Universität Zürich[UZH]

MPI II

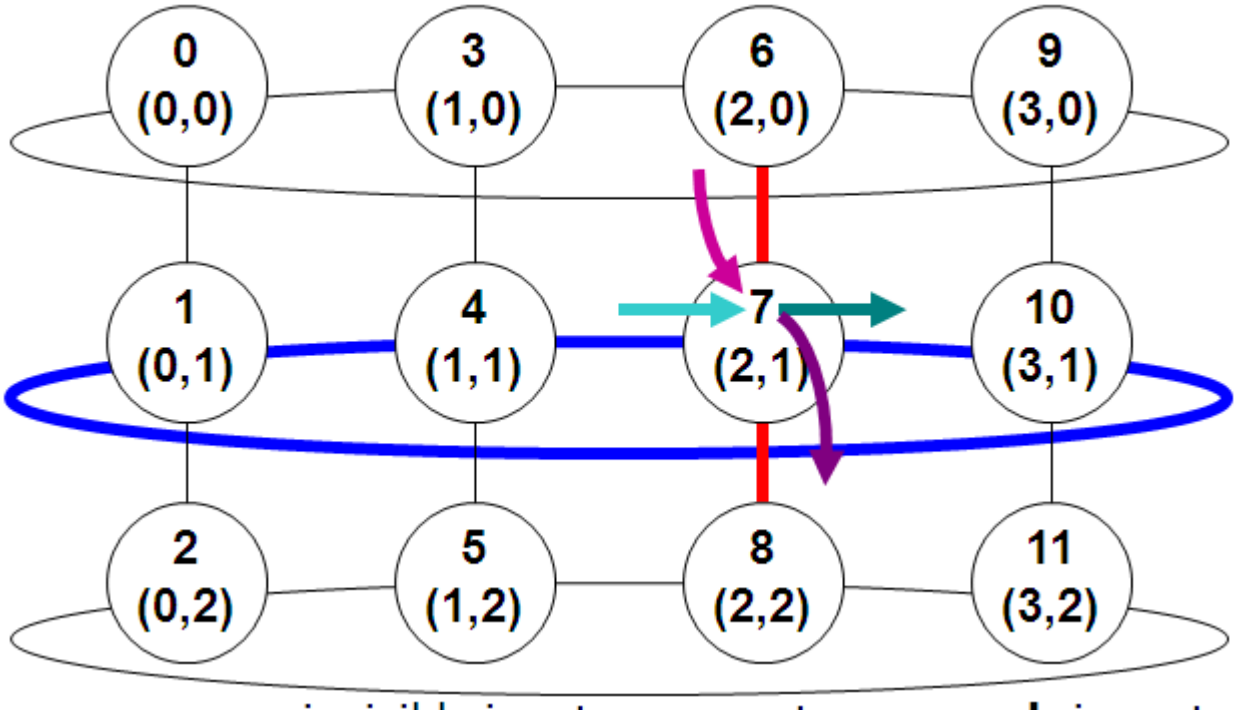
Simon Scheidegger
simon.scheidegger@gmail.com
August 1st, 2017
Open Source Macroeconomics Laboratory – BFI/UChicago

Including adapted teaching material from books, lectures and presentations by
B. Barney, G. Hager, M. Martinasso, R. Rabenseifner, O. Schenk, G. Wellein

Outline

1. Topology for managing rank numbering
2. User specific data type
3. Parallel I/O
4. Python & MPI

1. Topology for managing rank numbering.



MPI_Groups

$$V^s(k,\theta) = \max_{I,c,l} u(c,l) + \beta \mathbb{E}\Big\{ V^{s-1}(k^{s-1},\theta^{s-1}) | \theta \Big\},$$

$$s.t.$$

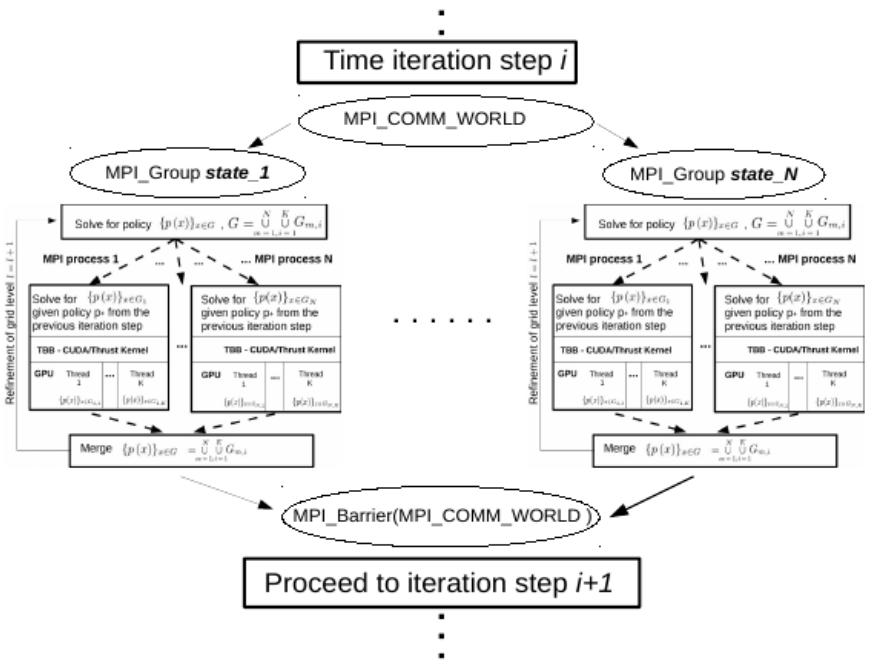
$$k_j^{s-1} = (1-\delta)\cdot k_j + I_j + \epsilon_j, \quad j = 1,...,d$$

