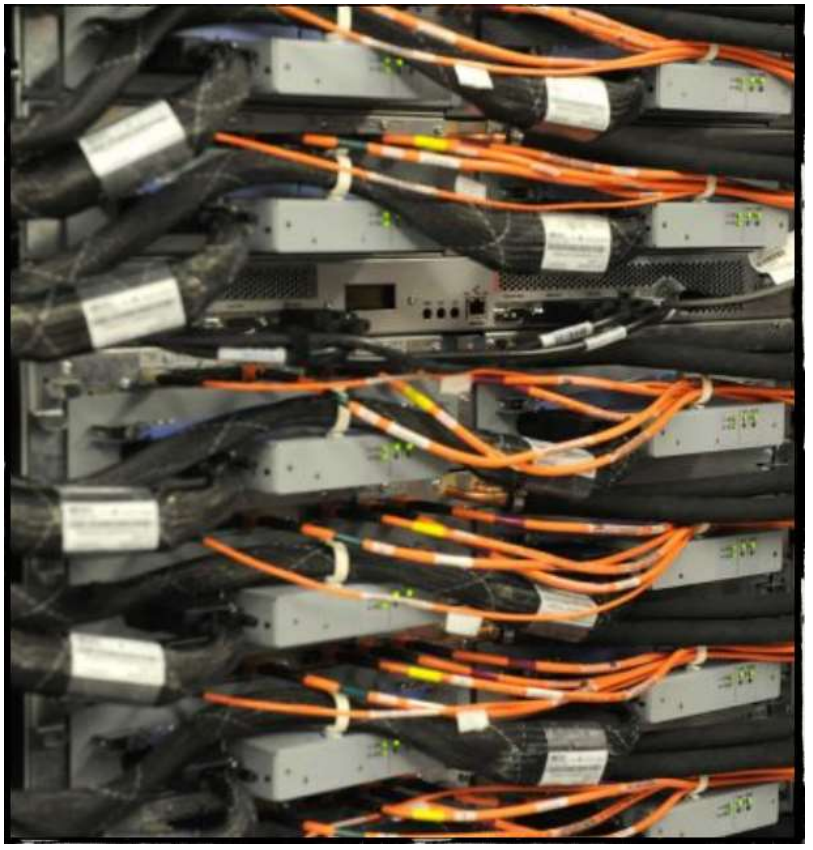
Lecture 11: Parallel computing

- Introduction to parallel computing
- Introduction to parallel computing
- Introduction to parallel computing
- Introduction to parallel computing

