TERM PROJECT

CS630-005 OPERATING SYSTEM

Topic:

PARALLELIZATION OF CODE

UTILIZE THE POWER OF
MULTIPLE CORES AND UNDERSTAND THE LIMITATION

LAKSHYA SAHARAN
ls565@njit.edu

GROUP: *Individual*

What is Parallelisation?

It is just like parallelization breaks down one huge task into smaller pieces, then working on those simultaneously rather than working on each piece one after another. This works much faster, especially if you have more processors or multiple cores in your computer that can all be used together to do the work.

Example:

1. Think of folding laundry: You can fold it all yourself if you're alone, but that will take some time. You can share the task with three others and divide the work: one person can fold shirts, one can fold pants, someone else can pair up socks, and another person can arrange the already-folded items in neat piles. And if all work simultaneously, you are done far quicker.

Multiple people folding laundry in parallel

2. Imagine you have 1,000 photos that need a filter applied. If your computer processes them one at a time, the task could take quite a while. With parallelization, though, you can break the photos into four batches of 250. Each batch is handled by a separate processor core, allowing all four groups to be processed simultaneously. This way, the entire task finishes much faster—about four times quicker than doing it sequentially.

Parallelizing the Sum of a Large Array

Suppose you want to compute the sum of numbers in a very big array containing 100 million elements. You can instead of processing the whole array sequentially divide it into smaller chunks. Let's say an array is divided into 10 fractions, for example:

- Part 1: Elements from 0 to 10 million
- Part 2: Elements from 10 million to 20 million
- ……
- Part 10: Elements from 90 million to 100 million

Each segment now goes off and processes itself independently, reducing its own elements to a sum. For instance:

- Chunk 1 (0–10 million) reduces to a total sum of **5046**
- Chunk 2 (10–20 million) reduces to a total sum of **8345**
- ….
- Chunk 10 (90–100 million) reduces to a total sum of **9123**

After all the chunks are processed, the results are synchronized and combined to get the total sum:
5046 + 8345 +..... + 9123

You would considerably reduce the execution time needed for such a large computation by dividing the work into smaller parts and computing them concurrently e.g., through use of multiple processor cores.

Computer architecture for instructions execution

There are several architecture patterns implemented for instruction execution:

SISD

It stands for Single Instruction, Single Data.
A single instruction is executed on a single piece of data at a time in serial manner.

SIMD

It stands for Single Instruction, Multiple Data.
A single instruction is executed on multiple data simultaneously. Think of multiple workers using the same tool to do something in parallel.

MIMD

It stands for Multiple Instruction, Multiple Data.
Here multiple processors can work on different tasks concurrently. Think of a library where different people can read different books simultaneously.

MPI (Message Passing Interface)

It is an API which helps multiple cores/processors to pass messages/data and coordinate with one another in order to complete certain tasks. If you have a very heavy computation task then you should divide it so that each core/processor can do it simultaneously with others. In large clusters, each core is stacked up with dedicated memory. So MPI plays a very crucial role in sending the data between these processors.

NJIT has its own cluster which is named as Wulver Supercomputer located at Databank. Wulver has 127 nodes and each node has 128 processors with dedicated memory. Students and researchers use it a lot for scientific research or large computation tasks. MPI runs in background making it possible for all those cores to work as a team and synchronize the result.

MPI Forum is the large community which is responsible for developing and maintaining it. Initially it was composed of educational institutes and researchers, but now it has grown wider to commercial purpose. There are several standard release MPI-1.0, MPI-2.0, MPI-3.0 (latest one released in 2012).

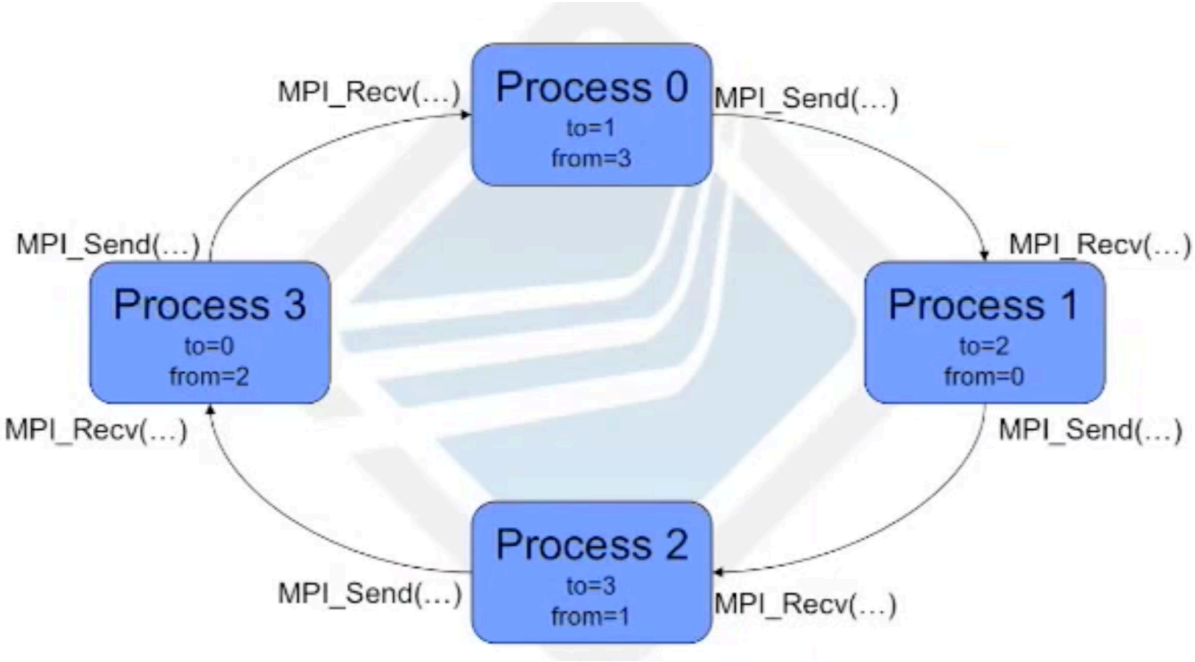
More details are available at their website: https://www.mpi-forum.org/