$$\Gamma_j = \frac{\zeta}{2} k_j \left(\frac{I_j}{k_j} - \delta \right)^2, \quad j = 1,...,d$$

$$\sum_{j=1}^{d} (c_j + I_j - \delta \cdot k_j) = \sum_{j=1}^{d} (f(k_j, l_j, \theta_j) - \Gamma_j)$$

$$\theta^{s-1} = g(\theta, \xi)$$

Discrete States



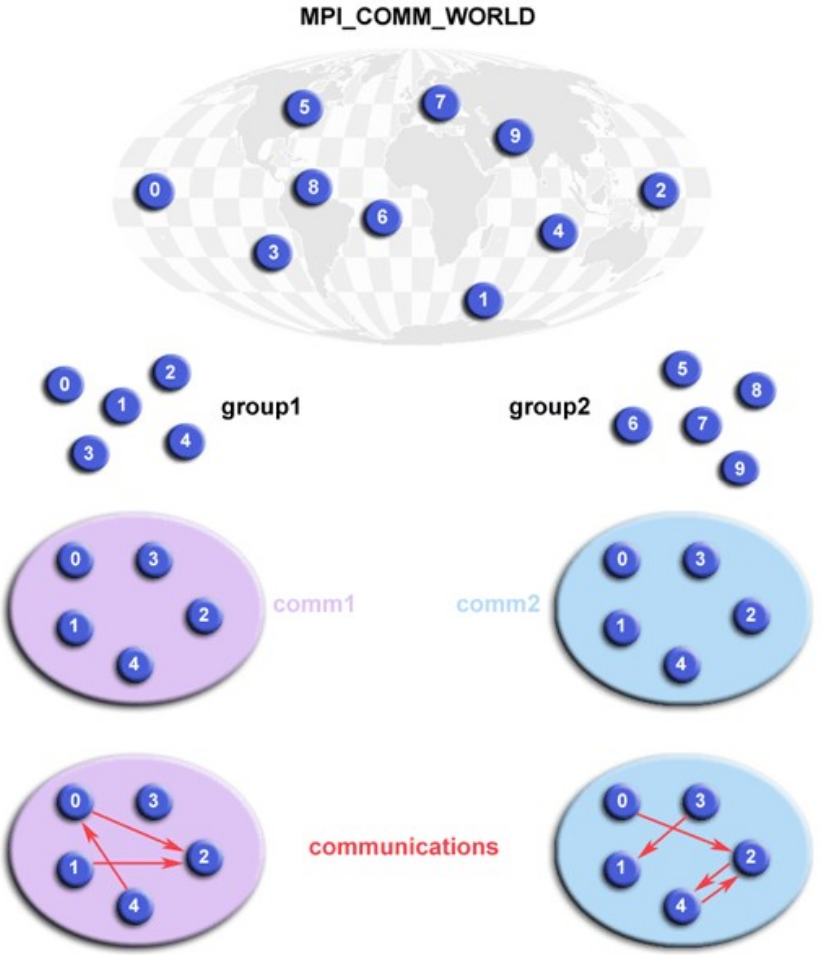


Fig. from https://computing.llnl.gov/tutorials/mpi/

Group and Communicator Management Routines

→ A group is an ordered set of processes, each with a unique integer rank. In MPI, a group is represented within system memory as an object. It is accessible to the programmer only by a ”handle”. A group is always associated with a communicator object.

→ A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. Like groups, communicators are accessible to the programmer only by ”handles”. The handle for the communicator that comprises all processes is MPI_COMM_WORLD.

From the programmer’s perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

Primary Purposes of Group and Communicator Objects

Goals:

→ Allow you to organize tasks, based upon function, into task groups.

→ Enable Collective Communications operations across a subset of related tasks.

→ Provide basis for implementing user defined virtual topology.

Remarks:

→ Groups/communicators can be created and destroyed during program execution.

→ Processes may be in more than one group/communicator having a unique rank within each group/communicator.

Example

Create two different process groups for separate collective communications exchange. → This requires creating new communicators.

> cd OSM_Lab/HPC_day3/code_day3/MPI

2. Have a look at the code

>vi 4a.MPI_group.cpp

3. compile by typing:

> make

4. run the code

>mpiexec -np 8 ./4a.MPI_group.exec

Example (II)

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

#define NPROCS 8

using namespace std;

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

int        rank, new_rank, sendbuf, recvbuf, numtasks,
           ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
MPI_Group  orig_group, new_group;   // required variables
MPI_Comm   new_comm;   // required variable

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks != NPROCS) {
  cout << "Must specify 8 MPI processes = " << NPROCS << " !! Terminating "<< endl;
  MPI_Finalize();
  return 0;
  }

sendbuf = rank;

// extract the original group handle
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

//  divide tasks into two distinct groups based upon rank
if (rank < NPROCS/2) {
  MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
  }
else {
  MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
  }

// create new new communicator and then perform collective communications
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

// get rank in new group
MPI_Group_rank (new_group, &new_rank);
cout << "rank= " << rank << " newrank= "<< new_rank << " recvbuf= " <<recvbuf << endl;