- OpenMP vs MPI

- Job queue, job submission

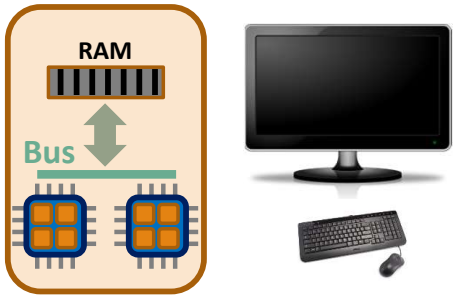


A parallel computer

symmetric multiprocessor (SMP) system

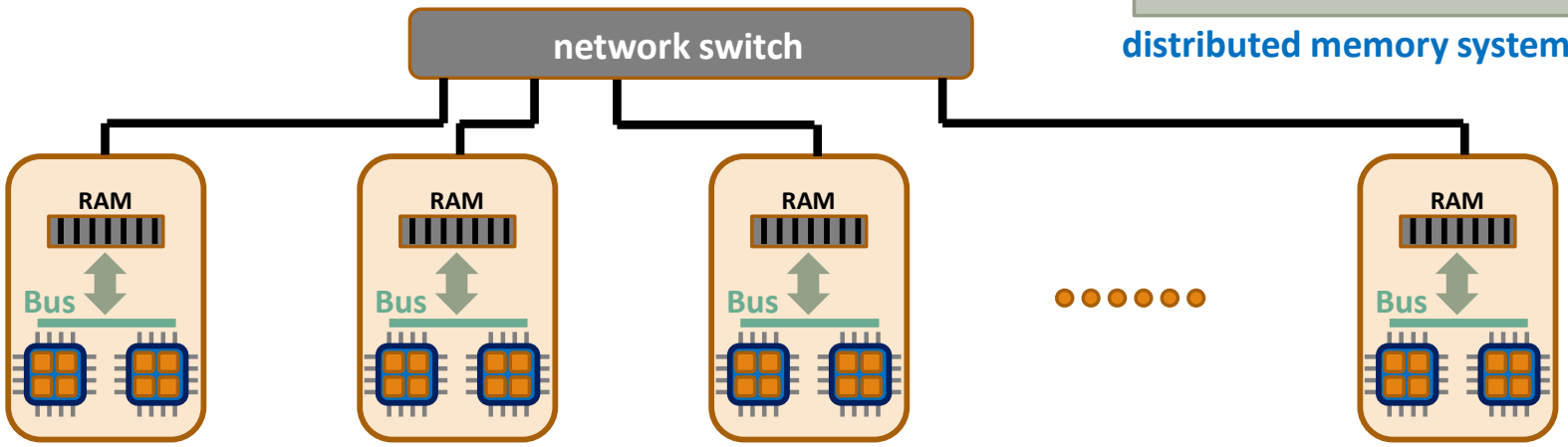
or

shared memory system



Each node can also have one or more graphics processing units (GPUs).

distributed memory system



The most powerful supercomputer (as of today) consists of 8,699,904 cores and has 4 times as many GPUs than CPUs

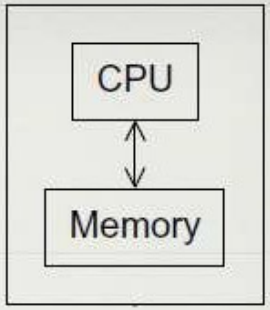
TOP 500
The List.

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107
6	**Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Swiss National Supercomputing Centre (CSCS) Switzerland	1,305,600	270.00	353.75	5,194

Parallel computing methods/interfaces

- **OpenMP** is a set of directives (add-ons to a compiler), libraries and environment variables, which allows to program for shared memory systems.
- Message Passing Interface (**MPI**) is a standardized library allowing users to write parallel programs for both shared-memory and distributed-memory systems.
  - MPI is currently the standard for using nearly all supercomputers.
  - MPI is available for C, C++, and Fortran 77/90/95, and Python
  - MPI is a standard, not an implementation. There are a variety of implementations, some are available for free (OpenMPI, MVAPICH2, …), some are provided by hardware vendors (Intel, IBM, …)
- **CUDA** is a special interface/library for using NVIDIA GPUs.

Notion of process



- In message-passing programs, a program running on one core-memory pair is called a process
- Two processes can communicate by calling functions (*e.g.*, one process calls a send function and the other calls a receive function.)
- The implementation of message-passing that we'll be using is called MPI, which is an abbreviation of Message-Passing Interface.

MPI

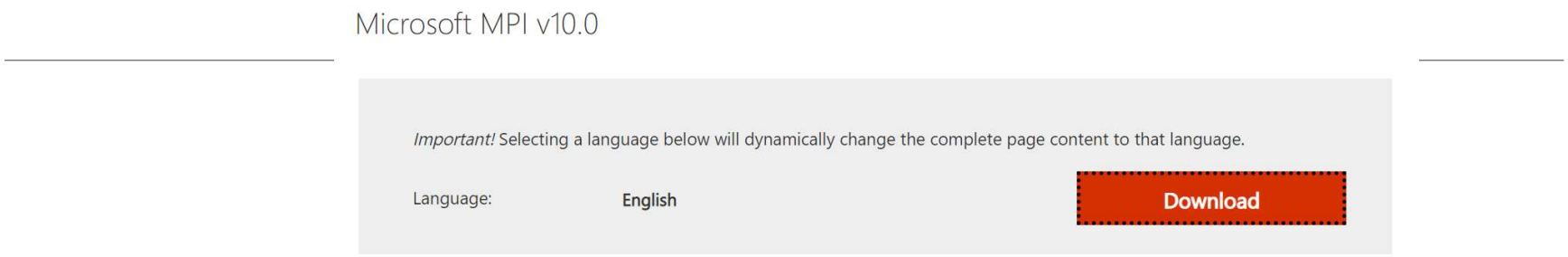
- MPI is not a new programming language.
- It defines a library of functions that can be called from C++, C, and Fortran programs.
  - We'll learn about some of MPI's different send and receive functions.
  - We'll also learn about some "global" communication functions that can involve more than two processes. (These functions are called collective communications.)
- In the process of learning about all of these MPI functions, we'll also learn about
  - data partitioning
  - I/O in distributed-memory systems.