Open MPI

People often get confused and pronounce it as one word OpenMPI, which is wrong. It is two separate words Open MPI. It is a community which creates APIs and features based on MPI and makes it open source so that great minds can contribute to its expansion and growth. They have very nice documentation of all the versions they have released along with tutorial videos and slides.

I have used Open MPI a lot in my programs to make them parallel. They have a very nice modular design which we can use in any kind of code that we are writing. I have started with a basic program which introduces its features and then went all the way to implement PI estimation using Monte-Carlo method which harnesses the power of MPI and multiple cores.

Trivial MPI Application Flow



This picture is very helpful in understanding the flow of MPI. As the name suggests we are passing some messages/data from one process to another and all these processes are running concurrently on different processors/cores.
Here Process 0 uses MPI_Send() function created by the Open MPI community to send data to Process 1 and the Process 1 receives it using MPI_Recv() function. This goes on in circular fashion with all the processes. So a process gets a message/data, manipulates it or uses it whatever and then passes it on to the next process.

Implementation of MPI

Now that we know about MPI, let's start by implementing a basic program to get our hands dirty and kind of dig into this concept.
I have used a simple C program to demonstrate all the functionality of MPI, so first lets check the code and then understand everything step by step.

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

int main(int argc, char** argv) {
    // first step of making a code parallel is to initialize the MPI module
```

```
    MPI_Init(&argc, &argv);

    // here will define the total number of processes which will be taking
part in our party
    int party_size;
    MPI_Comm_size(MPI_COMM_WORLD, &party_size);

    // assigning each process its unique rank/ID
    int individual_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &individual_rank);

    if (individual_rank == 0) {
        char message[] = "Hello, Process 1!"; // Message to send
        MPI_Send(message, strlen(message) + 1, MPI_CHAR, 1, 0,
MPI_COMM_WORLD);
        printf("Process 0 sent message: \"%s\" to process 1\n", message);
    } else if (individual_rank == 1) {
        char received_message[100]; // Buffer to receive message
        MPI_Recv(received_message, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Process 1 received message: \"%s\" from process 0\n",
received_message);
    }