MPI_Finalize();
}
```

>mpiexec -np 8 ./4a.MPI_group.exec

```
rank=  0  newrank=  0  recvbuf=  6
rank=  1  newrank=  1  recvbuf=  6
rank=  3  newrank=  3  recvbuf=  6
rank=  4  newrank=  0  recvbuf= 22
rank=  6  newrank=  2  recvbuf= 22
rank=  5  newrank=  1  recvbuf= 22
rank=  7  newrank=  3  recvbuf= 22
rank=  2  newrank=  2  recvbuf=  6
```

Reduction within group

0 + 1 + 2 + 3 = 6
4 + 5 + 6 + 7 = 22

Virtual topologies

- In terms of MPI, a **virtual topology** describes a **mapping/ordering of MPI processes into a geometric "shape".**

- The two main types of **topologies** supported by MPI are **Cartesian (grid) and Graph**.

- MPI topologies are **virtual** - there may be **no relation between the physical structure of the parallel machine and the process topology**.

- Virtual topologies are built upon MPI communicators and groups.

- Must be "programmed" by the application developer.

Why Use Them?

→ Convenience:

Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure.

For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbour communications for grid based data.

→ Communication Efficiency

Some hardware architectures may impose penalties for communications between successively distant "nodes". A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.

Domain Decomposition

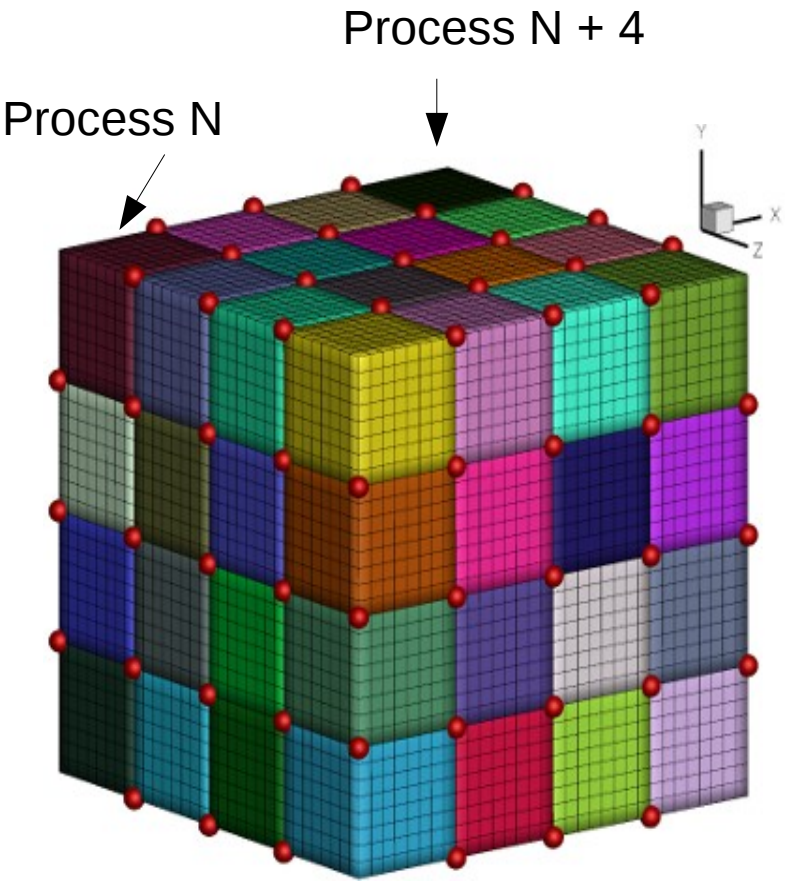
Cartesian distribution: data are distributed linearly between processors.

Useful e.g. when solving PDEs (“ghost zones”).

This is in general a more effective way of distribute the domain, since:

→ It is much more scalable.
→ Communicated data volume can be smaller (especially when a large number of processors is used).
→ It can better map the geometry of the problem and of the Algorithm.

However, it is more difficult to handle:
who are my neighbours?



Cartesian topology

```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder,
    comm_cart)
```

comm_old	input communicator
ndims	number of dimensions of Cartesian grid
dims	specifies the number of processes in each dimension
periods	specifies whether the grid is periodic (true) or not (false) in each dimension
reorder	ranking may be reordered (true) or not (false)
comm_cart	communicator with new Cartesian topology

row-major ordering

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Example (run in break – is involved)

A simplified mapping of processes into a Cartesian virtual topology appears below.

Example: We want to create a 4 x 4 Cartesian topology from 16 processors and have each process exchange its rank with four neighbours.

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

> cd OSM_Lab/HPC_day3/code_day3/MPI

2. Have a look at the code

>vi 5a.virtual_topo.cpp

3. compile by typing:

> make

4. run the code

>mpiexec -np 16 ./5a.virtual_topo.exec

Example

```
main(int argc, char *argv[])  {
int numtasks, rank, source, dest, outbuf, i, tag=1,
   inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,},
   nbrs[4], dims[2]={4,4},
   periods[2]={0,0}, reorder=0, coords[2];

MPI_Request reqs[8];
MPI_Status stats[8];
MPI_Comm cartcomm;   // required variable

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
   // create cartesian virtual topology, get rank, coordinates, neighbor ranks
   MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
   MPI_Comm_rank(cartcomm, &rank);
   MPI_Cart_coords(cartcomm, rank, 2, coords);
   MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
   MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);

   printf("rank= %d coords= %d %d  neighbors(u,d,l,r)= %d %d %d %d\n",
          rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],
          nbrs[RIGHT]);

   outbuf = rank;

   // exchange data (rank) with 4 neighbors
   for (i=0; i<4; i++) {
      dest = nbrs[i];
      source = nbrs[i];
      MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
                MPI_COMM_WORLD, &reqs[i]);
      MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
                MPI_COMM_WORLD, &reqs[i+4]);
      }