Running MPI

- MPI is a library, you can use it on any number machines, including
  - your own laptop/desktop
  - supercomputer
  - cluster
- OpenMPI is a free library you can install for your Linux, Max, or Windows Machine

MPI on your laptop

Windows environment



Linux (Debian/Ubuntu environment)

sudo apt install package_name

execute command as superuser

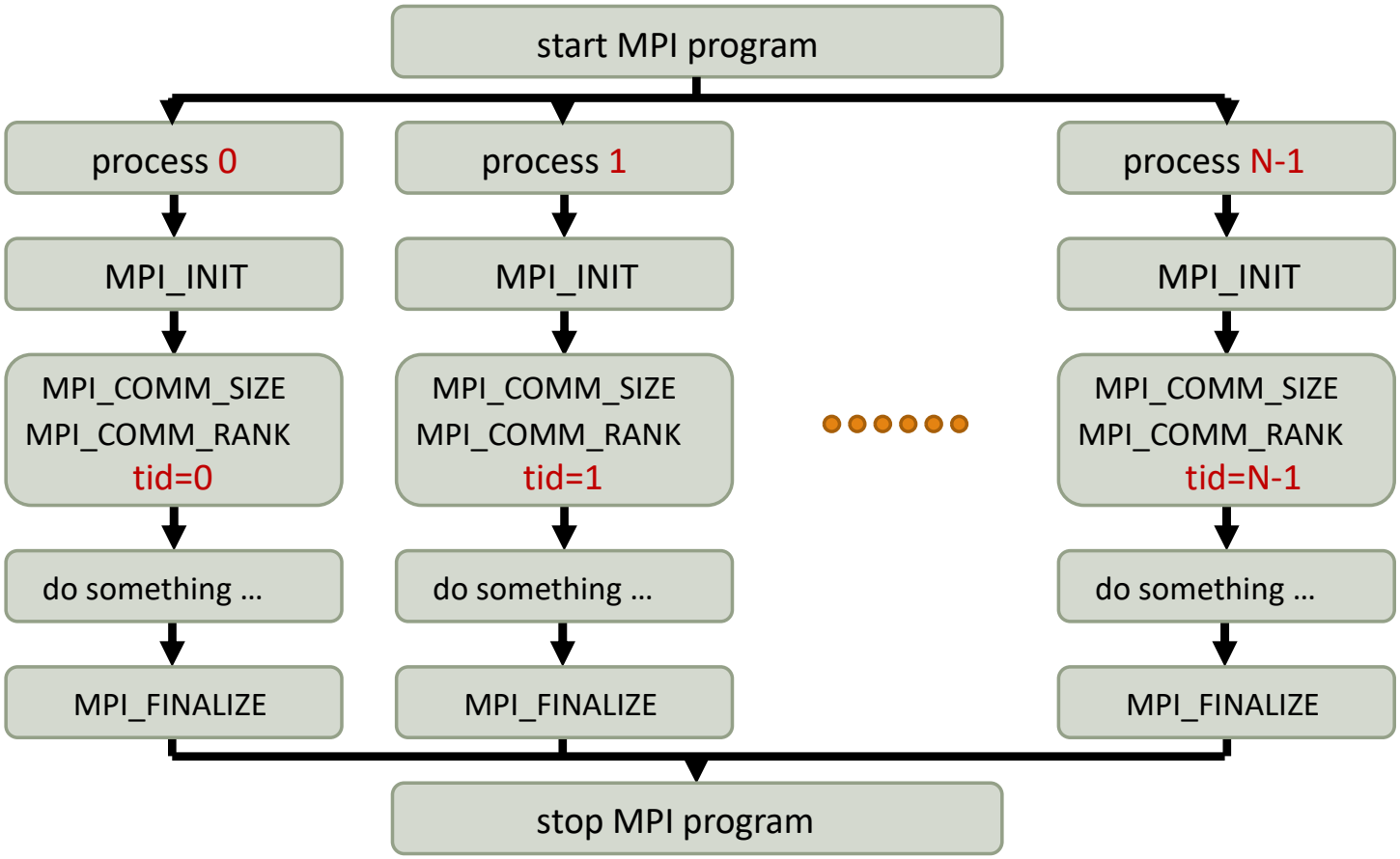
package manager

Install openmpi:

sudo apt install openmpi-bin
sudo apt install libopenmpi-dev
pip install mpi4py

Looks for installable packages with “package_name” in the **respositories.**

How an MPI program gets excecuted?



Python example

mpi4py python library

```
PS C:\Users\damien> pip3 install mpi4py
Collecting mpi4py
  Using cached mpi4py-3.0.3-cp38-cp38-win_amd64.whl (440 kB)
Installing collected packages: mpi4py
Successfully installed mpi4py-3.0.3
PS C:\Users\damien>
```

Simple MPI program

```
from mpi4py import MPI

size=MPI.COMM_WORLD.Get_size()
rank=MPI.COMM_WORLD.Get_rank()
name=MPI.Get_processor_name()

print("Hello World! I am rank ",rank, "out of ",size, " on ",name)
```

Running on 2 cores:

```
PS C:\Users\damien> mpiexec.exe -n 2 python.exe .\test.py
Hello World! I am rank  0 out of  2  on  LAPTOP-HFD43PLU
Hello World! I am rank  1 out of  2  on  LAPTOP-HFD43PLU
```

Running on 4 cores:

```
PS C:\Users\damien> mpiexec.exe -n 4 python.exe .\test.py
Hello World! I am rank  0 out of  4  on  LAPTOP-HFD43PLU
Hello World! I am rank  3 out of  4  on  LAPTOP-HFD43PLU