    MPI_Finalize(); // lets clean up after the party ends
    return 0;
}
```

Lets try to understand all these functions which I have included in this code. First of all this code will just show one process sending data to another process. Now let's see all these MPI based APIs one by one:

MPI_Init

This is the point where MPI comes into existence. You have to put it on the start of your code so that it can initialize your environment.
In the code we are passing two arguments inside of it, argc and argv. These are the command line arguments which will type when running this program using shell or terminal. So will be inputting something like ***-np 4,*** which means the number of processes to run are 4. Then this MPI_Init will initialize our program based on 4 processes.

MPI_Comm_size

Here Comm represents communicator. So this module basically is responsible for dealing with total size or number or processes that are going to participate in our program. We are going to initialize them using this ***party_size*** variable and this is going to help each process to be aware of the total size.

MPI_Comm_rank

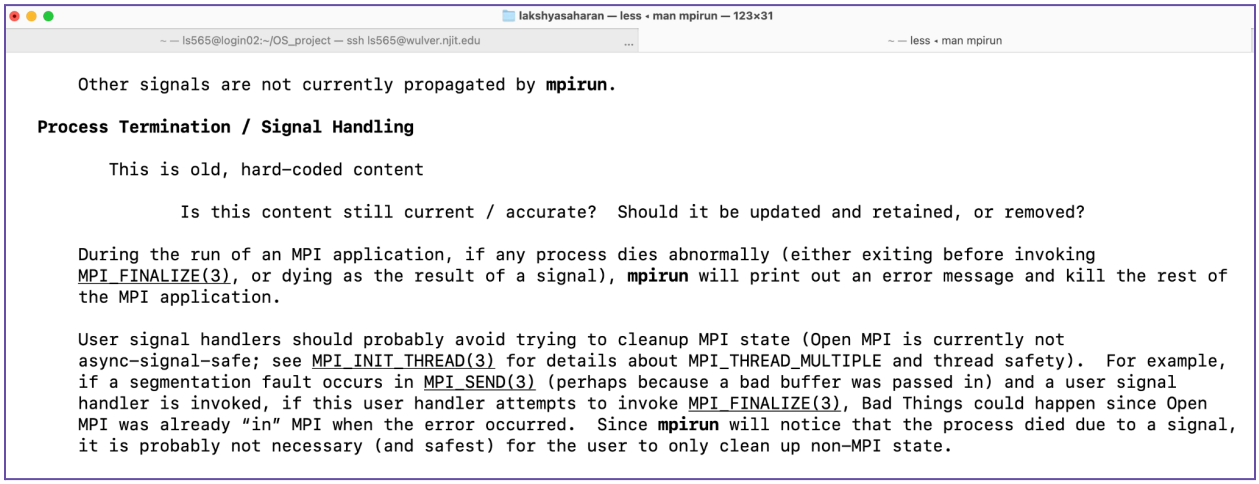
This module defines the rank or ID for each process. It's necessary because I will be able to point out a specific process using this ID. It also helps a process to determine who they are and which process to send the message. Each process rank will be stored in the variable ***individual_rank.***
The rank 0 is assigned to the first process and it acts as the manager. It also does some of the initialisation work. The rest of the processes are ranked from 1 and so one, they are kind of like workers.

MPI_Send & MPI_Recv

As the name suggests, these two modules are responsible for actually sending and receiving data among processes. They take a few arguments like ***party_size***, ***individual_rank***, data to be sent or received.

MPI_Finalize()

This is the final module which tells MPI that we are done for the day and now time to shut down and take a rest. It is very important to include it at the end of the program. Now it returns an exit status which can help us determine if there is any error or not. Maybe a process just died abnormally without invoking the MPI_Finalize() so there will be an error status.
Here is a screenshot from the manual page of ***mpirun*** which tells all the details about the termination.



All of this solidifies the concept which we learn from the chapters “Concurrency”, “Process and Threads” from the book Operating System by William Stalling.

Compile and run the program

Since we are using mpi for the code, we have to use ***mpicc*** and ***mpirun*** for compiling and running our code respectively instead of ***gcc***.

I have created the file ***mpi-intro.c*** with the code.
Now lets compile and run it using **mpicc** and **mpirun**.

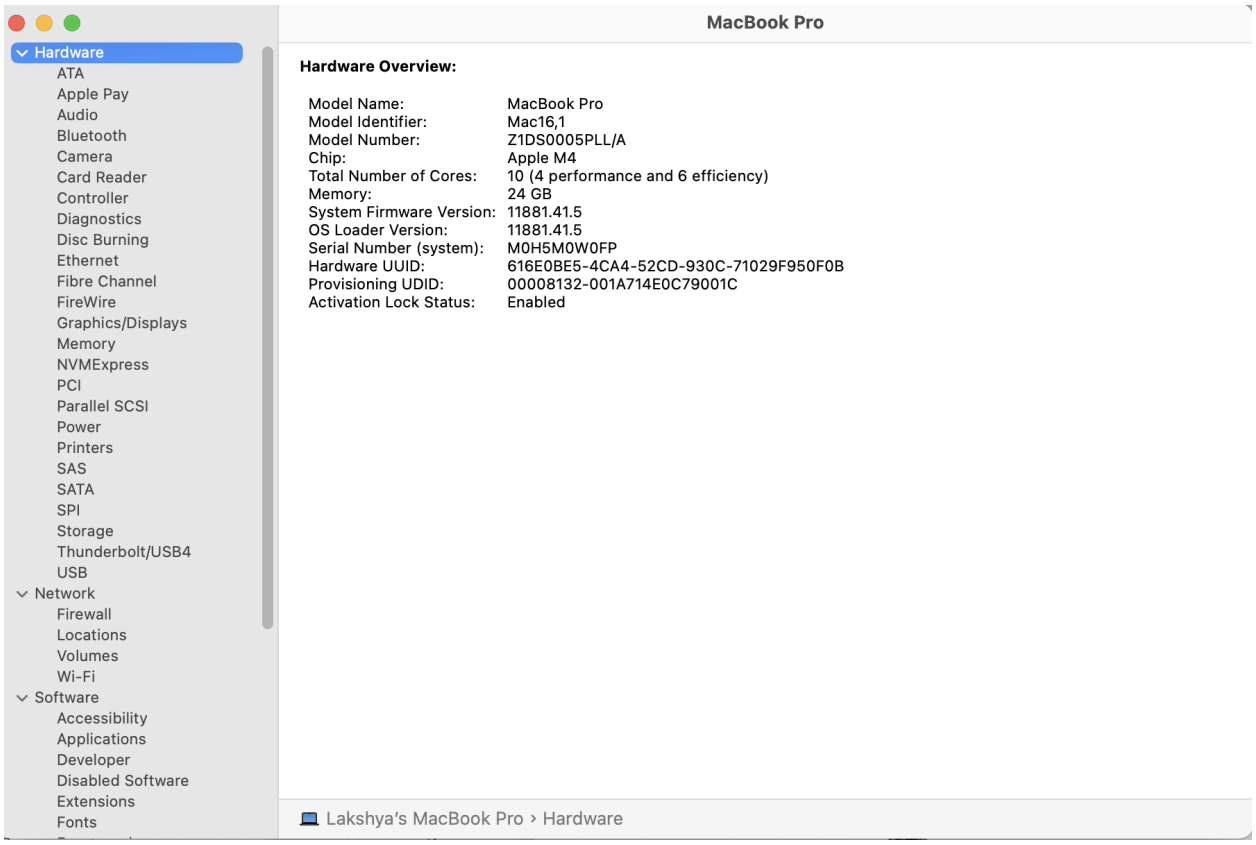
```
lakshyasaharan@mac mpi_basic % ls -l
total 8
-rw-r--r--@ 1 lakshyasaharan  staff  913 Nov 16 21:57 mpi-intro.c
lakshyasaharan@mac mpi_basic % mpicc mpi-intro.c -o mpi-intro.out
lakshyasaharan@mac mpi_basic % ls -lt
total 80
-rwxr-xr-x  1 lakshyasaharan  staff  34048 Nov 17 19:07 mpi-intro.out
-rw-r--r--@ 1 lakshyasaharan  staff    913 Nov 16 21:57 mpi-intro.c