   MPI_Waitall(8, reqs, stats);

   printf("rank= %d                  inbuf(u,d,l,r)= %d %d %d %d\n",
          rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]);  }
else
   printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

>mpiexec -np 16 ./5.virtual_topo.exec

```
rank=   2 coords=  0 2 neighbors(u,d,l,r)=  -2  6  1  3
rank=   3 coords=  0 3 neighbors(u,d,l,r)=  -2  7  2 -2
rank=  10 coords=  2 2 neighbors(u,d,l,r)=   6 14  9 11
rank=   6 coords=  1 2 neighbors(u,d,l,r)=   2 10  5  7
rank=   7 coords=  1 3 neighbors(u,d,l,r)=   3 11  6 -2
rank=  15 coords=  3 3 neighbors(u,d,l,r)=  11 -2 14 -2
rank=  14 coords=  3 2 neighbors(u,d,l,r)=  10 -2 13 15
rank=  11 coords=  2 3 neighbors(u,d,l,r)=   7 15 10 -2
rank=  13 coords=  3 1 neighbors(u,d,l,r)=   9 -2 12 14
rank=   5 coords=  1 1 neighbors(u,d,l,r)=   1  9  4  6
rank=   0 coords=  0 0 neighbors(u,d,l,r)=  -2  4 -2  1
rank=   4 coords=  1 0 neighbors(u,d,l,r)=   0  8 -2  5
rank=   8 coords=  2 0 neighbors(u,d,l,r)=   4 12 -2  9
rank=  12 coords=  3 0 neighbors(u,d,l,r)=   8 -2 -2 13
rank=   3                  inbuf(u,d,l,r)=  -2  7  2 -2
rank=   9 coords=  2 1 neighbors(u,d,l,r)=   5 13  8 10
rank=  11                  inbuf(u,d,l,r)=   7 15 10 -2
rank=   7                  inbuf(u,d,l,r)=   3 11  6 -2
rank=  15                  inbuf(u,d,l,r)=  11 -2 14 -2
rank=   6                  inbuf(u,d,l,r)=   2 10  5  7
rank=  12                  inbuf(u,d,l,r)=   8 -2 -2 13
rank=   8                  inbuf(u,d,l,r)=   4 12 -2  9
rank=  13                  inbuf(u,d,l,r)=   9 -2 12 14
rank=   1 coords=  0 1 neighbors(u,d,l,r)=  -2  5  0  2
rank=   9                  inbuf(u,d,l,r)=   5 13  8 10
rank=  14                  inbuf(u,d,l,r)=  10 -2 13 15
rank=   5                  inbuf(u,d,l,r)=   1  9  4  6
rank=   2                  inbuf(u,d,l,r)=  -2  6  1  3
rank=   0                  inbuf(u,d,l,r)=  -2  4 -2  1
rank=   4                  inbuf(u,d,l,r)=   0  8 -2  5
rank=  10                  inbuf(u,d,l,r)=   6 14  9 11
rank=   1                  inbuf(u,d,l,r)=  -2  5  0  2
```

2. User specific data type

MPI predefines its primitive data types:

C Data Types		Fortran Data Types
MPI_CHAR	MPI_C_COMPLEX	MPI_CHARACTER
MPI_WCHAR	MPI_C_FLOAT_COMPLEX	MPI_INTEGER
MPI_SHORT	MPI_C_DOUBLE_COMPLEX	MPI_INTEGER1
MPI_INT	MPI_C_LONG_DOUBLE_COMPLEX	MPI_INTEGER2
MPI_LONG	MPI_C_BOOL	MPI_INTEGER4
MPI_LONG_LONG_INT	MPI_LOGICAL	MPI_REAL
MPI_LONG_LONG	MPI_C_LONG_DOUBLE_COMPLEX	MPI_REAL2
MPI_SIGNED_CHAR	MPI_INT8_T	MPI_REAL4
MPI_UNSIGNED_CHAR	MPI_INT16_T	MPI_REAL8
MPI_UNSIGNED_SHORT	MPI_INT32_T	MPI_DOUBLE_PRECISION
MPI_UNSIGNED_LONG	MPI_INT64_T	MPI_COMPLEX
MPI_UNSIGNED	MPI_UINT8_T	MPI_DOUBLE_COMPLEX
MPI_FLOAT	MPI_UINT16_T	MPI_LOGICAL
MPI_DOUBLE	MPI_UINT32_T	MPI_BYTE
MPI_LONG_DOUBLE	MPI_UINT64_T	MPI_PACKED
	MPI_BYTE	
	MPI_PACKED	

Derived Data Types

MPI also provides facilities for you to **define your own data structures** based upon sequences of the MPI primitive data types.

Such user defined structures are called **derived data** types.

Primitive data types are contiguous.

Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.
MPI provides several methods for constructing derived data types:
- **Contiguous** *(we will below only consider this one as an example)*
- **Vector**
- **Indexed**
- **Struct**

- **MPI derived data types (differently from C or Fortran) are created (and destroyed) at run-time** through calls to MPI library routines.
  Implementation steps:

1. Construct the data type
2. Allocate the data type
3. Use the data type
4. Deallocate the data type

Allocate and destroy the data type

A constructed data type must be committed to the system before it can be used in a communication.

```
MPI_TYPE_COMMIT(DATATYPE, IERR)
MPI_TYPE_FREE(DATATYPE, IERR)
```

e.g. Contiguous Datatype
MPI_TYPE_CONTIGOUS constructs a type-map consisting of the replication of a data type into contiguous locations.

```
MPI_TYPE_CONTIGUOUS(count, oldtype, newtype, ierr)
```

count	number of BLOCKs to be added
oldtype	oldtype Datatype of each element
newtype	new derived datatype

Example

Create a data type representing a row of an array and distribute a different row to all processes.

