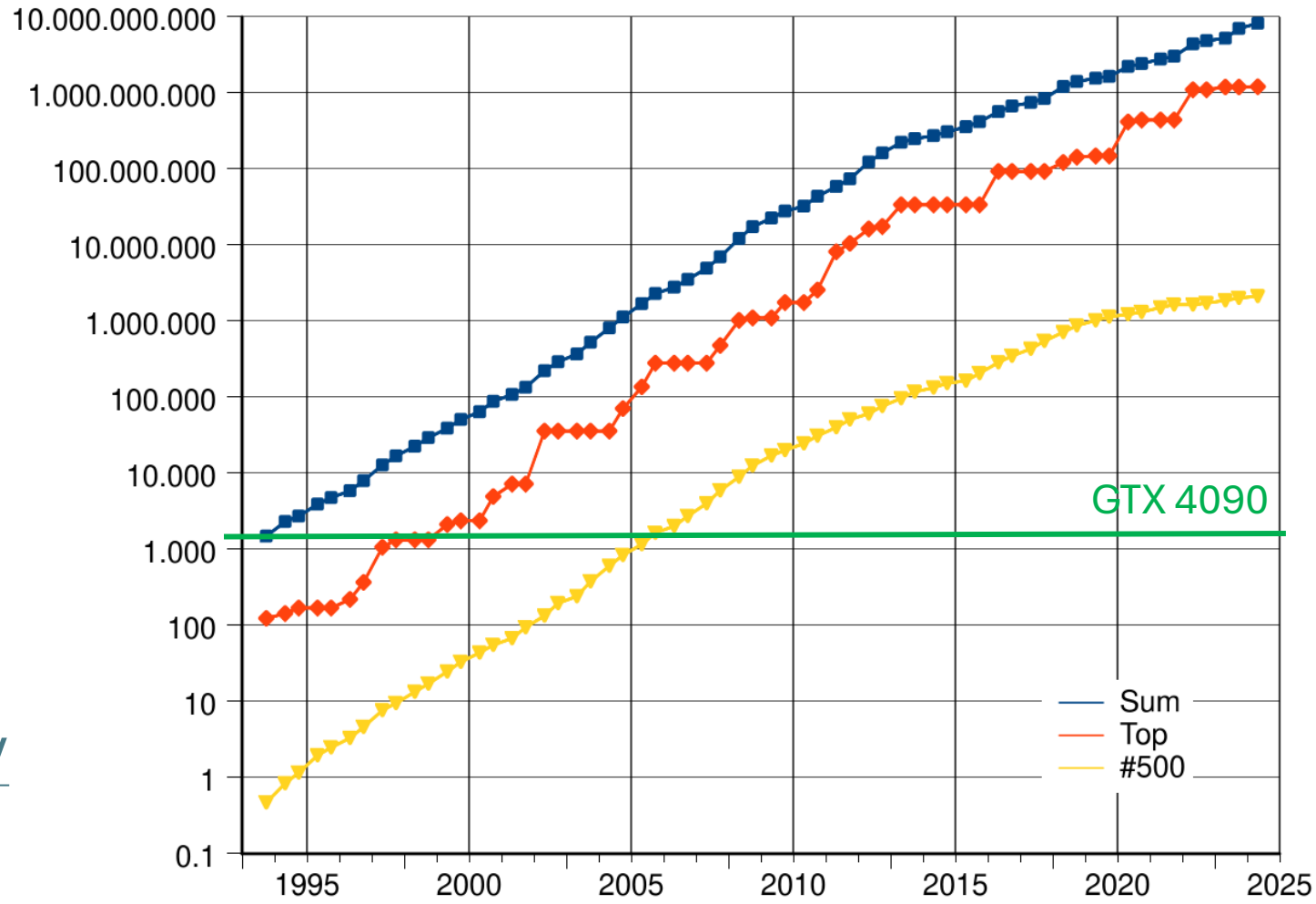
Picture: museum.ipsj.or.jp

1990++: Intel Paragon

1000 to 4000 Intel i860 processors. MPI 🥳

Top500.org

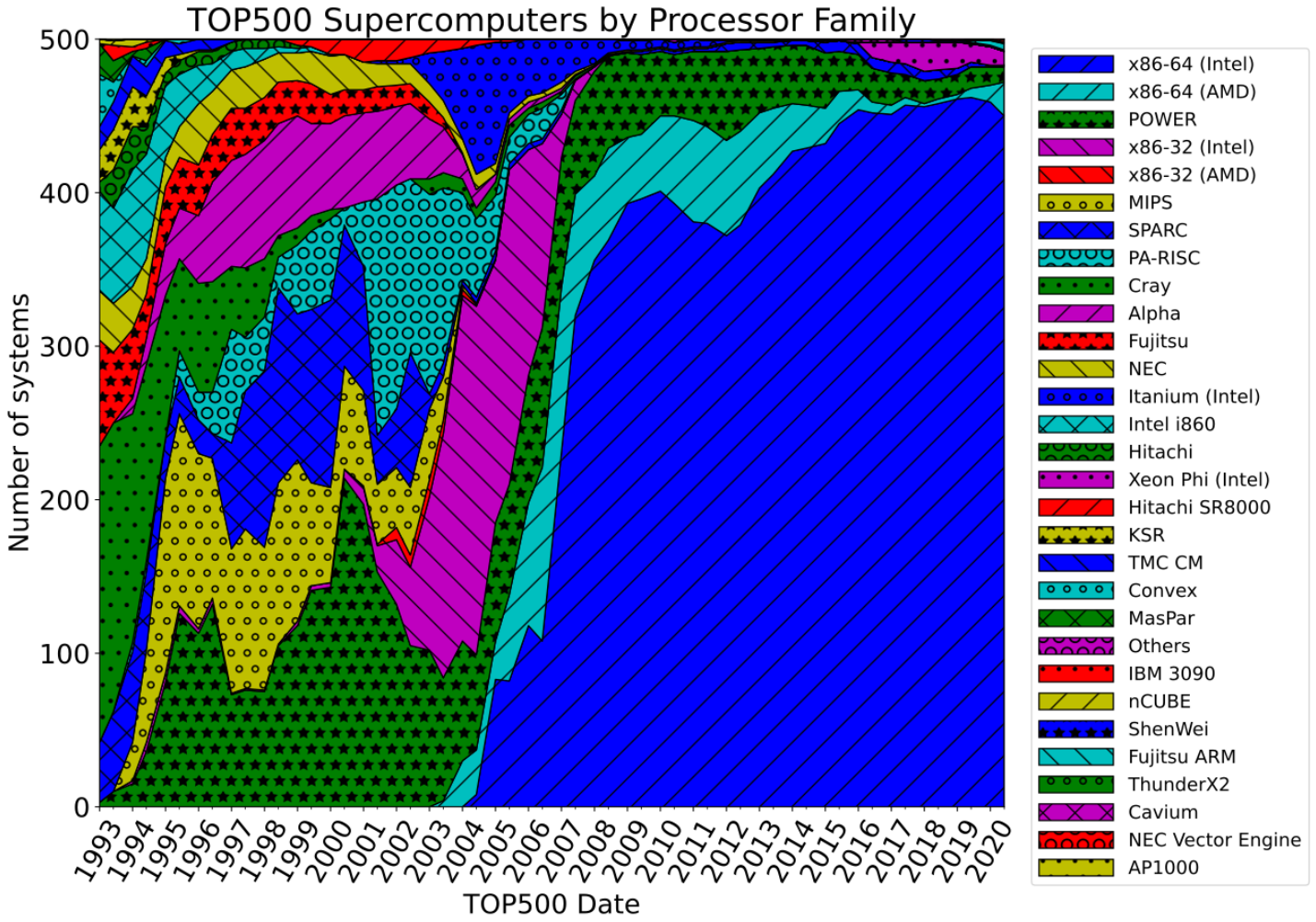
- Est. 1993
- Tracks the top-500 super computers
- LINPACK benchmark 64-bit floating-point op
- Take a look at www.top500.org/25years/



