Introduction to Parallel Computing III

ZHANG, Rui

Department of Physics, HKUST

Oct 14th, 2023

Outline

- Recap of The Last Class
- MPI
- GPU
- Hands-on Exercise
- Assignments and mini-Project

Recap of The Last Class

- Recurrence Relation
- Python for Parallel Computing
  - Threading - I/O-bound tasks
    - Use Lock to resolve race condition.
  - Multiprocessing - CPU-bound tasks
    - Sharing data between processes: Value and Array.
  - Queue
  - Pool

Message Passing Interface (MPI) Concepts

- MPI is for communications among processes having separate address spaces.
  - Process – a program running on one core-memory pair.
- Interprocess communications consist of
  - synchronization;
  - moving data from one process's address space to another's.
- Communicator – a collection of processes that can send messages to each other.
- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called MPI_COMM_WORLD.

MPI Concepts

- Some sends/receives may block until another process acts:
  - synchronous send operation blocks until receive is issued;
  - receive operation blocks until message is sent.
- Blocking subroutine returns only when the operation has completed
  - MPI_send.
  - MPI_recv.
- Non-blocking operations return immediately and allow the sub-program to perform other work.

MPI Facts

- MPI is an API, not a language.
- It is for parallel computers, clusters and heterogeneous networks.
- Portability
  - 6 basic functions.
  - >100 functions.
- Initially for C/C++ and Fortran, now for Python as well.

An *application programming interface* (API) is a computing interface which defines interactions between multiple software intermediaries. It defines the kinds of calls or requests that can be made, how to make them, the data formats that should be used, the conventions to follow, etc. It can also provide extension mechanisms so that users can extend existing functionality in various ways and to varying degrees.

A Minimal MPI Program

C:

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

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  printf("Hello, world!\n");
  MPI_Finalize();
  return 0;
}
```

Fortran:

```
program main
use MPI
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```

Python:

```
from mpi4py import MPI

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

print(f”my rank is {rank}.”)
```

Essential Components of MPI

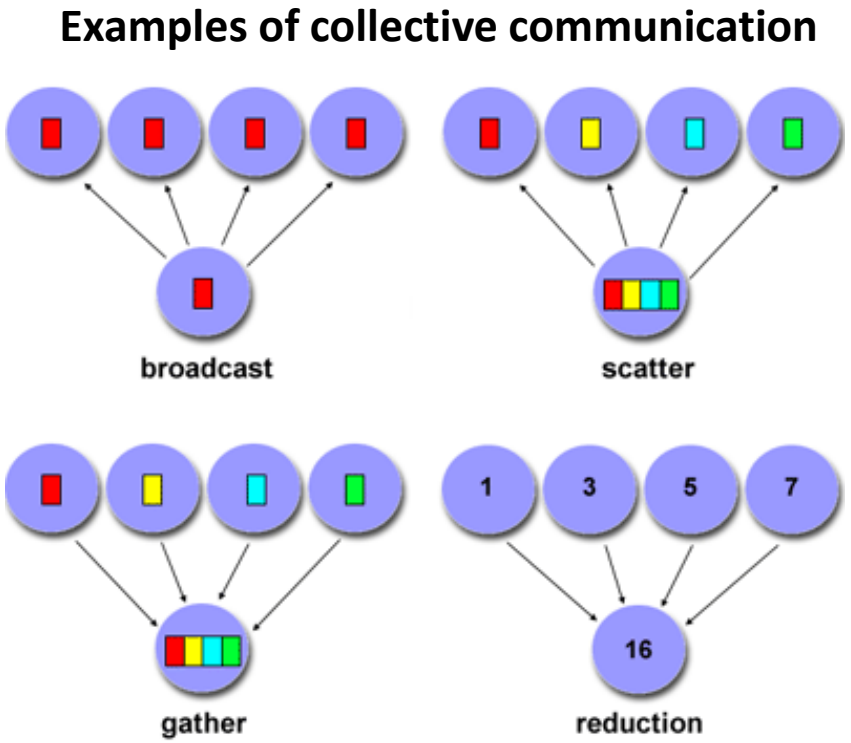
- MPI_INIT initializes the environment
  - C: int MPI_INIT(int *argc, char ***agrv).
- MPI_FINALIZE terminates the environment
  - C: int MPI_Finalize().
- In Python, the above two are executed by the system. No need to explicitly call them.
- Communicator: MPI_COMM_WORLD (for C/C++, Fortran) or MPI.COMM_WORLD (for Python).