> cd OSM_Lab/HPC_day3/code_day3/MPI

2. Have a look at the code

```
/1.0, 2.0, 3.0, 4.0, &
 5.0, 6.0, 7.0, 8.0, &
 9.0, 10.0, 11.0, 12.0, &
 13.0, 14.0, 15.0, 16.0 /
```

>vi 6a.derived_data.cpp

3. compile by typing:

> make

4. run the code

Rank 0 `/1.0, 2.0, 3.0, 4.0,`

Rank 1 `5.0, 6.0, 7.0, 8.0,`

Rank 2 `9.0, 10.0, 11.0, 12.0,`

Rank 3 `13.0, 14.0, 15.0, 16.0`

>mpiexec -np 4 ./6a.derived_data.exec

Example data type

```
main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] =
  {1.0, 2.0, 3.0, 4.0,
   5.0, 6.0, 7.0, 8.0,
   9.0, 10.0, 11.0, 12.0,
   13.0, 14.0, 15.0, 16.0};
float b[SIZE];

MPI_Status stat;
MPI_Datatype rowtype;   // required variable

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

// create contiguous derived data type
MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

if (numtasks == SIZE) {
   // task 0 sends one element of rowtype to all tasks
   if (rank == 0) {
      for (i=0; i<numtasks; i++)
        MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
      }

   // all tasks receive rowtype data from task 0
   MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
          rank,b[0],b[1],b[2],b[3]);
   }
else
   printf("Must specify %d processors. Terminating.\n",SIZE);

// free datatype when done using it
MPI_Type_free(&rowtype);
MPI_Finalize();
}
```

>mpiexec -np 4 ./6.derived_data.exec

send one of those elements, not 4

```
rank=  0  b=  1.00000000  2.00000000  3.00000000  4.00000000
rank=  1  b=  5.00000000  6.00000000  7.00000000  8.00000000
rank=  2  b=  9.00000000  10.0000000  11.0000000  12.0000000
rank=  3  b=  13.0000000  14.0000000  15.0000000  16.0000000
```

3. Parallel I/O

High Performance Computing (HPC) applications often do I/O for:

– **Reading initial conditions or datasets for processing**.

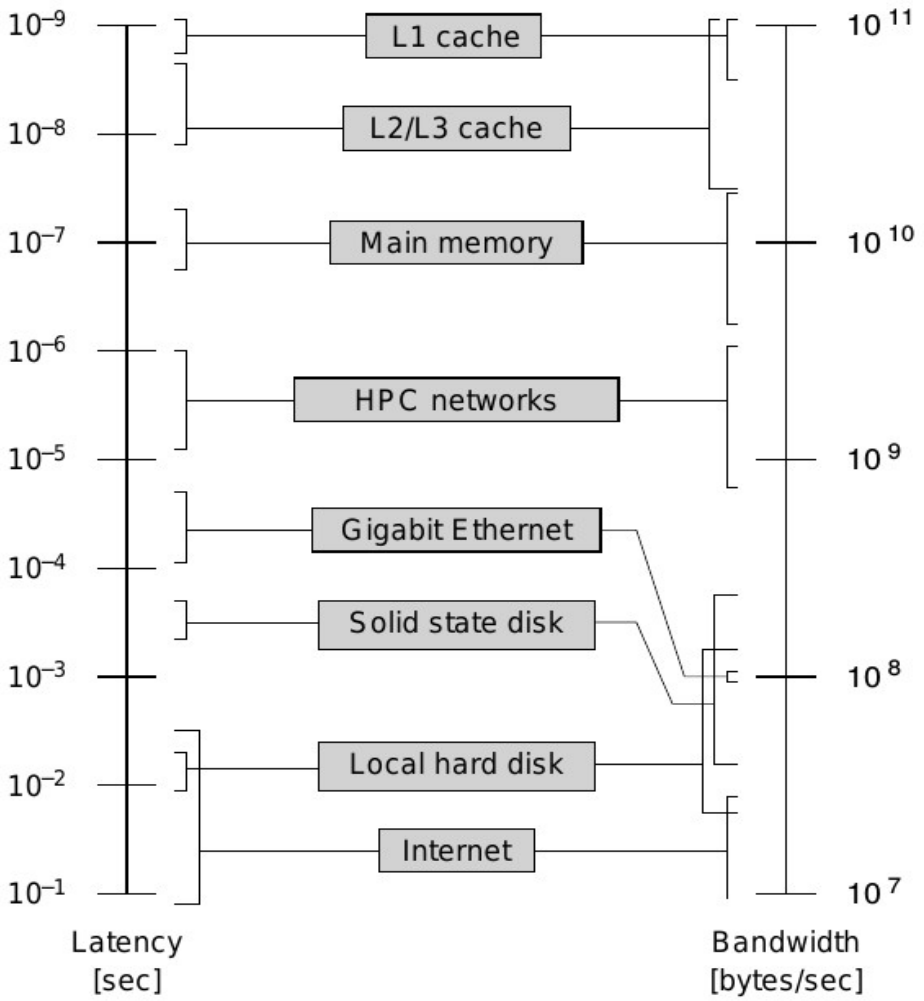
– Writing numerical data from simulations
  → Parallel applications commonly need to write distributed arrays to disk.
  → Saving application-level checkpoints.