Top500 Performance in GFLOPS 64-bit floating point (double)

Top500.org

- Est. 1993
- Tracks the top-500 super computers
- LINPACK benchmark 64-bit floating-point op
- Take a look at www.top500.org/25years/



How to parallelize?

- Each CPU/Core has dedicated memory:
  - Distributed Calculation Power
  - Distributed Data
- Split tasks into independent subtasks:
  - Multiple levels.
  - Multiple solutions.
- Classic HPC: Loop level – All data interacts sooner or later.

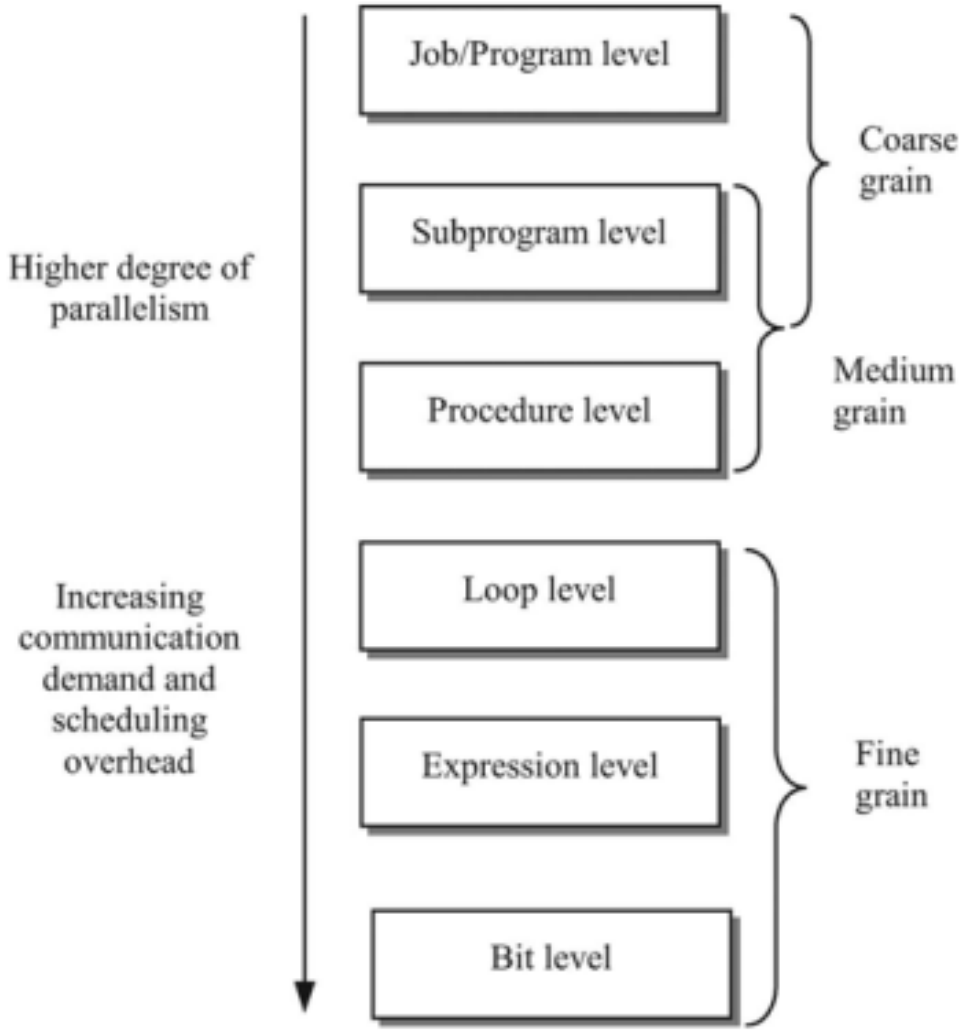


Figure 1.3. Levels of parallelism

M. O. Tokhi, M. A. Hossain, M. H. Shaheed: Parallel Computing for Real-time Signal Processing and Control

How to parallelize?

- Lowest level (on CPU):
  - Shared memory
  - Threads, OpenMP, pthreads
  - MPI
  - SIMD
- Mid level (HPC):
  - Distributed memory
  - MPI
- High level:
  - Independent Processes
  - Use the OS/Service to start many processes