Essential Components of MPI

- MPI_COMM_SIZE reports the number of processes
  - C: MPI_Comm_size(MPI_COMM_WORLD, &size);
  - Python: MPI.COMM_WORLD.Get_Size() or comm.Get_Size().
- Q: is this number the same as cpu_count()?
- MPI_COMM_RANK reports the rank (between 0 and size-1)
  - C: MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  - Python: MPI.COMM_WORLD. Get_Rank() or comm.Get_Rank().
- Timing functions: MPI_Wtime or MPI.Wtime, MPI_Wtick or MPI.Wtick.
- More functions can be found: https://www.mpich.org/static/docs/v3.1/www3/

Types of Communication

- Point to Point Communication
  - communication involves two processes.
  - Python: comm.send(data, dest, tag) and comm.recv(source, tag).
- Collective Communication
  - communication that involves a group of processes.
  - barrier:
    - C: int MPI_Barrier(MPI_Comm comm).
    - Python: comm.barrier().
  - broadcast, scatter, gather.
  - reduction, max, min, etc.



For generic Python objects, use MPI_send;
for buffer-like Python objects, use MPI_Send.

Sending a Message

- C: int MPI_Send(void *buf, int count, MPI_Datatype datatype,
  int dest, int tag, MPI_Comm comm)
- Python: comm.send(buf, dest, tag).
- *buf* is the starting point of the message with count elements, each described with datatype.
- *dest* is the rank of the destination process within the communicator comm.
- *tag* is an additional nonnegative integer piggyback information, additionally transferred with the message.
- The *tag* can be used by the program to distinguish different types of messages.

Receiving a Message

For generic Python objects, use MPI_recv; for buffer-like Python objects, use MPI_Recv.

status.SOURCE = source id of the sender
status.TAG = tag of message

- C: int MPI_Recv(void *_buf_, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm,
  MPI_Status *_status_)
- Python: comm.recv(buf, source, tag).
- buf/count/datatype describe the receive buffer.
- Receiving the message sent by process with rank source in comm.
- Envelope information is returned in *status*. If we don't care this info, we can pass MPI_STATUS_IGNORE.
- Only messages with matching tag are received.
- Wildcards:
  - C: MPI_ANY_SOURCE, MPI_ANY_TAG.
  - Python: MPI. ANY_SOURCE, MPI. ANY_TAG.

A Summary of the Basic Functions in MPI

- Q: How many basic functions in MPI?
- A: 6.
- Q: What are them?
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv

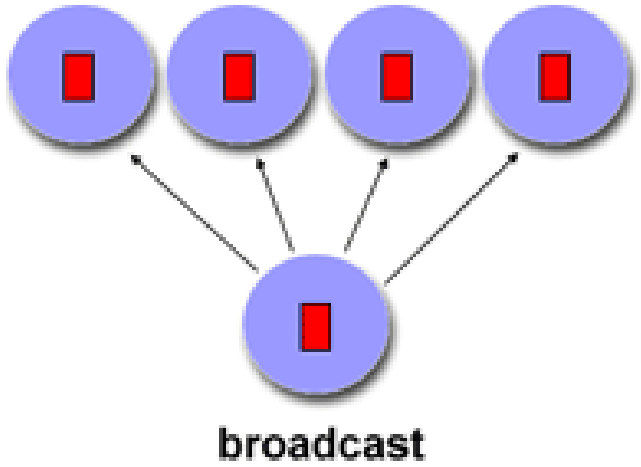
Lowercase vs. Uppercase in mpi4py

- Arbitrary Python objects are converted to byte streams when sending.
- Byte stream is converted back to Python object when receiving.
- Conversions give overhead to communication.
- (Contiguous) NumPy arrays can be communicated with very little overhead with upper case methods:
  - Send(data, dest, tag)
  - Recv(data, source, tag)

https://www.csc.fi/documents/200270/224366/mpi4py.pdf/825c582a-9d6d-4d18-a4ad-6cb6c43fefd8