- Application state is written to a file for restarting the application in case of a system failure.

Efficient I/O without stressing out the HPC system is challenging:

- **Load and store operations are more time-consuming than multiply operations.**
- **Total Execution Time = Computation Time + Communication Time + I/O time.**
- Optimize all the components of the equation above to get best performance.

Recall: data access speed



Load data

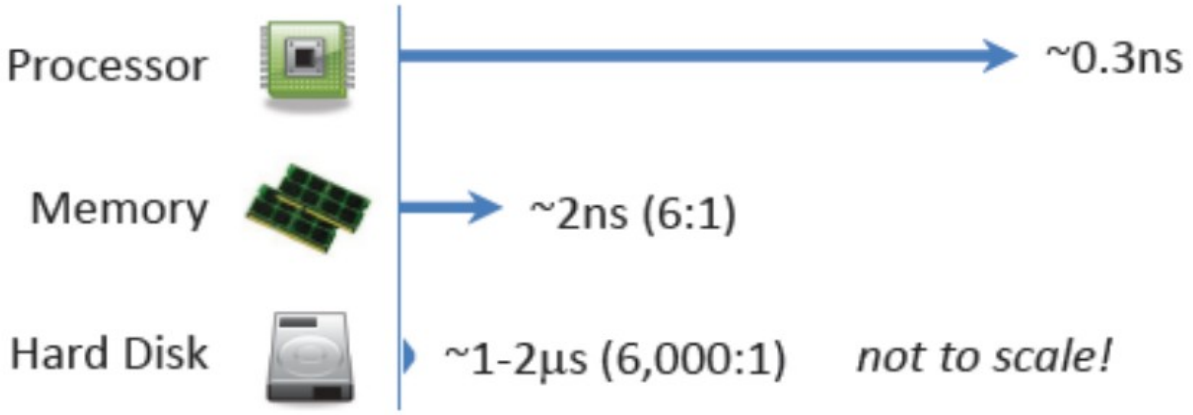
Figure 3.1: Typical latency and bandwidth numbers for data transfer to and from different devices in computer systems. Registers have been omitted because their "bandwidth" usually matches the computational capabilities of the compute core, and their latency is part of the pipelined execution.

Fig. from Hager & Wellein

Relative Speed of Components in HPC Platform

An HPC platform's I/O subsystems are typically slow as compared to its other parts.

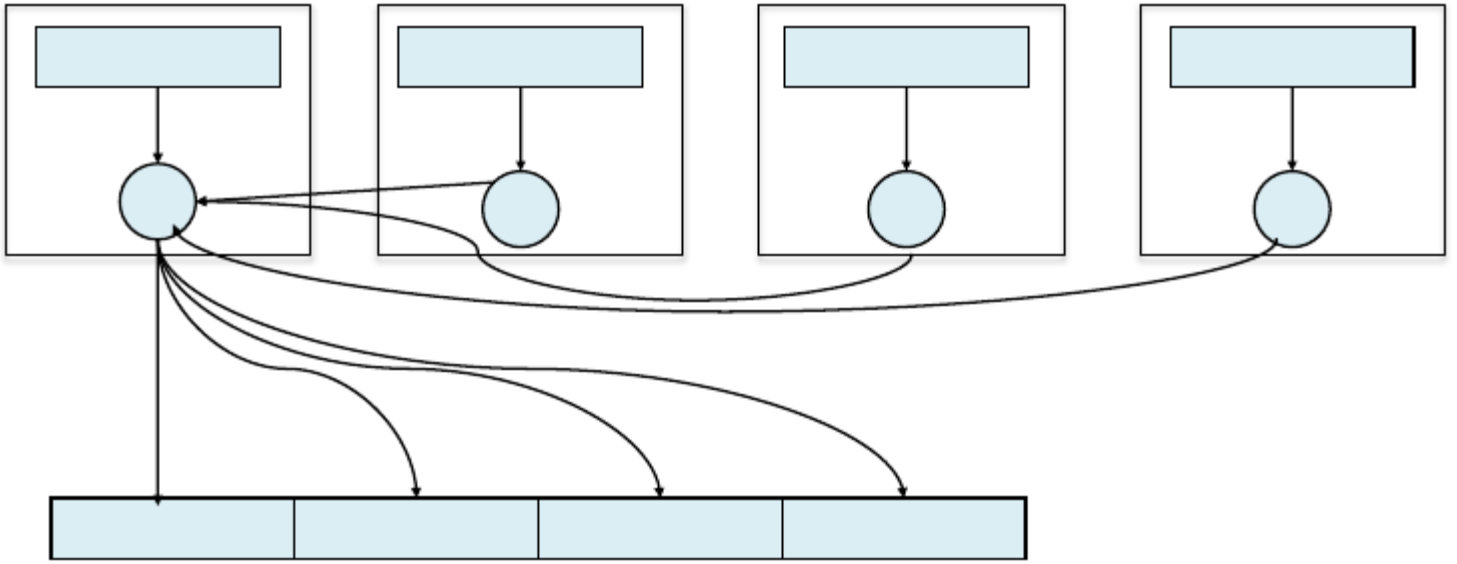
The I/O gap between memory speed and average disk access stands at roughly 10^{-3}



Typical I/O pattern

- All processes send data to master process, and then the process designated as master writes the collected data to the file.