Hello World! I am rank  1 out of  4  on  LAPTOP-HFD43PLU
Hello World! I am rank  2 out of  4  on  LAPTOP-HFD43PLU
```

Job server (slurm job server)

sinfo: list of job queues and available nodes

```
[damien@matisse ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
temprush     up 14-06:00:0      0    n/a
defq*        up    6:00:00      4  down* node[002,006-008]
defq*        up    6:00:00      4   idle node[001,003-005]
q064         up 2-00:00:00      7  down* node[009,011-012,014,016-018]
q064         up 2-00:00:00      5   idle node[010,013,015,019-020]
q128         up    8:00:00      0    n/a
```

squeue: current job queue, list of running and waiting jobs.

```
[damien@matisse ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
             88347      defq PYRIDINE   grayj6 PD       0:00      1 (PartitionTimeLimit)
```

sbatch: submit a job to the queue using a script. The script specifies how many nodes/cores to run on, which queue to submit to, time allowed to run, output file name, etc.

run using, sbatch myscript

scancel: cancels a submitted job with a specified JobID.

example of batch script

```
#!/bin/bash -x
#SBATCH -J myrun
#SBATCH --export=ALL
#SBATCH -p QACP
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH -t 1:00:00
#SBATCH -o stdout.txt
#SBATCH -e stderr.txt
#SBATCH --mail-type=ALL
#SBATCH --mail-user=xyz@rpi.edu
EXE_FILE=./a.out
srun hostname -s | sort -n > hosts.$SLURM_JOB_ID
NPROCS=2
mpirun -hostfile hosts.$SLURM_JOB_ID -np $NPROCS $EXE_FILE > out
rm hosts.$SLURM_JOB_ID
```