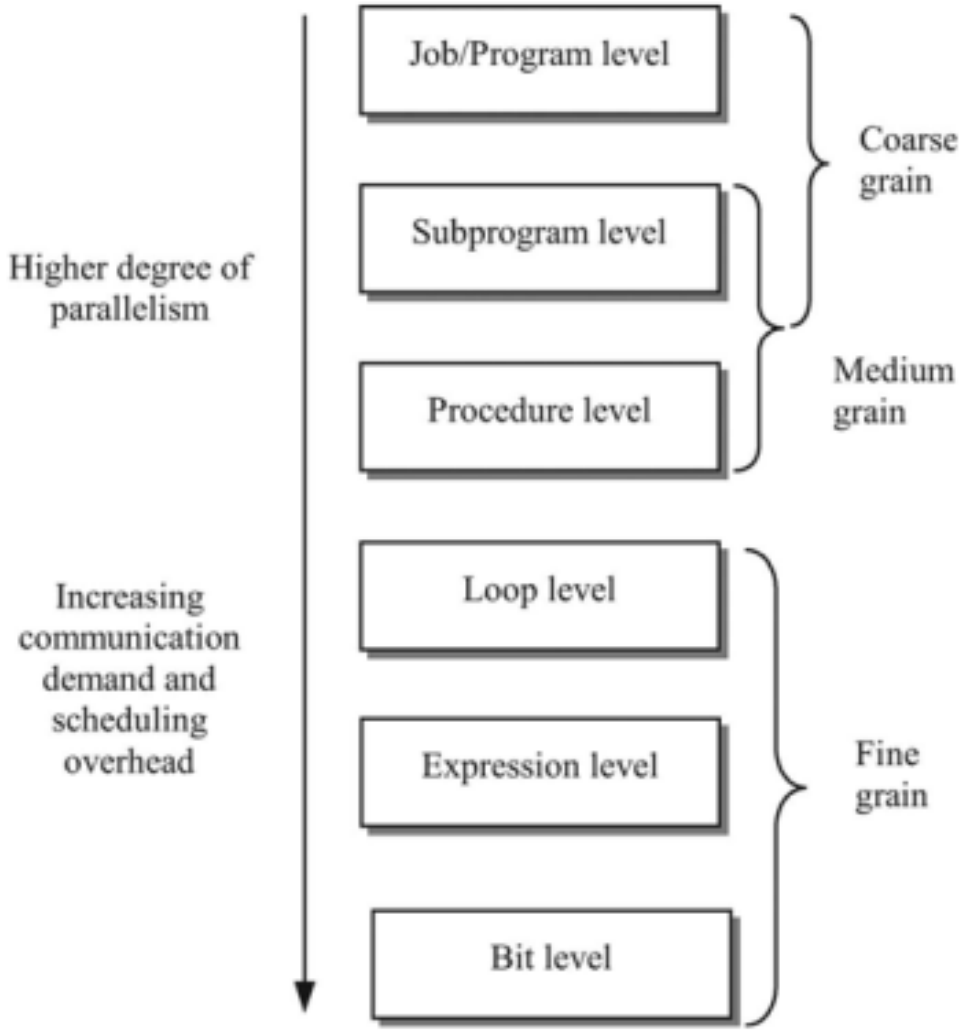


Figure 1.3. Levels of parallelism

M. O. Tokhi, M. A. Hossain, M. H. Shaheed: Parallel Computing for Real-time Signal Processing and Control

HPCluster at UIBK

- LEO 3-5 (I wasn't here before):
  - Distributed Memory
  - Some GPUs nodes since Leo3e
  - Now LCC
  - **Run with MPI**
- Mach 2:
  - "It comprises one large shared memory system with 2048 Intel Xeon (Westmere) cores and is equipped with an overall of 16 TB of main memory. The nodes are joined by SGI's NUMAlink 5 interconnect."
  - Distributed Shared Memory

Introduction to Message Passing Interface

Message Passing Interface

- Challenge: Distribute one job across many nodes/computers/CPUs

- Concept: Start multiple **processes** (simplest case: same executables):
  - Each process has its own memory
  - Each process has its own core (ideally)
  - Can be on the same machine, can be in the same network

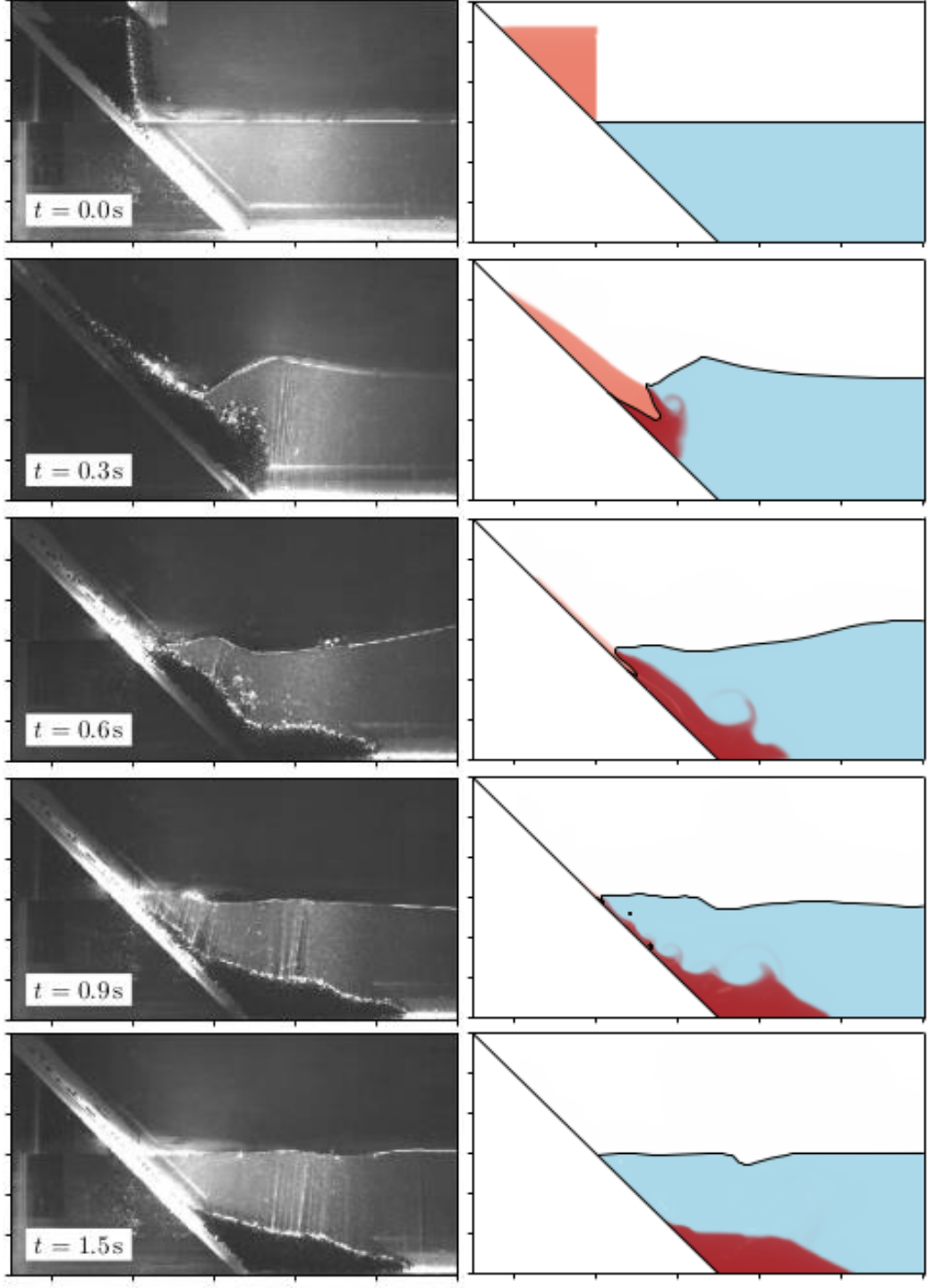
- **New Problem: How to coordinate and exchange information?**