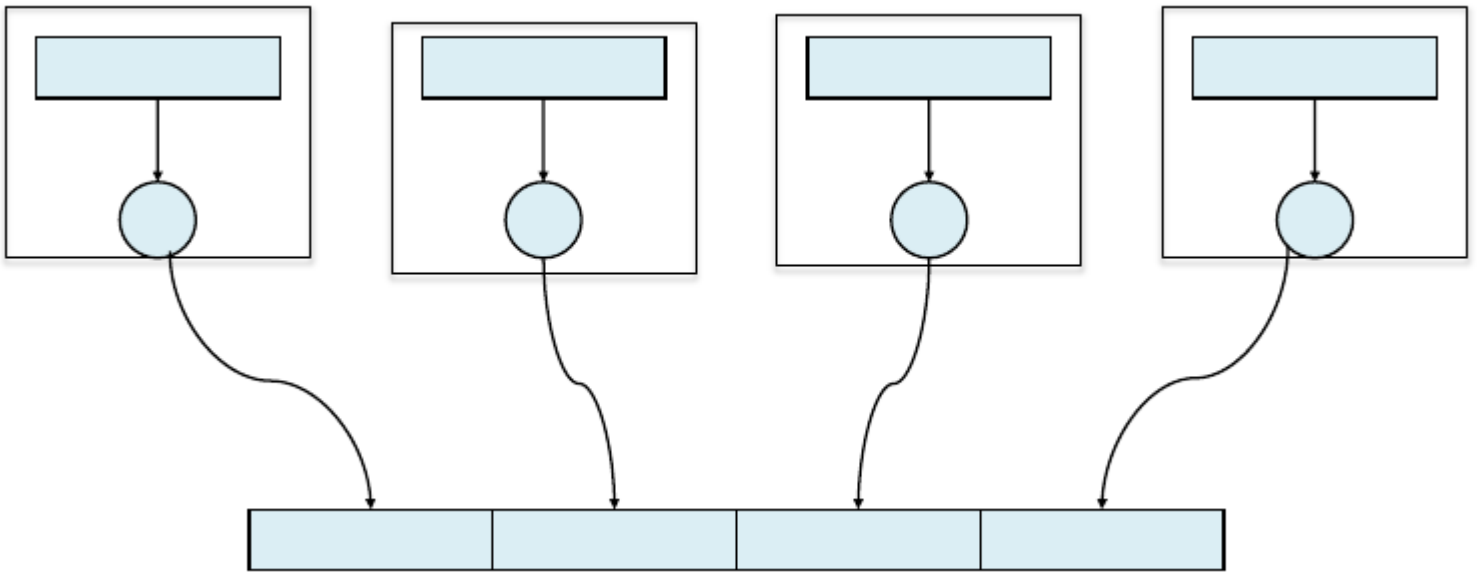
- This sequential nature of I/O can limit performance and scalability of many applications.



Desired pattern

- Multiple processes participating in reading data from or writing data to a common file in parallel.

- This strategy improves performance and provides a single file for storage and transfer purposes.



Parallel I/O: possible strategies

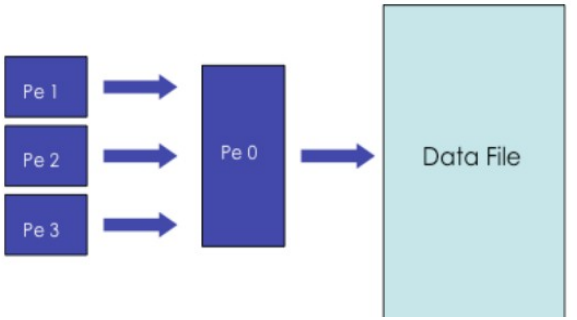
Master-Worker
– Only one processor takes care of the I/O

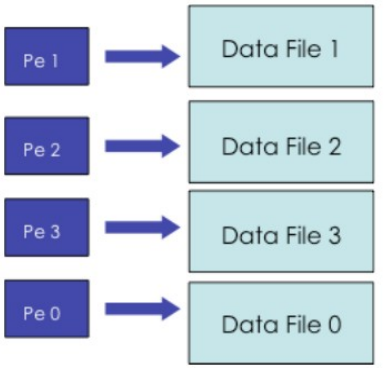
Distributed
– All processors perform the I/O independently

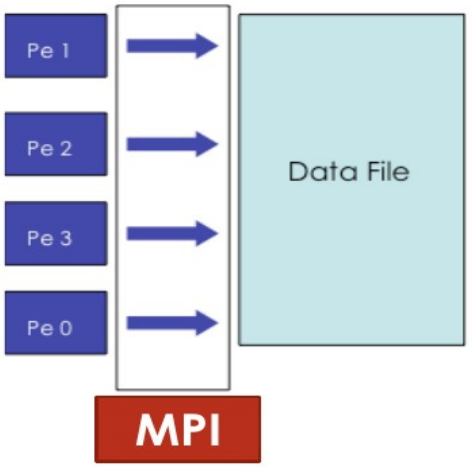
Coordinated
– All processors perform the I/O in an “organized” way

MPI I/O
– The most efficient solution

High level libraries (**HDF5 – https://support.hdfgroup.org/HDF5/**,...)
– The most “elegant” (and sometimes efficient) solution







MPI I/O & Files*

MPI I/O:

- MPI extends the same concepts introduced for parallelizing an algorithm to I/O.

- MPI I/O can be interpreted as message passing to (write) or from (read) a disk.

- A file written with MPI I/O has NOTHING special. It can be read by a sequential (non MPI) code.