lakshyasaharan@mac mpi_basic % mpirun ./mpi-intro.out

Process 0 sent message: "Hello, Process 1!" to process 1
Process 1 received message: "Hello, Process 1!" from process 0
```

As you can see the output, process 0 is able to successfully send a message to process 1.
Now let's use my Macbook’s cores for the next program.

Using laptop's multiple cores/processors

I am using my Apple's Macbook Pro M4 which has 10 cpu cores. This can be done on any machine by the way. We want to now actually visualize the use of multiple cores and understand how process to core mapping works and also how inter process communication takes place.

Here is a snapshot of specification of my laptop:



As you can see I have 10 cores which are divided into 4 performance and 6 efficiency.
Performance cores → Used for heavy tasks like gaming, video editing or compiling code.
Efficiency cores → Used for lightweight tasks and battery efficiency.

Now let us implement the C code for utilizing all these cores.

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

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

    int party_size;
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &party_size);

    int individual_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &individual_rank);

    char processorName[MPI_MAX_PROCESSOR_NAME];
    int nameLength;
    MPI_Get_processor_name(processorName, &nameLength);

    printf("Process %d among %d is doing work on %s\n", individual_rank,
party_size, processorName);

    // added sleep which will mimic like the process is some computational
task in this time
    usleep(800000); // 0.8 seconds
    printf("Process %d has completed its task!\n", individual_rank);

    MPI_Finalize();
    return 0;
}
```

In this code the section inside the MPI environment will run concurrently on different cores. I have added a delay for each process for better visualization of output. It's a good starting point for understanding MPI.

Now let's check out the output for different numbers of cores.

```
lakshyasaharan@mac multi-core % ls -l
total 8
-rw-r--r--  1 lakshyasaharan  staff  642 Nov 16 22:02 multi-core.c
lakshyasaharan@mac multi-core % mpicc multi-core.c -o multi-core.out

#Let's run it for different number of cores

#-----1 core------
lakshyasaharan@mac multi-core % mpirun -np 1 multi-core.out
Process 0 among 1 is doing work on mac.resource.campus.njit.edu
Process 0 has completed its task!



#-----4 core------
lakshyasaharan@mac multi-core % mpirun -np 4 multi-core.out
Process 3 among 4 is doing work on mac.resource.campus.njit.edu
```

```
Process 0 among 4 is doing work on mac.resource.campus.njit.edu
Process 1 among 4 is doing work on mac.resource.campus.njit.edu
Process 2 among 4 is doing work on mac.resource.campus.njit.edu
Process 1 has completed its task!
Process 3 has completed its task!
Process 0 has completed its task!
Process 2 has completed its task!



#-----10 core------
lakshyasaharan@mac multi-core % mpirun -np 10 multi-core.out
Process 6 among 10 is doing work on mac.resource.campus.njit.edu
Process 8 among 10 is doing work on mac.resource.campus.njit.edu
Process 1 among 10 is doing work on mac.resource.campus.njit.edu
Process 3 among 10 is doing work on mac.resource.campus.njit.edu
Process 0 among 10 is doing work on mac.resource.campus.njit.edu
Process 4 among 10 is doing work on mac.resource.campus.njit.edu
Process 7 among 10 is doing work on mac.resource.campus.njit.edu
Process 2 among 10 is doing work on mac.resource.campus.njit.edu
Process 9 among 10 is doing work on mac.resource.campus.njit.edu
Process 5 among 10 is doing work on mac.resource.campus.njit.edu
Process 6 has completed its task!
Process 9 has completed its task!
Process 8 has completed its task!
Process 1 has completed its task!
Process 3 has completed its task!
Process 4 has completed its task!
Process 0 has completed its task!
Process 7 has completed its task!
Process 2 has completed its task!
Process 5 has completed its task!
```

Well, that ran pretty nicely for different cores.
Notice that the output is not ordered, instead it is in a random way. Well that is because it is **running parallel**.

What will happen if I input 11 cores or 20 cores?

It will actually show an error that the slot is not found, because I have only 10 cores, so anything beyond that will be problematic.

Lets see this into action.

```
lakshyasaharan@mac multi-core % mpirun -np 20 multi-core.out
--------------------------------------------------------------------------
There are not enough slots available in the system to satisfy the 20
slots that were requested by the application:

  multi-core.out

Either request fewer procs for your application, or make more slots
available for use.

A "slot" is the PRRTE term for an allocatable unit where we can
launch a process.  The number of slots available are defined by the
environment in which PRRTE processes are run:

  1. Hostfile, via "slots=N" clauses (N defaults to number of
     processor cores if not provided)
  2. The --host command line parameter, via a ":N" suffix on the
     hostname (N defaults to 1 if not provided)
  3. Resource manager (e.g., SLURM, PBS/Torque, LSF, etc.)
  4. If none of a hostfile, the --host command line parameter, or an
     RM is present, PRRTE defaults to the number of processor cores

In all the above cases, if you want PRRTE to default to the number
of hardware threads instead of the number of processor cores, use the
--use-hwthread-cpus option.

Alternatively, you can use the --map-by :OVERSUBSCRIBE option to ignore the
number of available slots when deciding the number of processes to
launch.
--------------------------------------------------------------------------
lakshyasaharan@mac multi-core %
```

As you can see it is showing me an error that I don't have enough slots which means that MPI cannot place the specified processes on the cores available in my laptop.
There is a workaround solution to this problem using ***--oversubscribe*** in which we can specify as many processes and the MPI will map them to limited cores by allowing multiple processes on a single core. This actually is not beneficial as it will make the code run in serial manner and hence should only be used for testing purposes.