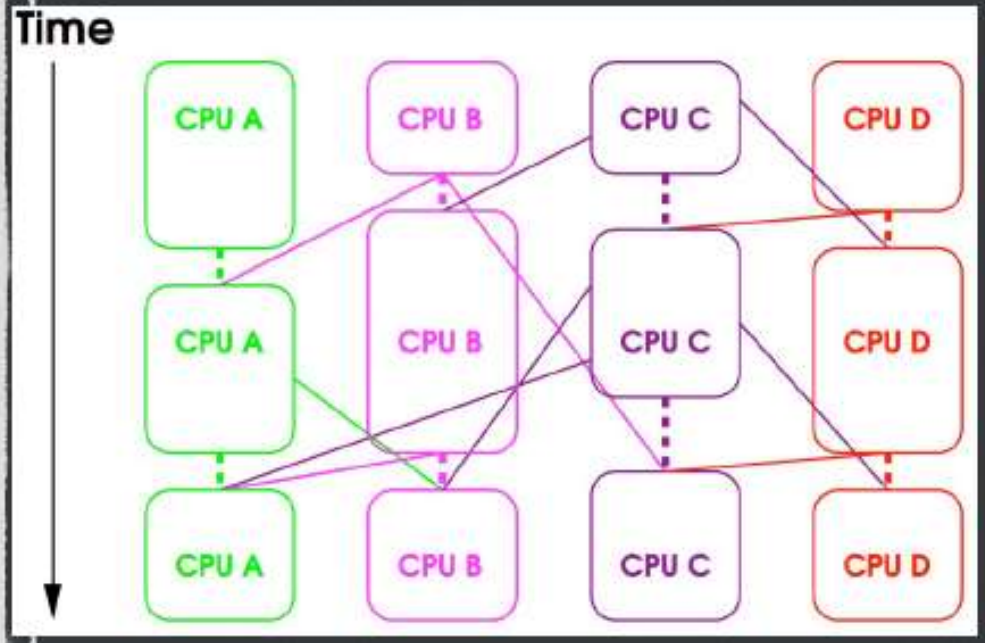
do not changed these lines

The MPI MODEL

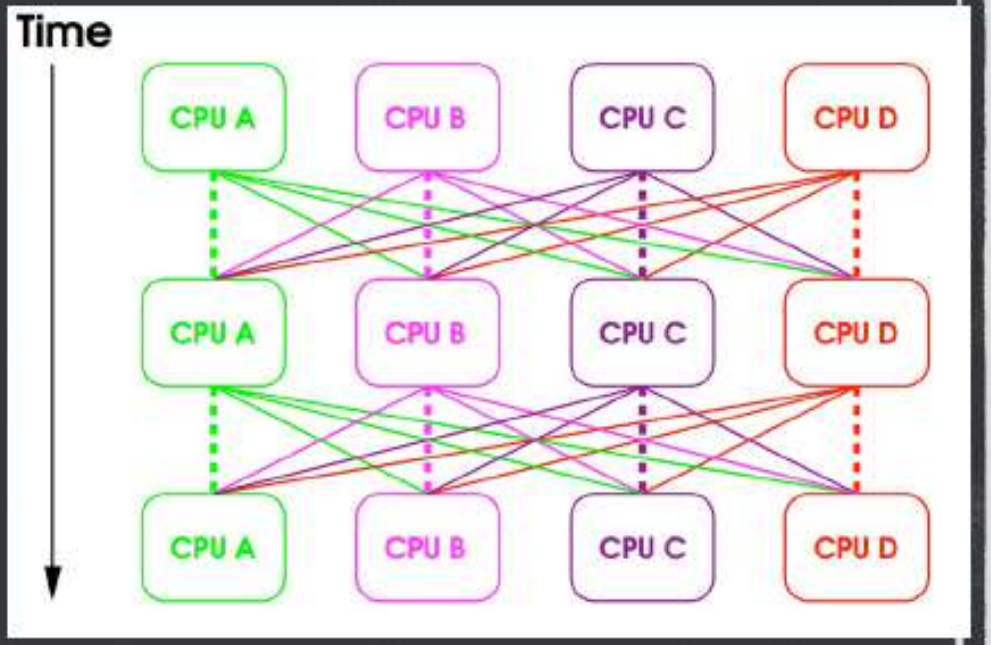
- You start up N independent processes
  - All of them start MPI and use it to communicate
- There is no "master" (initial or main process)
- Communications may be "point-to-point"
  - Only two communicating processes are involved
- Communications may be "collective"
  - All of the processes are involved
    - They must all make the same call, together
-