- MPI is a standard (www.mpi-forum.org) and library in form of e.g. openMPI to:
  - Pass memory between processes to
  - If processes are on separate machines: Ethernet or proprietary connections

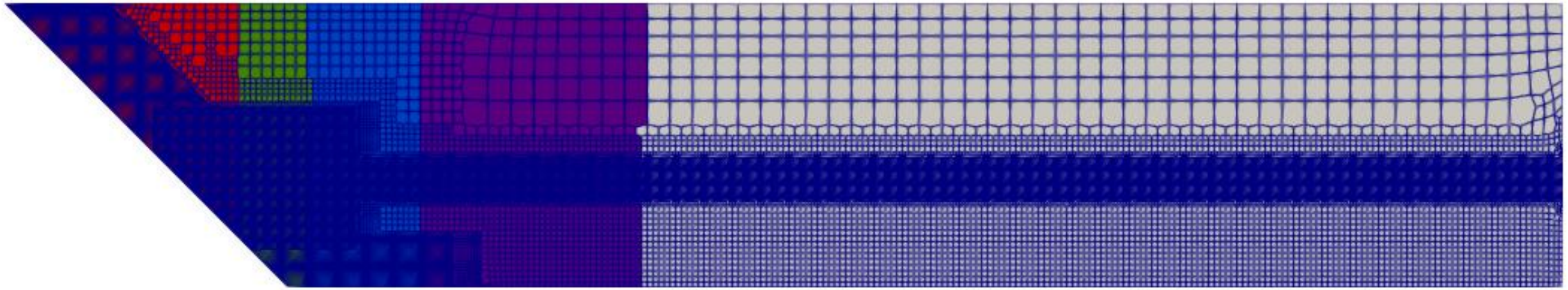
Example: Running distributed OpenFOAM simulation

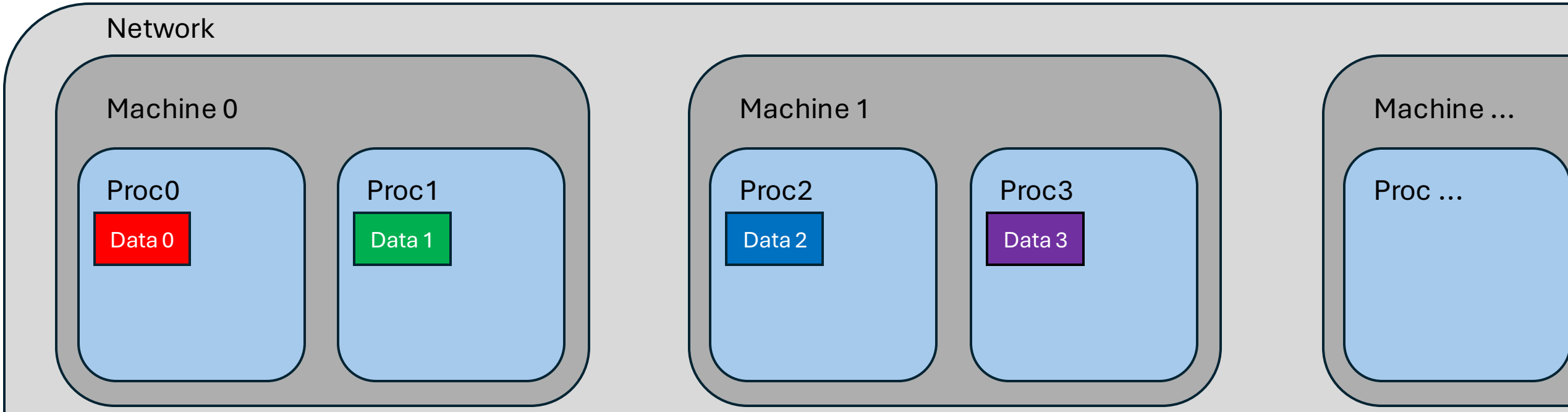
```
# create mesh in 2D
cartesian2DMesh
# copy blank initial fields (mesh independent)
cp -r 0/ 0.orig/
# set initial fields (mesh dependent)
setFields
# split the mesh and the data into (here 8) parts
decomposePar
# run the solver 8 times. npi for communication
mpirun -np 8 multiphaseTsunamiFoam
# stitch the data back together (optional)
reconstructPar
```

If you want to try it use OpenFOAM-v2312/tutorials/incompressible/simpleFoam/airFoil2D



Example: Distributed Data in OpenFOAM:





Prepare dev environment on Windows

In a PowerShell:

```
C:\Users\c0000000> wsl.exe --list --online
C:\Users\c0000000> wsl.exe --install Ubuntu
C:\Users\c0000000> wsl.exe
```

Then in the Linux shell (compiler, development library, mpi executables)

```
User@ZID-T9PCXXXX:~$ sudo apt-get update
User@ZID-T9PCXXXX:~$ sudo apt-get install g++ libopenmpi-dev openmpi-bin
```

Clone the repo and test:

```
User@ZID-T9PCXXXX:~$ git clone
https://github.com/mattisnowman/mpiplayground
User@ZID-T9PCXXXX:~$ ./mpiplayground/cpp/run.sh
```

MPI: Compiling and running on Linux

```
mpicxx -o mpi_pi mpi_pi.c -lm -Wall -std=c++11
# Compile C++. With Math (-lm) and all Warnings (-Wall). C++11 Standard.
# Reads code "mpi_pi.c", writes executable to mpi_pi
# MPI libraries are linked automatically thanks to mpicxx (instead of e.g. g++).
# run "mpicxx -show" to see what mpicxx is doing
```

- mpicxx calls the compiler:
  - -o mpi_pi mpi_pi.c Output and input names.
  - -lm -Wall Link math library, show all warnings
  - -std=c++11 C++11 Standard

```
# Run mpi_pi on 4 nodes.
# Oversubscribe: If there are not enough nodes, run multiple instances on a single
node.
mpirun -np 4 --oversubscribe ./mpi_pi
```

- mpirun –np 4 does two things:
  - Set up MPI environment on 4 nodes.
  - Execute the binary mpi_pi on those 4 nodes.
  - --oversubscribe if there are less than 4 nodes, run more than one binary per node.