Visualize Amdahl's law

Amdahl's law is a theorem which helps us understand how much speed up we get by making a certain part out code parallel.
Many people might think that if we just keep on increasing cores then the code will run faster and faster, but that is not the case. Overhead and bottleneck will ruin everything after a certain point. Amdahl's law helps us visualize this idea by a relationship between time taken by code running in serial vs time taken by it in parallel with a certain number of cores.

Suppose time taken serial run is ***t-seq*** and time taken in parallel run by ***t-par.*** Then the speed up will the ratio of these terms:

Speed up, **S = t-seq/t-par**

Now it's very naive to think that we can make the whole code run parallel, that's not possible. We can only make certain parts of the code parallel. So keeping this in mind the Amdahl's law can be stated as follow:

```
Speedup = 1 / (1 - parallel_portion + (parallel_portion / number_of_cores))
```

parallel_portion: part of the program that we can make parallel
number_of_cores: number of cores that we use for our code

Monte-Carlo Method

I have used the famous Monte-carlo method for estimating PI for visualizing Amdahl's law on multiple cores.

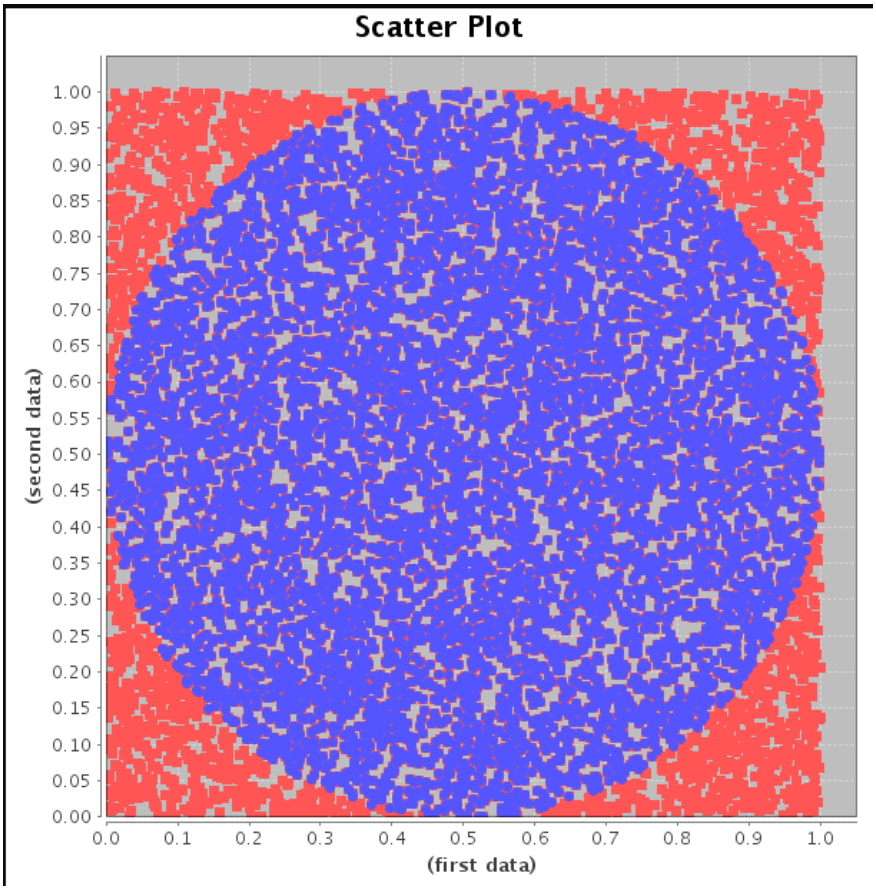
In this concept I will try to calculate PI by calculating the count of points inside the circle.
Let me explain this in depth:
Lets say we have a square of length ***2r*** and a circle of radius ***r*** inside that square.

Area of circle → $\boldsymbol{\Pi x r^2}$
Area of square → $\boldsymbol{2r \times 2r = 4r^2}$

Therefore,

```
Π = (Area of circle/Area of square) x 4
```

Now consider this on the x and y plane.



For any point (x,y) to be in the circle: $x^2 + y^2 <= 1$

Now we can generate this formula:

```
Π = (points inside circle/total number of points) x 4
```

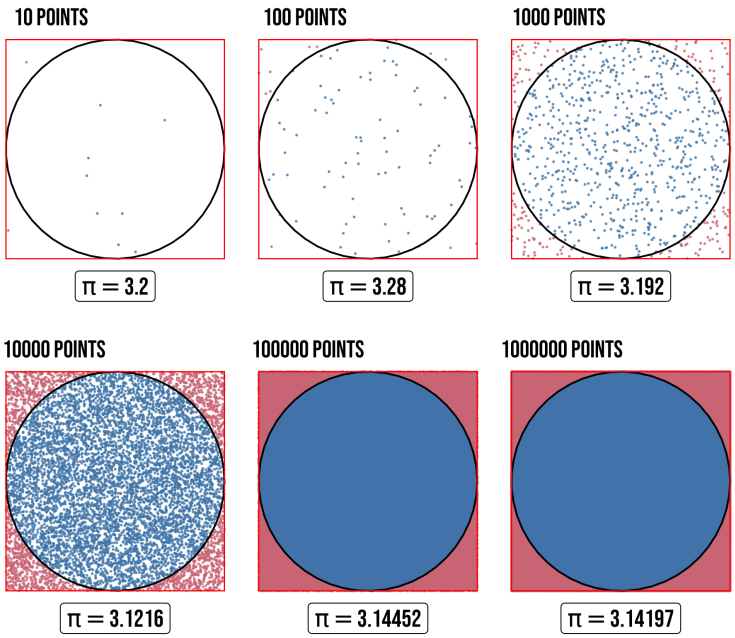
Now let us do it with the code, shall we.