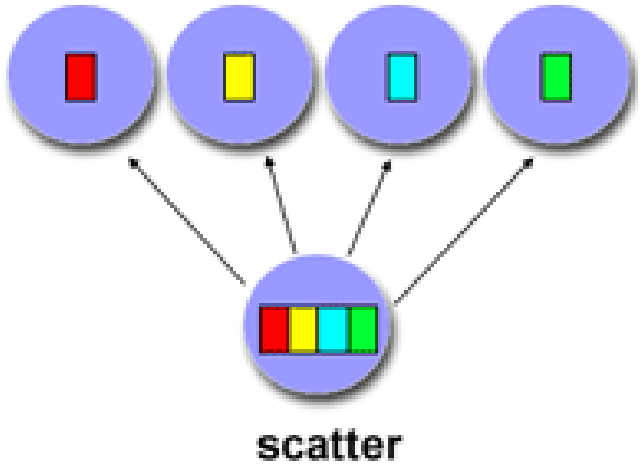
Broadcasting A Message

- C: int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- Python: comm.bcast(buf, root).
- This function broadcasts a message from the process with rank "root" to all other processes of the communicator comm.



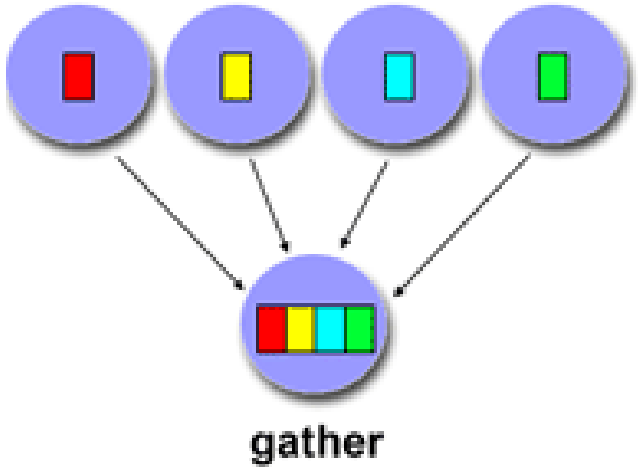
Scattering A Message

- C: int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm).
- Python: comm.scatter(sendbuf, recvbuf, root).
- This function sends data from one process to all other processes in a communicator comm.



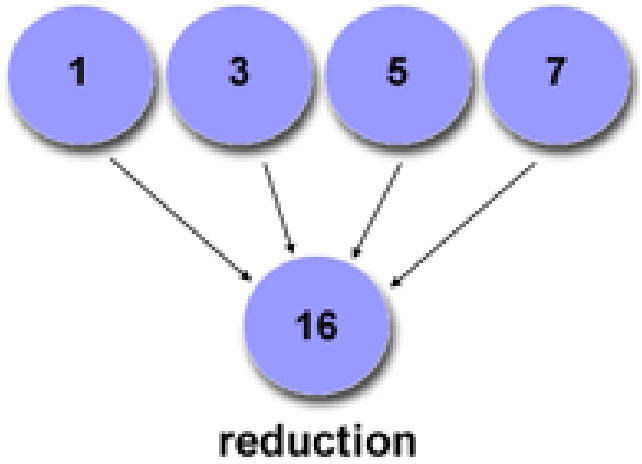
Gathering A Message

- C: int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm).
- Python: comm.gather(sendobj, root).
- This function gathers together values from a group of processes comm.



Reduction of Values in Processes

- C: int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm).
- Python: comm.reduce(sendbuf, recvbuf, op=SUM, root).
- This function reduces values on all processes to a single value.
- MPI_Op examples in C: MPI_MAX, MPI_SUM, MPI_PROD.
- Op examples in mpi4py: MPI.MAX, MPI.SUM, MPI.PROD.



Basic MPI Datatypes in C and Fortran

MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI Datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	
MPI_PACKED	

http://site.sci.hkbu.edu.hk/hpccc/sciblade/course/mpi/MPI-intro2mpi.ppt

Basic Datatypes in Python

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview

Example	Data Type
`x = "Hello World"`	str
`x = 20`	int
`x = 20.5`	float
`x = 1j`	complex
`x = ["apple", "banana", "cherry"]`	list
`x = ("apple", "banana", "cherry")`	tuple
`x = range(6)`	range
`x = {"name" : "John", "age" : 36}`	dict
`x = {"apple", "banana", "cherry"}`	set
`x = frozenset({"apple", "banana", "cherry"})`	frozenset
`x = True`	bool
`x = b"Hello"`	bytes
`x = bytearray(5)`	bytearray
`x = memoryview(bytes(5))`	memoryview

https://www.w3schools.com/python/python_datatypes.asp

MPI for Python: mpi4py

The comm.isend and comm.irecv methods return *Request* instances. The completion of these methods can be managed using the Request.test and Request.wait methods.

```
from mpi4py import MPI

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

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

Q: What is the data type for “data” in the above?
A: dict.

Non-blocking communication:

```
from mpi4py import MPI

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

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11)
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
```

Converting a C Program

```
#include <stdio.h>
static long num_steps = 10000000;
double step;
int main ()
{   int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0;i< num_steps; i++){
            x = (i+0.5)*step;
            sum = sum +
    4.0/(1.0+x*x);
    }
    pi = step * sum;
    printf("Est Pi= %f\n",pi);
}
```

```
#include "mpi.h"
#include <stdio.h>
static long num_steps = 10000000;
double step;
int main ()
{   int i; double x, pi, sum = 0.0;

    MPI_Init(&argc,&argv);

    step = 1.0/(double) num_steps;
    for (i=0;i< num_steps; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    printf("Est Pi= %f\n",pi);

    MPI_Finalize();
}
```

```
#include "mpi.h"
#include <stdio.h>
static long num_steps = 10000000;
double step;
int main ()
{           int i; double x, mypi, pi, sum = 0.0;

            int rank, size;
            MPI_Init(&argc,&argv);

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

            step = 1.0/(double) num_steps;
            for (i=rank;i< num_steps; i+=size){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            mypi = step * sum;
            printf("Est Pi= %f, Processor %d of %d
\n",mypi, rank, size);

            MPI_Finalize();
}
```

Compiling and Running MPI Programs

- Compilation:
  - mpi.h/mpif.h must be included.
  - gcc/gfortran, icc/ifort, pgcc/pgf90
  - No need for python.
- Execution:
  - In general, run “mpirun -n 8 a.out”
  - For python run “mpiexec -n 8 python *.py” or “mpirun -n 8 python *.py”.

When and when *not (necessarily)* to use MPI

- Portability and performance. ✓
- Irregular data structures. ✓
- Building tools for others - libraries. ✓
- Need to manage memory on a per-processor basis. ✓