And for python, just in case you prefer that…

In a PowerShell:

```
C:\Users\c0000000> wsl.exe --list --online
C:\Users\c0000000> wsl.exe --install Ubuntu
C:\Users\c0000000> wsl.exe
```

Then in the Linux shell (compiler, development library, mpi executables)

```
User@ZID-T9PCXXXX:~$ sudo apt-get update
User@ZID-T9PCXXXX:~$ sudo apt-get install openmpi-bin python3-mpi4py
```

Clone the repo and test:

```
User@ZID-T9PCXXXX:~$ git clone
https://github.com/mattisnowman/mpiplayground
User@ZID-T9PCXXXX:~$ ./mpiplayground/python/run.sh
```

MPI: Running with python on Linux

```
# Run mpi_pi on 4 nodes.
# Oversubscribe: If there are not enough nodes, run multiple instances on a single
node.
mpirun -np 4 --oversubscribe python3 mpi_pi.py
```

- Same as with C++, just run python3 executable with mpi script.

MPI: Getting libraries, config on Rocky Linux.

```
#!/bin/sh

sudo dnf install -y openmpi openmpi-devel

# MPI not in the standard paths on rocky linux? Here is how to add them:
export PATH=/usr/lib64/openmpi/bin:$PATH
export LD_LIBRARY_PATH=/usr/lib64/openmpi/lib/:$LD_LIBRARY_PATH
# Some adjustments. Bug workaround??
export PSM3_DEVICES='self,shm'
```

- Installation on Linux about the same everywhere
- Some paths need to be set manually in the terminal (export):
  - PATH contains all places where Linux searches for executables (+local dir)
  - LD_LIBRARY_PATH contains all shared libraries (*.so)
- `PSM3_DEVICES='self,shm'` a workaround for some intel stuff? (Thanks Eduard Reiterer)

MPI with C/C++: Setup & Boilerplate code

```c
#include <mpi.h>

int main (int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

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

    // code
    // code
    // code...

    MPI_Finalize();
    return 0;
}
```

- Size gives the total number of processes
- Rank gives the index of the current processor
- MPI works with Communicators. MPI_COMM_WORLD is a communicator that includes all processes.
- Remember to MPI_Init and MPI_Finalize.

MPI with C/C++: Hello World!

```cpp
#include <mpi.h>

int main (int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

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

    std::cout << "Hello MPI from Process "
              << rank << "/"
              << size << "!" << std::endl;

    MPI_Finalize();
    return 0;
}
```

1. Compile the code (mpicxx hello.cpp -o hellp)
2. Run it normally (./hello)
3. Run it with MPI (mpirun –np [2,4,8,16] ./hello)

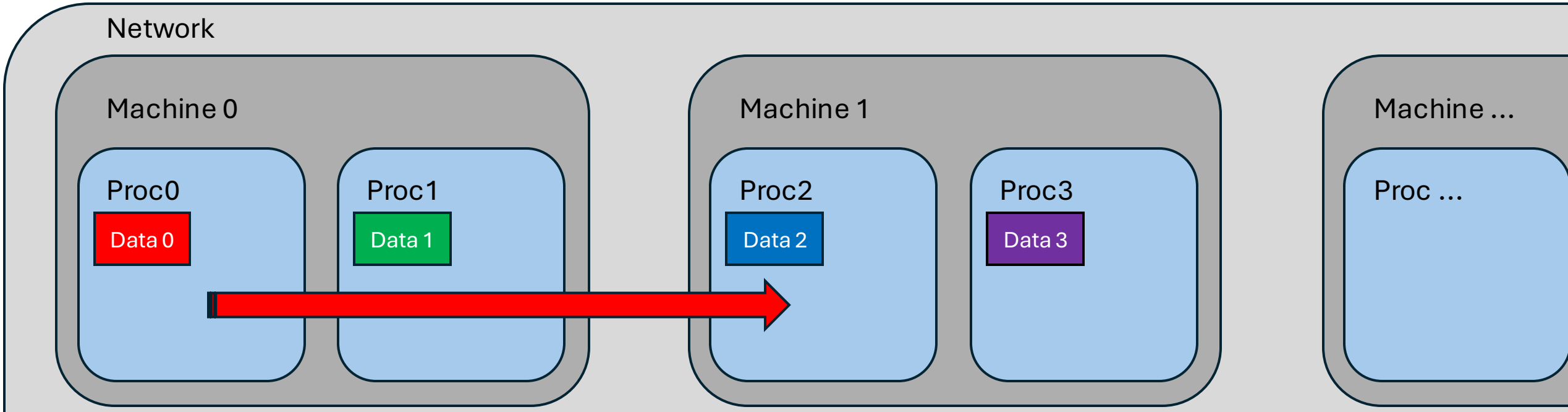
1. What happens?
2. Can you make sense out of it?

MPI_Send: Send local data to another node

```
int MPI_Send(const void *buf,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm)
```

www.mpich.org/static/docs/v4.1/www3/MPI_Send.html
www.mpich.org/static/docs/v4.1/www3/Constants.html

- buf: data to be send
- count: how big buf is
- datatype: e.g. MPI_DOUBLE
- dest: target node

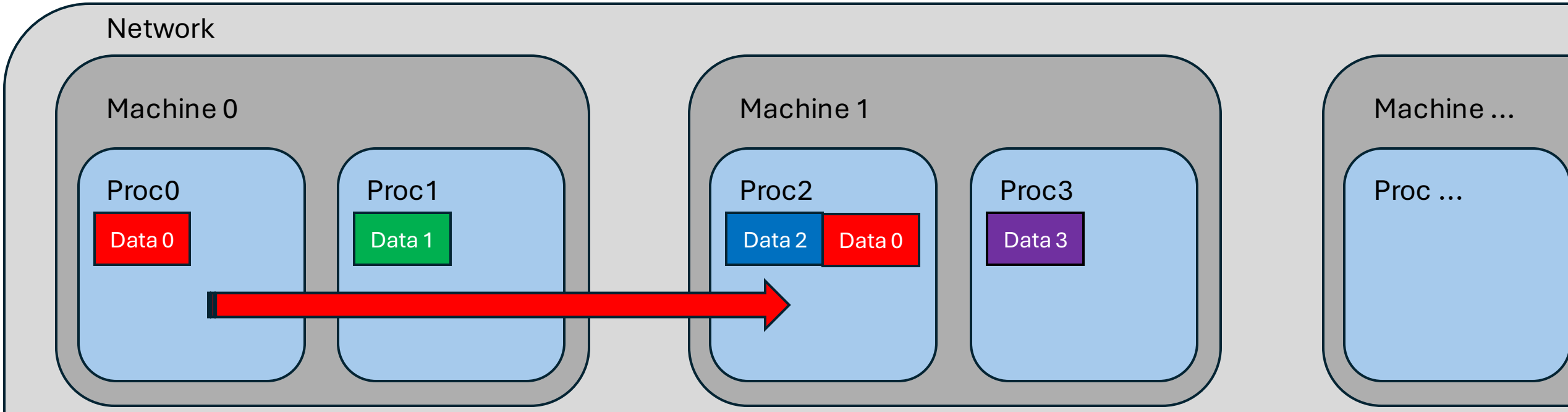


MPI_Recv: Receive data from another node

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

www.mpich.org/static/docs/v4.1/www3/MPI_Recv.html

- buf: here goes the data
- count: how big buf is
- datatype: e.g. MPI_DOUBLE
- source: sender node