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


// This is the insane number of points that each of our lovely processes
will handle.
#define TOTAL_NUMBER_OF_POINTS 10000000

// this function is ran by each process to generate a random number
double generate_random() {
```

```
    return (double)rand() / (double)RAND_MAX;
}

// each process will run this function to count the points inside the
circle
int count_points_inside_circle(int number_of_points) {
    int cnt = 0;
    for (int i = 0; i < number_of_points; i++) {
        double x = generate_random();
        double y = generate_random();
        if (x * x + y * y <= 1.0) {
            cnt++;
        }
    }
    return cnt;
}

int main(int argc, char** argv) {
    int party_size, individual_rank;
    int local_cnt = 0, total_cnt = 0;
    double pi_estimation;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &party_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &individual_rank);

    // only the first process will run this part
    if (individual_rank == 0) {
        srand(time(NULL));
    }

    // number of points to be handled by each process
    int points_for_each_process = TOTAL_NUMBER_OF_POINTS / party_size;

    // Start the timer
    double start_time = MPI_Wtime();

    // each process will gather its own count
    local_cnt = count_points_inside_circle(points_for_each_process);

    // MPI_Reduce will collect the count of each process and then
accumulate it to generate a total count
    MPI_Reduce(&local_cnt, &total_cnt, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);
```

```
    // first process will do the final work of calculating the pi using
this formula and then prints it
    if (individual_rank == 0) {
        pi_estimation = (double)total_cnt / TOTAL_NUMBER_OF_POINTS * 4.0;

        // End the timer
        double end_time = MPI_Wtime();

        printf("Pi value estimation: %.6f\n", pi_estimation);
        printf("Parallel execution time: %.6f seconds\n", end_time -
start_time);
    }

    // clean up after the party ends
    MPI_Finalize();
    return 0;
}
```

- In this code I am defining 10 million points for our calculation. Higher is the number, better we can estimate the value of PI. Check the picture below:



- The `generate_random()` function will generate a random number between 0 and 1 which is the coordinate of a point (x,y)

- Then the function `count_points_inside_circle(int number_of_points)` will be utilized by each process concurrently to count the points inside the circular trajectory.
- All the MPI functions were already defined in our first program, they are the same here.
- Please note that the rank 0 process is the first process which does all the initialisation and finalizing work for the program.
- Each process will concurrently count the points and update the variable `local_cnt`.
- Then `MPI_Reduce` will gather this variable `local_cnt` of all the processes and accumulate a grand total.
- I am calculating the time using `MPI_Wtime` API

Output on my laptop

Lets check the output of this program on my laptop. Will try to gradually increase the number of cores.