TYPES OF COMMUNICATION

PROCESS-TO-PROCESS

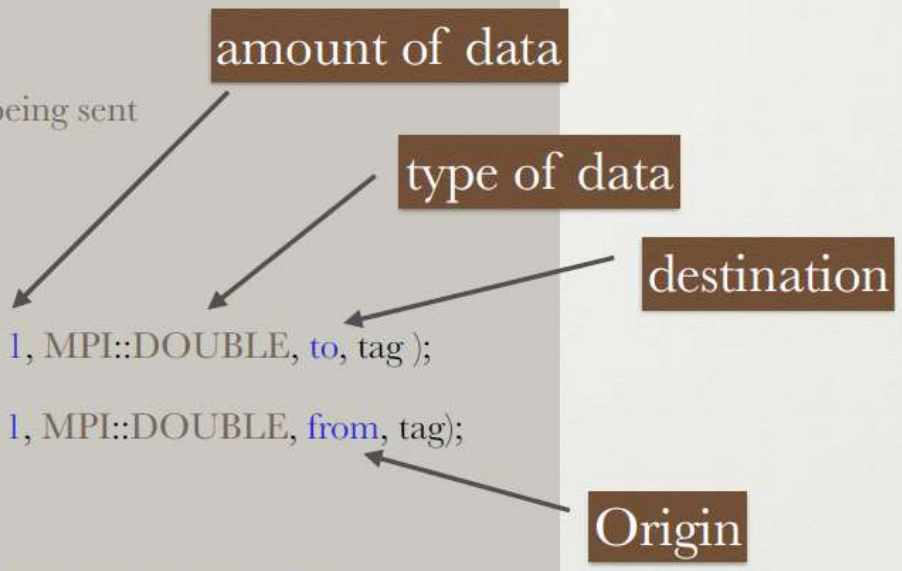


COLLECTIVE



sending and receiving messages

- Communication between processors involves:
  - identify sender and receiver
  - the type and amount of data that is being sent
- Notice the difference in referencing:
  - Scalar
    - MPI::COMM_WORLD.Send (&x, 1, MPI::DOUBLE, to, tag);
    - MPI::COMM_WORLD.Recv (&x, 1, MPI::DOUBLE, from, tag);
  - Vector
    - MPI::COMM_WORLD.Send (y, 10, MPI::DOUBLE, to, tag);
    - MPI::COMM_WORLD.Recv (y, 10, MPI::DOUBLE, from, tag);



standard send: syntax

```
COMM_WORLD.Send (buf, count, datatype, dest, tag)
```

buf is the name of the array/variable to be broadcasted
count is the number of elements to be sent
datatype is the type of the data
dest is the rank of the destination processor
tag is an arbitrary number which can be used to distinguish among messages

standard receive: syntax

```
COMM_WORLD.Recv (buf, count, datatype, source, tag)
```

source is the rank of the processor from which data will be accepted
(this can be the rank of a specific processor or a wild card- MPI::ANY_SOURCE)
tag is an arbitrary number which can be used to distinguish among messages
(this can be a wild card: MPI::ANY_TAG)

Potential pitfalls:
Hanging programs is the main enemy to the beginner programmer!

- Note that the semantics of MPI Recv suggests a potential pitfall in MPI programming:
- If a process tries to receive a message and there's no matching send, then the process will block forever.
- When we design our programs, we therefore need to be sure that every receive has a matching send.
- Perhaps even more important, we need to be very careful when we're coding that there are no inadvertent mistakes in our calls to MPI Send and MPI Recv.
  - For example, if the tags don't match, or if the rank of the destination process is the same as the rank of the source process, the receive won't match the send, and either a process will hang, or, perhaps worse, the receive may match another send.