MPI_Send and MPI_Recv: Minimal example

Write an example that creates a random number in processor 0 and sends it to processor 2. Print it on processor 2. Start with the boilerplate. Here are some tips:

Check if you are on proc with ID:

```
if (rank == ID) {
    //code
}
```

Get a random integer:

```
value = std::rand();
```

Print stuff:

```
std::cout << "value is "
    << value << std::endl;
```

MPI_SEND for a single int value:

```
MPI_Send(&value, 1, MPI_INT,
dest_ID, 0, MPI_COMM_WORLD);
```

*buf = &value, count = 1, type = MPI_INT,
dest = dest_ID, tag = 0

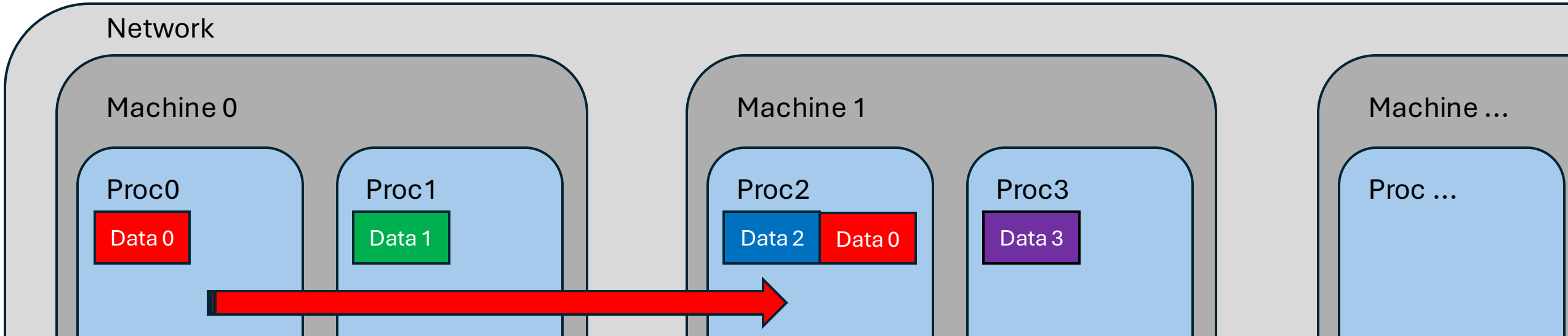
MPI_RECV for a single int value:

```
MPI_Recv(&value, 1, MPI_INT,
MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, 0);
```

*buf = &value, count = 1, type = MPI_INT,
source = whatefver, tag = whatever,
STATUS = I don't care.

MPI_Send and MPI_Recv: Minimal example

```
int source_ID = 0;
int dest_ID = 2;
int value;
if (rank == source_ID) {
    value = std::rand();
    std::cout << "proc " << rank << ": sending " << value << " to proc " << dest_ID << std::endl;
    MPI_Send(&value, 1, MPI_INT, dest_ID, 0, MPI_COMM_WORLD);
} else if (rank == dest_ID) {
    MPI_Recv(&value, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, 0);
    std::cout << "proc " << rank << ": received " << value << " from proc " << source_ID <<
std::endl;
}
```

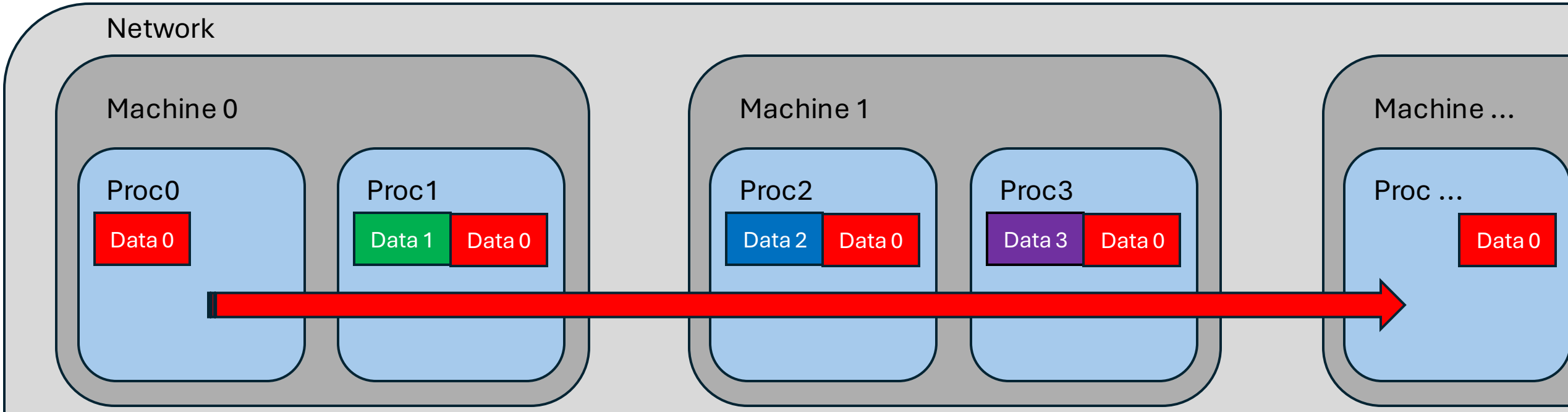


MPI_Bcast: 1 to N communication

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

- sendbuf: here goes the c
- count: how big *buf is
- datatype: e.g. MPI_DOUBLE
- root: sender node, all others are receiver