```
lakshyasaharan@mac monte-carlo-pi % ls -l
total 80
-rw-r--r--  1 lakshyasaharan  staff   1986 Nov 17 22:03 pi-parallel.c
-rwxr-xr-x  1 lakshyasaharan  staff  34192 Nov 17 22:04 pi-parallel.out
lakshyasaharan@mac monte-carlo-pi % mpirun -np 1 pi-parallel.out
Pi value estimation: 3.141338
Parallel execution time: 0.132022 seconds
lakshyasaharan@mac monte-carlo-pi % mpirun -np 2 pi-parallel.out
Pi value estimation: 3.141659
Parallel execution time: 0.067870 seconds
lakshyasaharan@mac monte-carlo-pi % mpirun -np 4 pi-parallel.out
Pi value estimation: 3.140542
Parallel execution time: 0.036594 seconds
lakshyasaharan@mac monte-carlo-pi % mpirun -np 8 pi-parallel.out
Pi value estimation: 3.141802
Parallel execution time: 0.023342 seconds
lakshyasaharan@mac monte-carlo-pi % mpirun -np 10 pi-parallel.out
Pi value estimation: 3.142051
Parallel execution time: 0.019723 seconds
```

As you can see the output, as we keep increasing the cores, the time of execution is gradually decreasing.

Does this mean that we can keep increasing the cores and see performance boost?

Ans:
Noooooo!!!!!

There is a limit on this speed increase with the number of cores, after a certain point the performance will become stagnant. And if you still don't stop and keep increasing the cores then you will surprisingly see the performance actually go down and execution time will increase. This happens because of the overhead in managing all the cores.

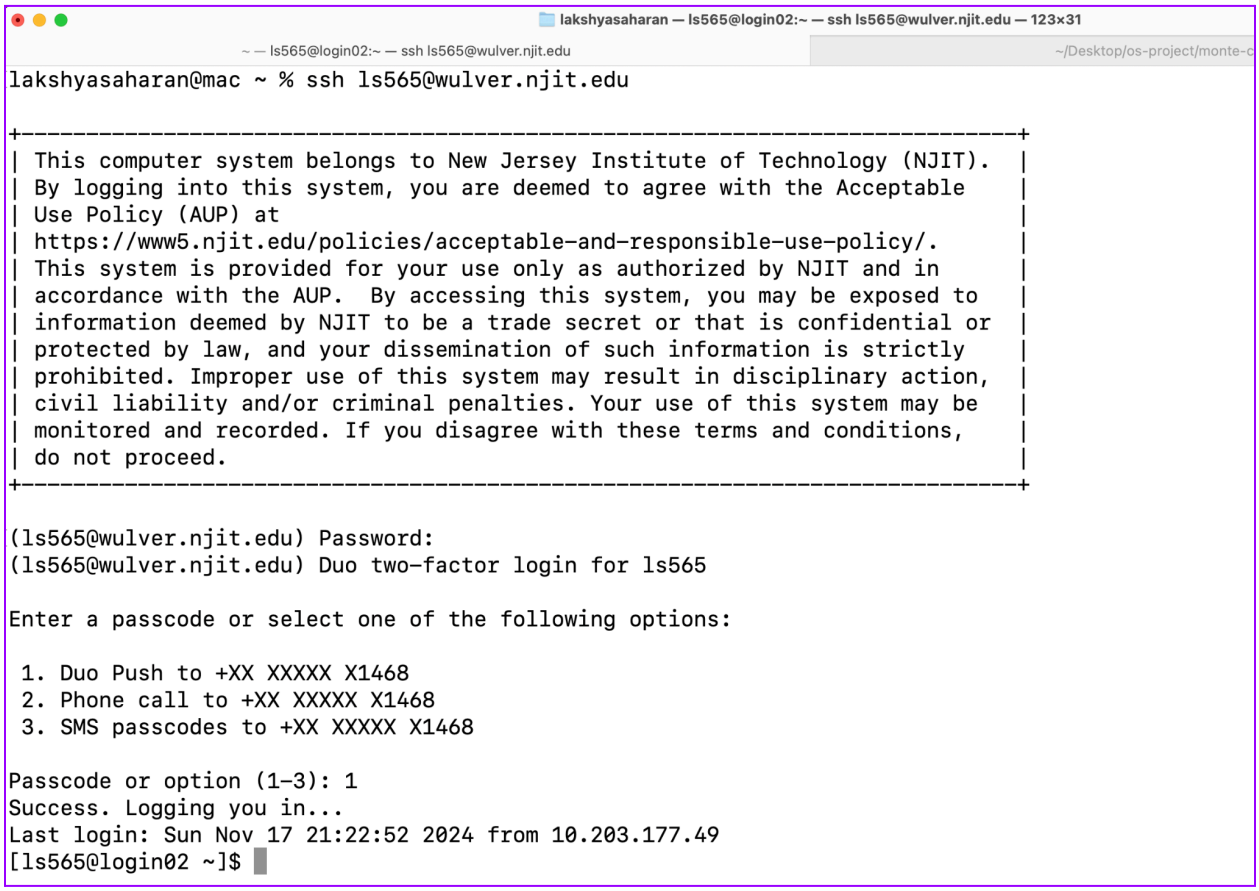
Parallel Slowdown

This is the phenomena where the overhead of managing multiple cores is so much that the execution time starts to increase.
A perfectly said metaphor:
"It is naively admitted that a task can be done faster by a team than by a single worker The problem is that a lot of time can be wasted within a team because of waits chats and misunderstandings"

Lets check this out using our NJIT's HPC.

Using Wulver Supercomputer with 800 cores

Will transfer all the files on our NJIT's High Performance Computing cluster and then run our code for up to 800 cores.



I have logged into the cluster.
Now I can use the supercomputer's compute nodes for running my code.

For that first we need to transfer all the files. So I have created a directory as OS_Project and it has my code.

```
[ls565@login02 OS_project]$ ls -l
total 130
-rwxr-xr-x 1 ls565 ls565 26032 Nov 17 22:25 a
-rw-r--r-- 1 ls565 ls565     0 Nov 17 12:51 mpi1.err
-rw-r--r-- 1 ls565 ls565  2641 Nov 17 12:54 mpi1.out
-rw-r--r-- 1 ls565 ls565  1986 Nov 17 12:47 pi_parallel.c
-rw-r--r-- 1 ls565 ls565  1304 Nov 17 22:30 submit.sh
[ls565@login02 OS_project]$
```

So as you can see the code is inside the file pi_parallel.c
a → the compiled file using mpicc
mpi1.out → the output file after the computation is completed
mpi1.err → error log file in case something goes wrong during computation
submit.sh → it is a bash script used by SLURM for running the job

Will run our code from 1 core to all the way upto 800 cores.
All these configurations are defined under the submit.sh script.
Here is what is inside of it:

```
#!/bin/bash -l
#SBATCH --job-name=mpi_check
#SBATCH --output=mpi.out # %x.%j expands to slurm JobName.JobID
#SBATCH --error=mpi.err
#SBATCH --partition=general
#SBATCH --qos=standard
#SBATCH --account=kjc59 # Replace PI_ucid which the NJIT UCID of PI
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=120
#SBATCH --time=59:00  # D-HH:MM:SS
#SBATCH --mem-per-cpu=4000M

echo "----1 core-----"
mpirun -np 1 ./a
echo "----12 cores-----"
mpirun -np 12 ./a
echo "----16 cores-----"
mpirun -np 16 ./a
echo "----32 cores-----"
mpirun -np 32 ./a
echo "----94 cores-----"
mpirun -np 94 ./a
echo "----120 cores-----"
mpirun -np 120 ./a
echo "----140 cores-----"
mpirun -np 140 ./a
echo "----160 cores-----"
mpirun -np 160 ./a
echo "----190 core-----"
mpirun -np 190 ./a
echo "----200 cores-----"
mpirun -np 200 ./a
echo "----400 cores-----"
mpirun -np 400 ./a
echo "----600 cores-----"
mpirun -np 600 ./a
```