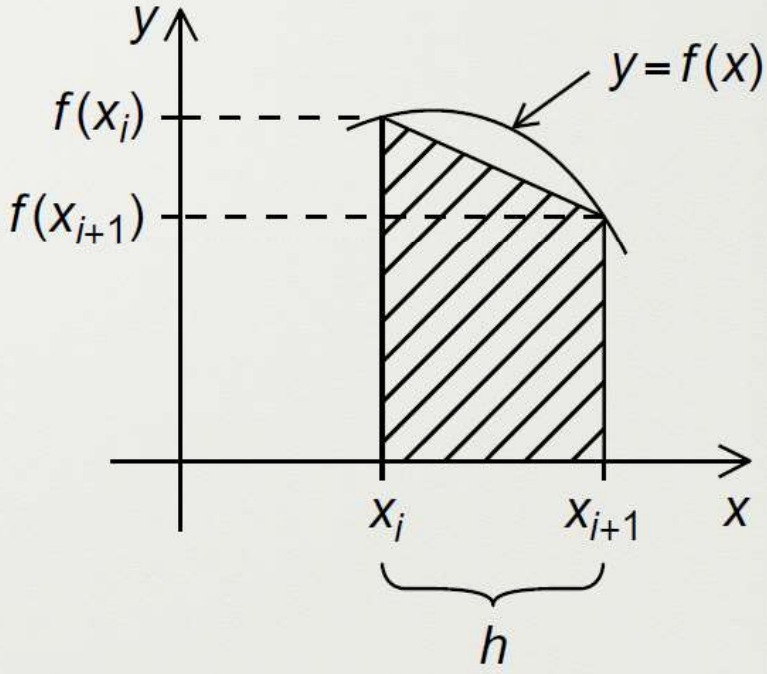
Trapezoid rule Serial version

- Since we chose the n subintervals so that they would all have the same length, we also know that if the vertical lines bounding the region are $x = a$ and $x = b$, then

$$x_0 = a,\ x_1 = a + h,\ x_2 = a + 2h, \ldots,\ x_{n-1} = a + (n-1)h,\ x_n = b,$$

- Thus, if we call the leftmost endpoint x_0, and the rightmost endpoint x_n, we have

$$h = \frac{b-a}{n}.$$



- and the sum of the areas of the trapezoids —our approximation to the total area — is

$$h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2].$$

$$\int_0^{\pi} \sin(x)\, dx$$

Trapezoid Method (serial)

```
import numpy as np

def f(x):
    return np.sin(x)

a=0
b=np.pi

xs=np.linspace(a,b,1000)
h=xs[1]-xs[0]

integral = h*(f(xs).sum() - f(xs[0])/2.0 - f(xs[-1])/2.0)
print("Integral sin(x) over [0,pi] = ",integral)
```

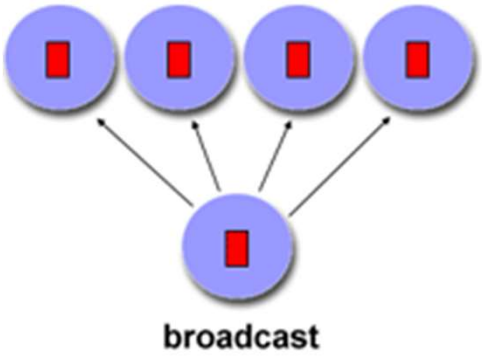
Trapezoid Method (parallel)