www.mpich.org/static/docs/v4.1/www3/MPI_Bcast.html

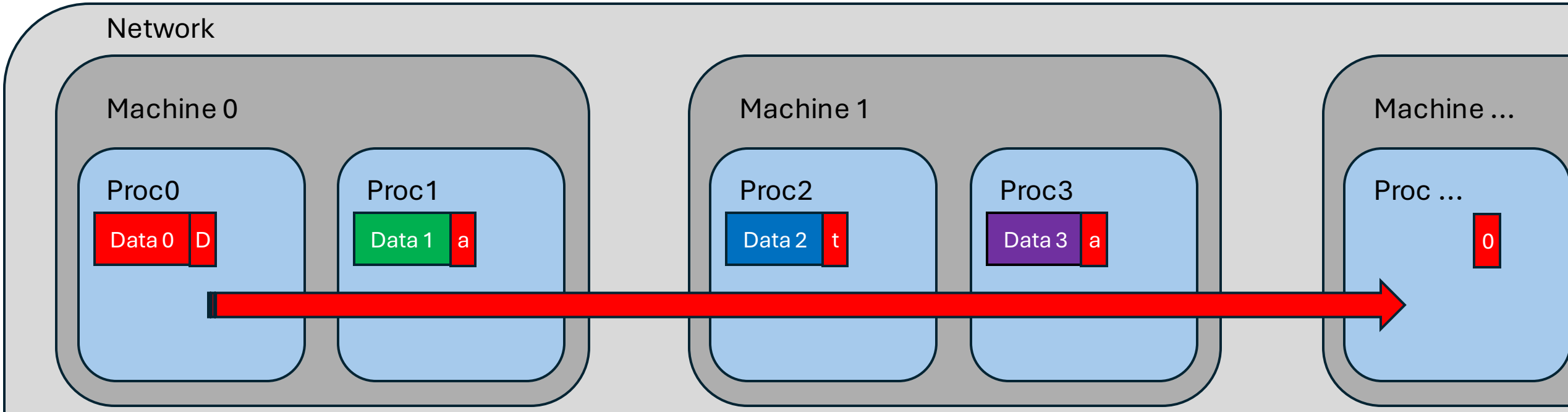


MPI_Scatter: Also 1 to N communication

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

- sendbuf: sending data
- recvbuf: receiving data
- root: sender node

www.mpich.org/static/docs/v4.1/www3/MPI_Scatter.html

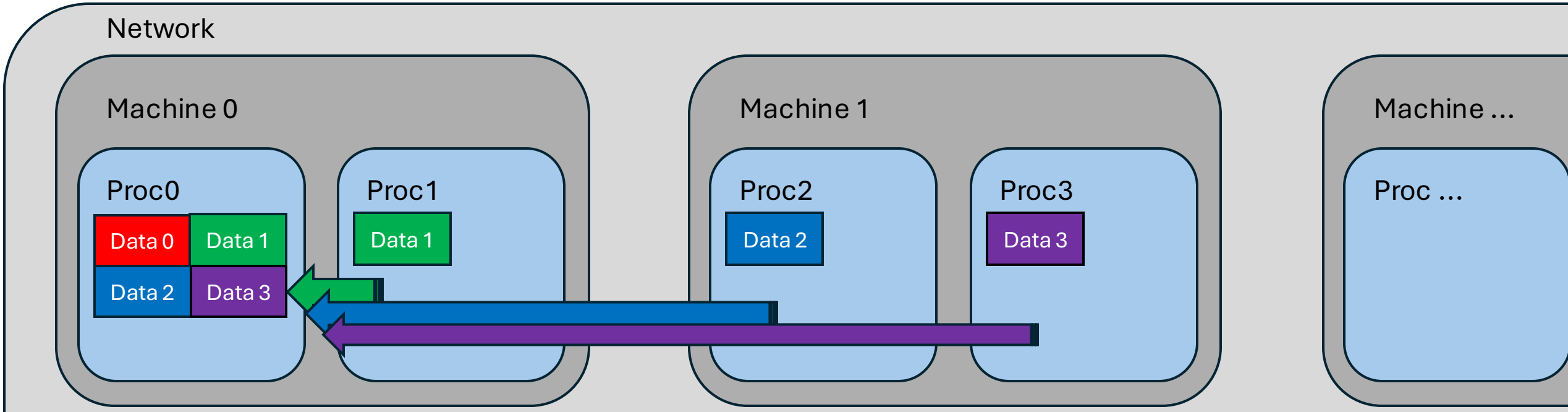


MPI_Gather: N to 1 communication

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

www.mpich.org/static/docs/v4.1/www3/MPI_Gather.html

- sendbuf: data to be send
- Recvbuf: data to be written
- root: receiver node, all others are sender

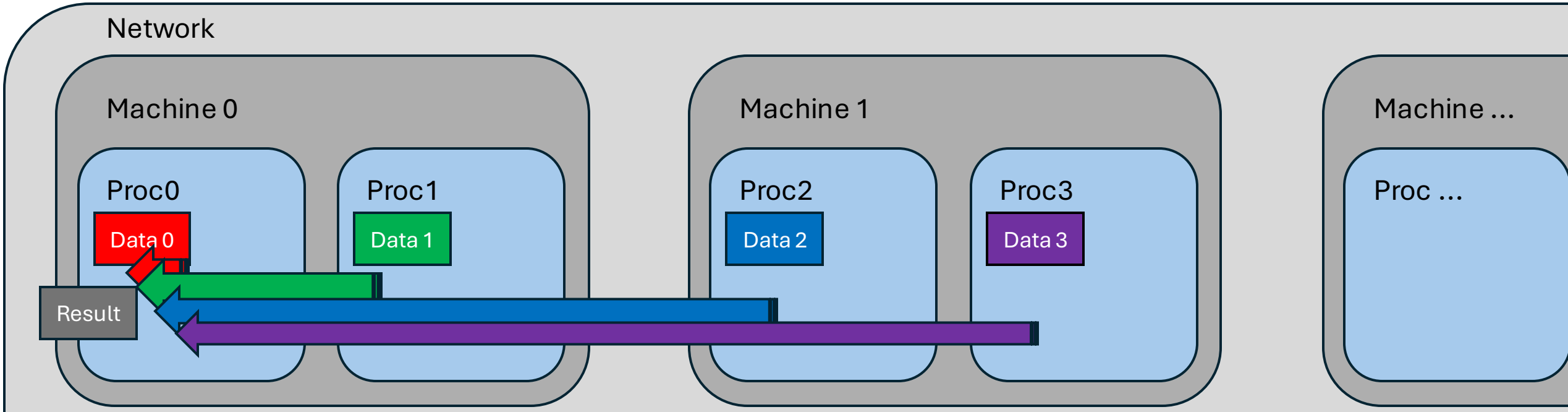


MPI_Reduce: Do an operation while gathering

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

- sendbuf: data to be send
- recvbuf: data to be written
- Op: reduce operation, e.g. MPI_MAX
- root: receiver node

www.mpich.org/static/docs/v4.1/www3/MPI_Reduce.html
www.mpich.org/static/docs/v4.1/www3/Constants.html



What else is there in MPI?