- Regular computation matches HPF. ✗
- Solution (e.g., library) already exists. ✗
- Require Fault Tolerance. ✗
- Embarrassingly parallel data division. ✗

https://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiintro/MPIIntro.PPT

GPGPU Computing

A graphics processing unit (*GPU*) is a specialized, electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. - *wiki*.

- General-purpose computing on graphics processing units (GPGPU, rarely GPGP) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU) - *wiki*.
- Platforms or Implementations:
  - CUDA: for NVIDIA GPUs, supporting C/C++, fortran, Python, Matlab.
  - OpenCL: open source defined by Khronos Group.
  - Others: SYCL, C++AMP, etc.

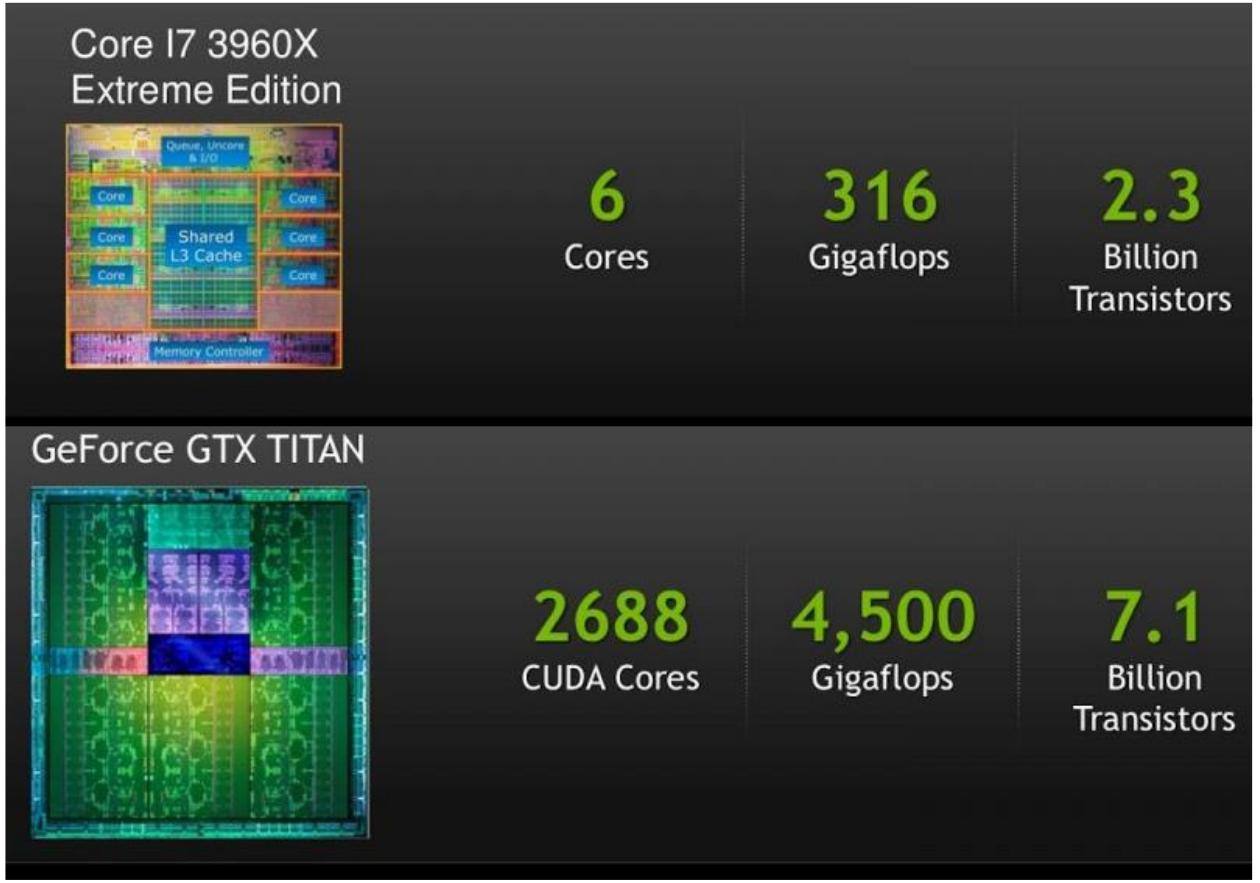
https://news-cdn.softpedia.com/images/news2/The-GPGPU-Technology-2.jpg

GPU Computing Applications

- Linear algebra.
- Signal & image processing.
- Neural networks, deep learning and other machine learning algorithms.
- Monte Carlo simulations.
- Particle or lattice-based simulations.
- In-memory datasets.

CPU versus GPU

- CPU
  - optimized for low-latency access to cashed data sets.
  - control logic for out-of-order and speculative execution.
- GPU
  - optimized for data-parallel, throughput computation.
  - architecture tolerant of memory latency.
  - more transistors dedicated to computation.



https://videocardz.com/39721/nvidia-geforce-gtx-titan-released

CPU	GPU
Task parallelism	Data parallelism
A few heavyweight cores	Many lightweight cores
High memory size	High memory throughput
Many diverse instruction sets	A few highly optimized instruction sets
Explicit thread management	Threads are managed by hardware

https://blog.cherryservers.com/gpu-vs-cpu-what-are-the-key-differences

CUDA

- CUDA stands for Compute Unified Device Architecture.