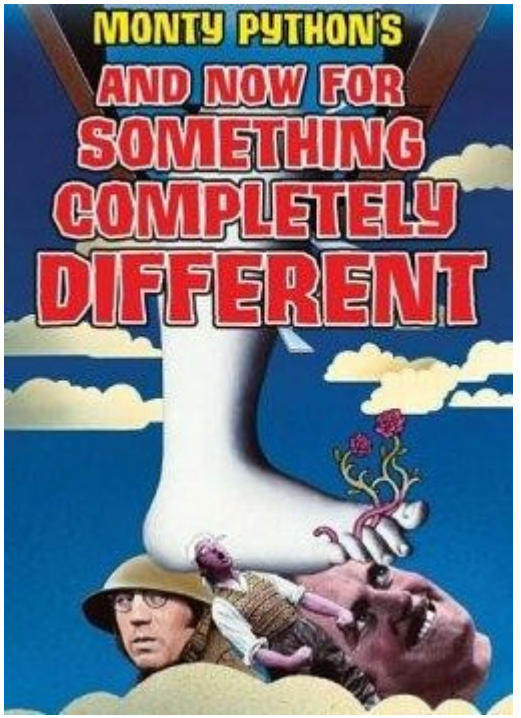
MPI Files:

- An MPI file is an ordered collection of data items.

- MPI supports random or sequential access to any set of these items.

- A file is opened collectively by a group of processes (communicator).

*see HDF5 lecture



python™

MPI

4. MPI in Python

See **https://mpi4py.scipy.org**

→ **MPI for Python** supports convenient, pickle-based communication of generic Python object as well as fast, near C-speed, direct array data communication of buffer-provider objects (e.g., NumPy arrays).

Communication of generic Python objects:
You have to use **all-lowercase methods** (of the Comm class), like send(), recv(), bcast(). Note that isend() is available, but irecv() is not.

Collective calls like scatter(), gather(), allgather(), alltoall() expect/return a sequence of Comm.size elements at the root or all process. They return a single value, a list of Comm.size elements, or None.

Global reduction operations reduce() and allreduce() are naively implemented, the reduction is actually done at the designated root process or all processes.

“Hello World” in Python

Go to **OSM_Lab/HPC_day3/code_day3/MPI4PY**

Run with

> mpiexec -np 4 python hello.py

Make MPI available

```
#hello.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = MPI.COMM_WORLD.Get_size()
print "hello world from process ", rank, " from total ", size , "processes"
```

Point-to-Point Communication

Go to **OSM_Lab/HPC_day3/code_day3/MPI4PY/pointtopoint.py**

```
#passRandomDraw.py
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print "Process", rank, "drew the number", randNum[0]
        comm.Send(randNum, dest=0)

if rank == 0:
        print "Process", rank, "before receiving has the number", randNum[0]
        comm.Recv(randNum, source=1)
        print "Process", rank, "received the number", randNum[0]
```

MPI Broadcast in Python

Go to **OSM_Lab/HPC_day3/code_day3/MPI4PY/bcast.py**

```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

#intialize
rand_num = numpy.zeros(1)

if rank == 0:
        rand_num[0] = numpy.random.uniform(0)

comm.Bcast(rand_num, root = 0)
print "Process", rank, "has the number", rand_num
```

MPI Reductions in Python

- Estimate integrals using the trapezoid rule.
- A range to be integrated is divided into many vertical slivers, and each sliver is approximated with a trapezoid.

$$area \approx \sum_{i=0}^{n} \frac{[f(a) + f(b)]}{2} \cdot \Delta x = \left[\frac{f(a) + f(b)}{2} + \sum_{i=0}^{n} f(a + i\Delta x) + f(a + (i+1)\Delta x) \right] \cdot \Delta x$$

MPI Reductions in Python

```python
import numpy
import sys
from mpi4py import MPI
from mpi4py.MPI import ANY_SOURCE

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

#takes in command-line arguments [a,b,n]
a = float(sys.argv[1])
b = float(sys.argv[2])
n = int(sys.argv[3])

#we arbitrarily define a function to integrate
def f(x):
        return x*x

#this is the serial version of the trapezoidal rule
#parallelization occurs by dividing the range among processes
def integrateRange(a, b, n):
        integral = -(f(a) + f(b))/2.0
        # n+1 endpoints, but n trapazoids
        for x in numpy.linspace(a,b,n+1):
                        integral = integral + f(x)
        integral = integral* (b-a)/n
        return integral


#h is the step size. n is the total number of trapezoids
h = (b-a)/n
#local_n is the number of trapezoids each process will calculate
#note that size must divide n
local_n = n/size

#we calculate the interval that each process handles
#local_a is the starting point and local_b is the endpoint
local_a = a + rank*local_n*h
local_b = local_a + local_n*h

#initializing variables. mpi4py requires that we pass numpy objects.
integral = numpy.zeros(1)
total = numpy.zeros(1)

# perform local computation. Each process integrates its own interval
integral[0] = integrateRange(local_a, local_b, local_n)

# communication
# root node receives results with a collective "reduce"
comm.Reduce(integral, total, op=MPI.SUM, root=0)

# root process prints results
if comm.rank == 0:
        print "With n =", n, "trapezoids, our estimate of the integral from"\
        , a, "to", b, "is", total
```

Go to
OSM_Lab/HPC_day3/code_day3/MPI4PY/reduction.py

Run with
> mpiexec -n 4 python reduction.py a b N
→ integration range [a,b], discretiztion N

> mpiexec -n 4 python reduction.py 0.0 1.0 1000

OUTPUT = ???

f(x) = x^2

Reduction

Questions?

1. Advice – RTFM
https://en.wikipedia.org/wiki/RTFM

2. Advice – http://lmgtfy.com/
http://lmgtfy.com/?q=message+passing+interface