- All to all communication: MPI_Allgather, MPI_Allreduce
- Syncronize: MPI_Barrier (wait in all processes)
- Get time: MPI_Wtime
- Split Communicator: MPI_Comm_split, MPI_Comm_join, …
- Register your own type: MPI_Type_create_struct, MPI_Type_commit
- Non-blocking functions: MPI_Isend, MPI_Irecv
- File I/O (less common): MPI_File_open, MPI_File_set_view, …
- Much, much more…

Proof to me that this really is how it is done…

- Foam::Sum(field) is calculating the sum of a field on a single node.
- Foam::gSum(field) is calculating the sum of a field **on all nodes.**

```
matti@tsunami:~/OpenFOAM/OpenFOAM-v2312$ grep -nr "Foam::gSum"
...
modules/external-solver/src/petsc4Foam/utils/petscUtils.C:64:
Foam::solveScalar Foam::gSum(Vec input)
```

```
Foam::solveScalar Foam::gSum(Vec input)
{
    PetscScalar val;
    AssertPETSc(VecSum(input, &val));
```

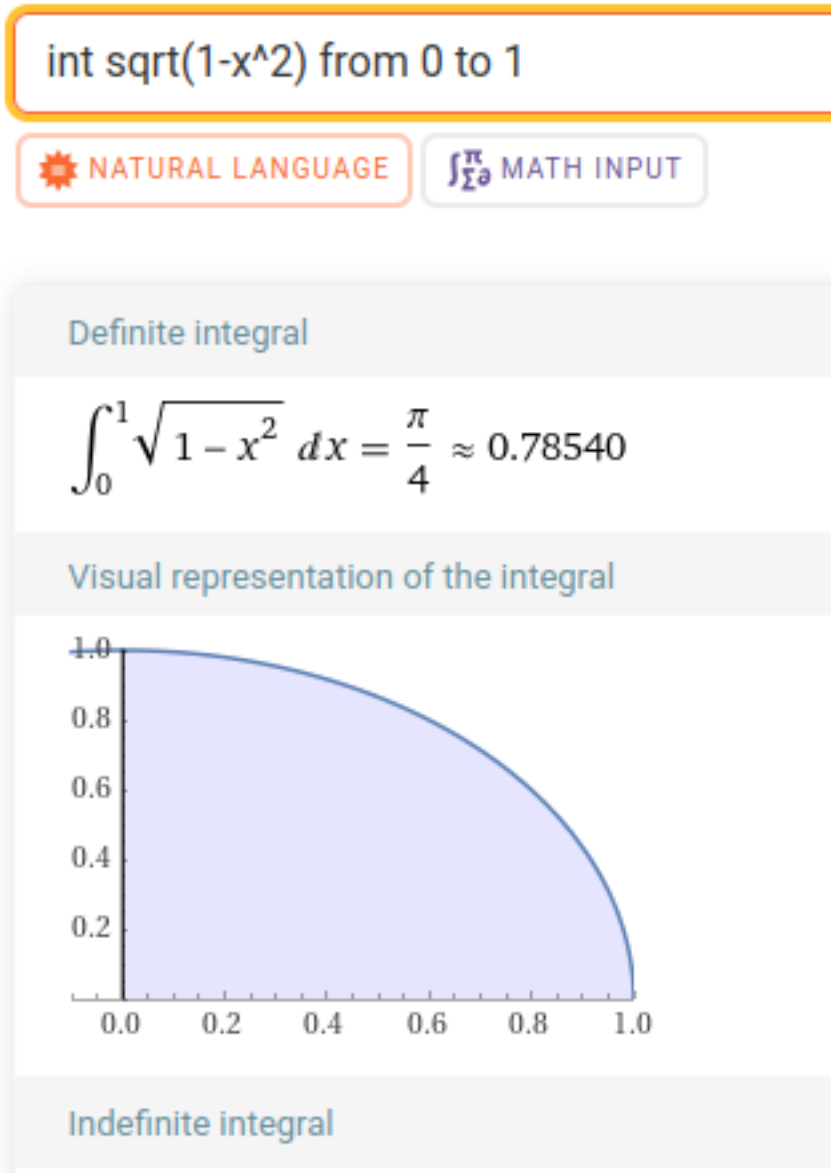
```
PetscErrorCode VecSum(Vec v, PetscScalar *sum)
{
    PetscScalar tmp = 0.0;
    ...
    [pseudocode: x = v]

    for (PetscInt i = 0; i < n; ++i) tmp += x[i];
    ...
    PetscCallMPI(MPIU_Allreduce(MPI_IN_PLACE, &tmp, 1, MPIU_SCALAR, MPIU_SUM,
                 PetscObjectComm((PetscObject)v)));
    *sum = tmp;
    PetscFunctionReturn(PETSC_SUCCESS);
}
```

Do GPUs (NVLink, Cuda) replace MPI?

"MPI is fully compatible with CUDA, CUDA Fortran, and OpenACC, all of which are designed for parallel computing on a single computer or node. There are a number of reasons for wanting to combine the complementary parallel programming approaches of MPI & CUDA (/CUDA Fortran/OpenACC)."

https://developer.nvidia.com/mpi-solutions-gpus

MPI Exercise: Distributed calculation of PI



- We can calculate PI with numerical integration of a quarter circle ($f(x) = \sqrt{1-x^2}$)
- Split the interval [0, 1] in N = 1e7 equal parts. Calculate their area (1/N * f(x)). Distribute the parts equally to all processes (easy option: for (int i = rank; i < N; i += size)).
- Develop 2 versions:
  - Gather/reduce results in every step of the for loop.
  - Gather/reduce results after the loop. Use a temporary variable within the loop.
- Run your examples with [1,2,4,8,16,32] nodes. How do they scale with number of nodes? Can you make sense of it? How would you describe the difference between the applications?

MPI Exercise: Tips

Add this to the beginning/end to get execution duration:

```
MPI_Barrier(MPI_COMM_WORLD);
const double begin = MPI_Wtime();

// do stuff

MPI_Barrier(MPI_COMM_WORLD);
const double end = MPI_Wtime();
if (!rank)
    std::cout << "Procs = " << size
              << ", Exec. duration = "
              << end - begin;
```

This does not include e.g. creating of processes. For that you have to use Linux bash commands, but this is good enough.

This is a very easy way to distribute over size processes:

```
for (int i = rank; i <= N; i += size)
{
    double x = from + i * d;
    double fx = f(x);
    part_sum += fx * d;
}
```

The two versions should have MPI_Reduce in:

```
for (int i = rank; i <= N; i += size){
    // do stuff
    MPI_Reduce(...);
}
```

And outside the loop (different granularity):

```
for (int i = rank; i <= N; i += size){
    // do stuff
}
MPI_Reduce(...);
```