- It is a parallel computing platform and application programming interface that allows software to use certain types of graphics processing unit for GPGPU.
- Requirements:
  - A CUDA-capable graphic card. You can check https://developer.nvidia.com/cuda-gpus
  - Supported version of Microsoft Windows.
  - Supported version of Microsoft Visual Studio.
  - NVIDIA CUDA toolkit.
- CUDA can also be used in Matlab.

GPGPU Using CUDA+Python

- Numba = NumPy + Mamba: Numba turns Python into a “compiled language”.
- Anaconda
  - numba
  - cudatoolkit

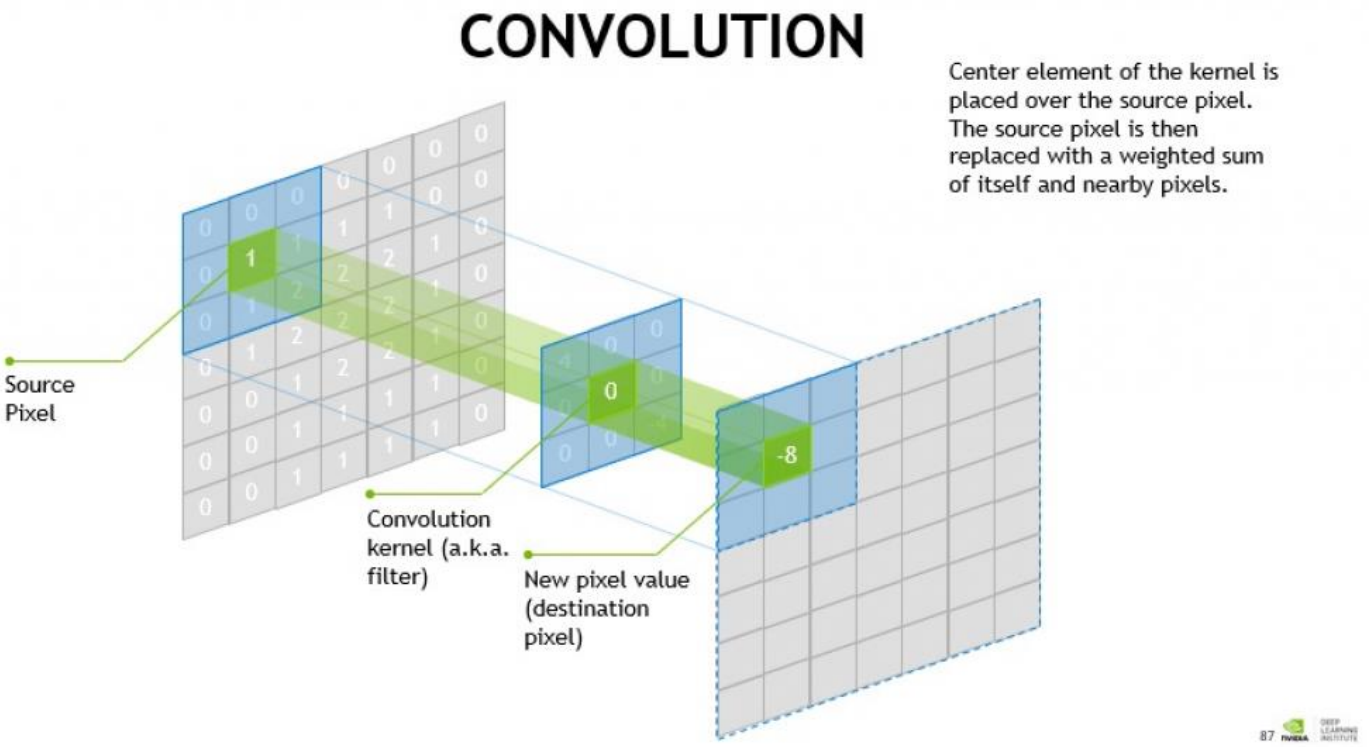
numba is a general-purpose just-in-time (JIT) compiler for Python functions. It provides a way to implement custom GPU algorithms in purely Python syntax when the cudatoolkit package is present.

Anaconda is a free, cross-platform, open-source distribution of the Python and R programming languages for scientific computing, that aims to simplify package management and deployment.

https://www.youtube.com/watch?v=2EbHSCvGFM0

GPU-Enabled Machine Learning

- Neural Networks are *Embarrassingly parallel.*
- Example: Convolutional Neural Networks (CNN).
- PyTorch: a Python-based scientific computing package targeted at two sets of audiences:
  - A replacement for NumPy to use the power of GPUs.
  - a deep learning research platform that provides maximum flexibility and speed.
- PyTorch supports Python 3.X and can integrate with CUDA.



https://www.edge-ai-vision.com/2018/09/whats-the-difference-between-a-cnn-and-an-rnn/

Other Machine Learning Packages

- Specialized hardware accelerator or computer system.
- Example: Tensor Processing Unit (TPU).
  - TPU is an AI accelerator application-specific integrated circuit (ASIC) developed by Google specifically for neural network machine learning, particularly using Google's own TensorFlow software.
- TensorFlow: open-source machine learning platform.
  - It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.