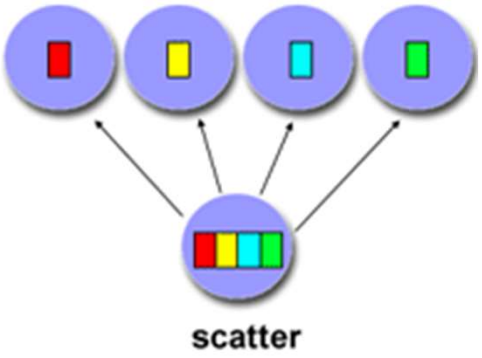
```python
from mpi4py import MPI
import numpy as np

comm =MPI.COMM_WORLD
size=comm.Get_size()
rank=comm.Get_rank()
name=MPI.Get_processor_name()


def f(x):
    return np.sin(x)

a=0
b=np.pi
N=1000

alocal= rank*(b-a)/size   # local a (for process)
blocal= (rank+1.0)*(b-a)/size # local b (for process)

xs=np.linspace(alocal,blocal,int(1000/size))
h=xs[1]-xs[0]

integral=h*(f(xs).sum() - f(xs[0])/2.0 -f(xs[-1])/2.0)

print("Integral from process ",rank," is ",integral)
```

```
PS C:\Users\damien> mpiexec.exe -n 2 python .\test.py
Integral from process  0  is  0.999999174233066
Integral from process  1  is  0.999991742330914
```

Collective communication

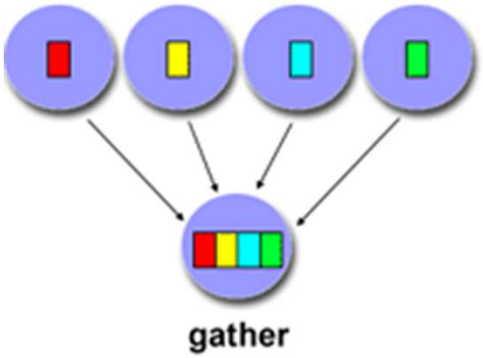
MPI_BCAST

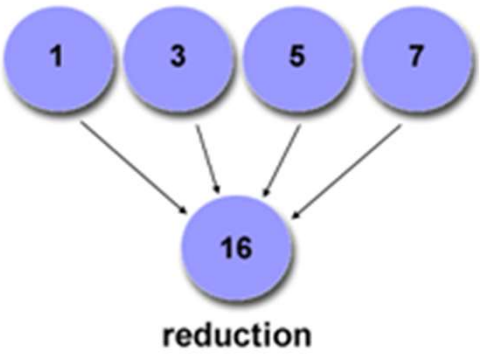




MPI_SCATTER

MPI_GATHER





MPI_REDUCE

https://computing.llnl.gov/tutorials/mpi/

```python
from mpi4py import MPI
import numpy as np

comm =MPI.COMM_WORLD
size=comm.Get_size()
rank=comm.Get_rank()
name=MPI.Get_processor_name()


def f(x):
    return np.sin(x)

a=0
b=np.pi
N=1000

alocal= rank*(b-a)/size   # local a (for process)
blocal= (rank+1.0)*(b-a)/size # local b (for process)

xs=np.linspace(alocal,blocal,int(1000/size))
h=xs[1]-xs[0]

integral=h*(f(xs).sum() - f(xs[0])/2.0 -f(xs[-1])/2.0)

print("Integral from process ",rank," is ",integral)

integral_total = comm.reduce(integral, op=MPI.SUM)

if (rank==0):
    print("Total value of the integral is: ",integral_total)
```

```
PS C:\Users\damien> mpiexec.exe -n 2 python .\test.py
Integral from process  1  is  0.9999991742330914
Integral from process  0  is  0.999999174233066
Total value of the integral is:  1.9999983484661574
```

Monte-Carlo Integration

$$\int_0^1 \int_0^1 \int_0^1 \dots \int_0^1 f(x_1, x_2, x_3, \dots, x_n) dx_1 dx_2 dx_3 \dots dx_n = V\langle f \rangle$$

multidimensional "volume"

Average value of $f(x_1, x_2, \dots, x_n)$ over integration region

Randomly select point in integration volume to determine the average value of f, $\langle f \rangle = \frac{1}{N} \Sigma_{\mathrm{i}=1}^{\mathrm{N}} f(\boldsymbol{x_i})$

Error $\propto \frac{\sigma^2}{\sqrt{N}}$ (average determined by N random points)

Easy to parallelize