```
#Examplery tensorflow code from tensorflow.org
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

TensorFlow Google Brain

OpenCL

- OpenCL stands for Open Computing Language.
- It is an open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs and other processors.
- Key concept: kernel.
  - Functions executed on an OpenCL device are called ”kernels". A single compute device typically consists of several compute units, which in turn comprise multiple processing elements (PEs). A single kernel execution can run on all or many of the PEs in parallel.

Parallel Computing in Python

- Simple Examples
- Threading
  - Lock
- Multiprocessing
  - Value and Array
- Queue
- Pool

Using Queue

- q1.py
- Q: how to use Queue in Threading module for the above script?
  - Load queue and Thread; see q2.py.
- Caution: don't mix the usage of the two types of Queue.

```
from multiprocessing import Process, Queue

def cal_square(numbers, q):
  for n in numbers:
    q.put(n*n)

#  print('inside process ',str(result))

if __name__ == "__main__":
  numbers = [2,3,5,7]
  q = Queue()
  p = Process(target=cal_square, args=(numbers,q,))

  p.start()
  p.join()

  while q.empty() is False:
    print(q.get())
```

https://www.youtube.com/watch?v=_1ZwkCY9wxk

Using Map and Pool

- map1.py
- map2.py
  - close(): Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.
- p = Pool(processes=3)

```
from multiprocessing import Pool
import time

def f(n):
  sum = 0
  for x in range(1000):
    sum += x*x
  return sum

if __name__ == '__main__':
  t1 = time.time()
  array = [1,2,3,4,5]
  p = Pool(processes=8)
  result = p.map(f,range(10000))
  p.close()
  p.join()

  print(f"Pool took {time.time()-t1}")

  t2 = time.time()
  result=[]
  for x in range(10000):
    result.append(f(x))

  print(f"Serial processing took {time.time()-t1}")
```

https://www.youtube.com/watch?v=_1ZwkCY9wxk

Module “concurrent.futures”*

- The concurrent.futures module provides a high-level interface for asynchronously executing callables.
- The asynchronous execution can be performed with threads (processes), using ThreadPoolExecutor (ProcessPoolExecutor). Both implement the same interface, which is defined by the abstract Executor class.
- Concurrent_futures.py
- process-images.py

https://github.com/CoreyMSchafer/code_snippets/blob/master/Python/MultiProcessing/multiprocessing-demo.py
https://docs.python.org/3/library/concurrent.futures.html

MPI Using Python

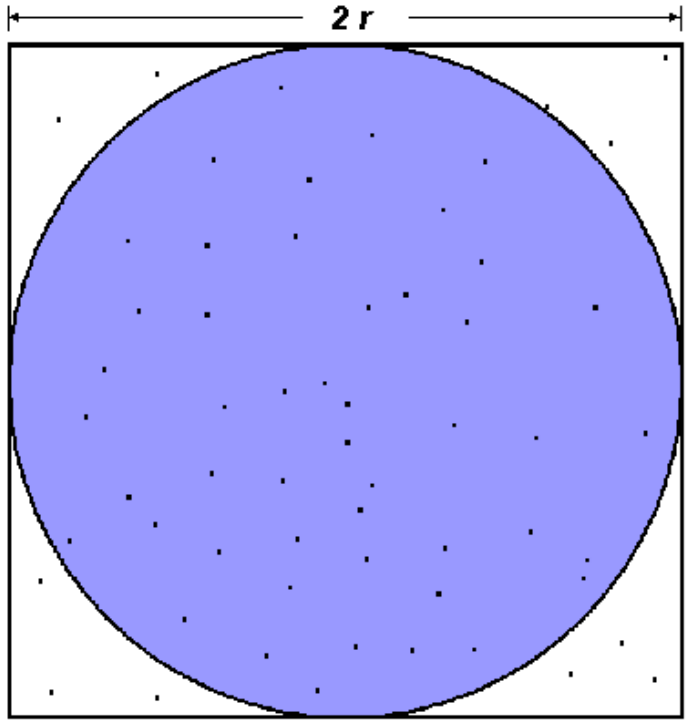
- mpi0.py
- mpi1.py
- mpi_send1.py
- mpi_send2.py
- mpi_send3.py
- mpi_bcast.py
- mpi_gather.py
- mpi_scatter.py
- mpi_reduce.py

GPGPU Demonstrations

- NUMBA
  - VectorAdd0.py
  - VectorAdd.py
- MATLAB
  - gpuDeviceCount
  - Example: A = gpuArray([1 0 1; -1 -2 0; 0 1 -1]); e = eig(A);
- PyTorch
  - torchExample1.py & torchExampe2.py

Exercise 1: Shared Data

- Use shared data to calculate the value of π.
  - *Hint:* use random() in numpy.
- CalPi1.py
- CalPi2.py



$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

Exercise 2: Pool

- Use Pool to find all the prime numbers that are smaller than, say, 100.
  - *Hint:* use sqrt() and %.
- PrimeFinder.py

Exercise 3: MPI Communications

- Write an MPI Ping-Pong program:
  - process 0 sends a message to process 1 (ping);
  - after receiving this message, process 1 sends a message back to process 0 (pong);
- mpi2.py

C:

```
if (my_rank==0) {
MPI_Send( ... dest=1 ...)
MPI_Recv( ... source=1 ...)
} else {
MPI_Recv( ... source=0 ...)
MPI_Send( ... dest=0 ...)
}
```

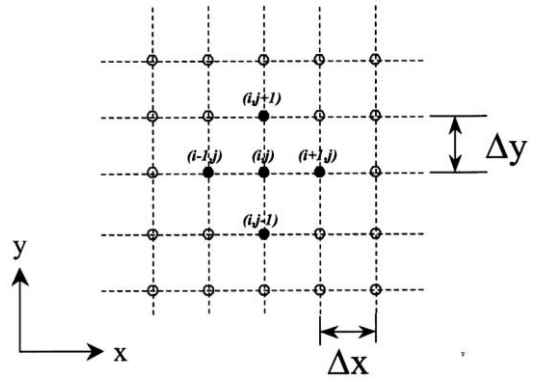
5001 Assignment 2 - Parallel Computing

- Assignment 2.1. **Relay race**: realize the following in python:
  - send an integer via processor 0 -> 1 -> 2 -> … -> np-1;
  - on processer np - 1, square the integer, and send it all the way back: np - 1 -> np - 2 -> … -> 1 -> 0;
  - on processer 0, print out the result and the elapsed time.
- Assignment 2.2. **Euler's number**: write an efficient parallel program to use the following infinite series to calculate Euler's number e:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots.$$

- Assignment 2.3. **Central Limit Theorem**: write a parallel program to demonstrate the Central Limit Theorem that the mean of a large number of independent and identically distributed random numbers follows the Gaussian distribution.

Mini-Project: Solving Differential Equations



- Write a program to solve $T(t)$ satisfying $\frac{dT(t)}{dt} = -T(t)$ with an initial condition $T(0) = 1$. Compare your result to the exact solution.
  - *hint*: use finite difference: $\big(T(t+\Delta t) - T(t)\big)/\Delta t = -T(t)$ and pick a good Δt.
- Write a parallel program to numerically solve the temperature distribution for a 2D system, i.e., a square plate with one side at a fixed temperature *T* = 20°C and the other three sides fixed at *T* = 40°C.
  - *hint*: heat equation: $\frac{\partial T}{\partial t} = \kappa\big(\partial_x^2 T + \partial_y^2 T\big)$ with κ a constant. Similar to the above, pick a Δx to numerically evaluate $\partial_x^2 T = \big(T(x-\Delta x) + T(x+\Delta x) - 2T(x)\big)/\Delta x^2$.
- Change the number of processes involved in the above 2D calculation and evaluate speedup and efficiency.

Materials used or referenced from the following resources

- https://cosy.univ-reims.fr/~fnolot/Download/Cours/HPC/introduction_to_parallel_computing.ppt
- https://www.dias.ie/jetsetschool/presentations/mpi_lecture.ppt
- https://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiintro/MPIIntro.PPT
- https://mpitutorial.com/tutorials/
- https://mpi4py.readthedocs.io/en/stable/
- https://www.youtube.com/watch?v=fKl2JW_qrso&list=WL&index=11
- https://www.youtube.com/channel/UCh9nVJoWXmFb7sLApWGcLPQ
- https://s3.amazonaws.com/ppt-download/intro-gpu-for-ml-print-170422004514.pdf?response-content-disposition=attachment&Signature=xnno5KBivr365JzZXzYoaT29Xh8%3D&Expires=1602727278&AWSAccessKeyId=AKIAIA5TS2BVP74IAVEQ