```
echo "----700 cores-----"
mpirun -np 700 ./a
echo "----800 cores-----"
mpirun -np 800 ./a
```

Let me break down what is going on in here:

- ➢ The initial part with #SBATCH is defining the parameters for SLURM. We are defining our output and err log file names.
- ➢ The number of nodes to be used in our case is 8 because each node has 128 cores and to reach the 800 core mark we have to use 8 nodes.
- ➢ Here the inter process communication will take place over the network which will utilize InifiniBand. Meaning the processes will communicate with each other using a network spread across different nodes.

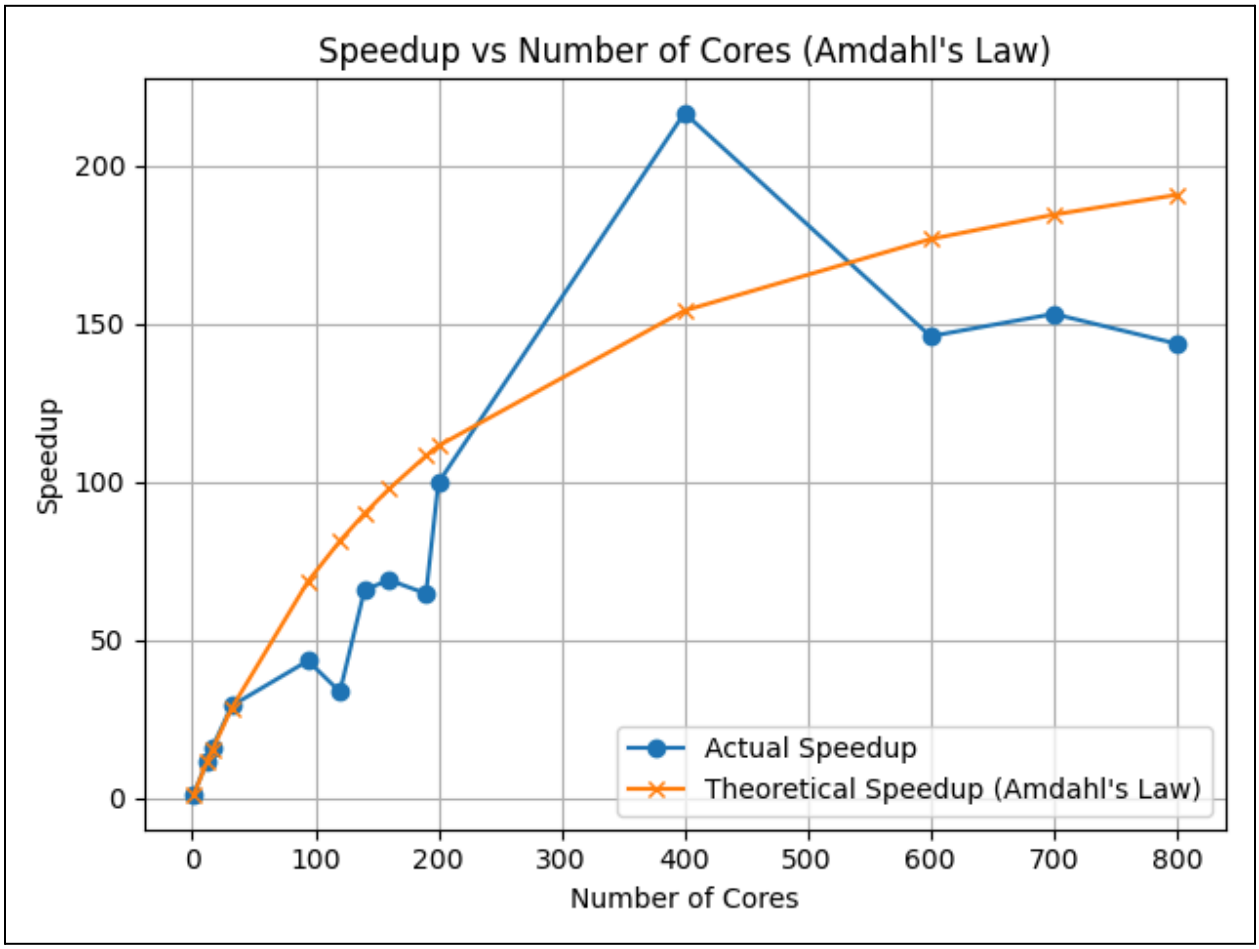
Result!!!!!

```
[ls565@login02 OS_project]$ tail -f mpi.out
----1 core-----
Pi value estimation: 3.141519
Parallel execution time: 0.250100 seconds
----12 cores-----
Pi value estimation: 3.141896
Parallel execution time: 0.021159 seconds
----16 cores-----
Pi value estimation: 3.140909
Parallel execution time: 0.015876 seconds
----32 cores-----
Pi value estimation: 3.140067
Parallel execution time: 0.008559 seconds
----94 cores-----
Pi value estimation: 3.142987
Parallel execution time: 0.005756 seconds
----120 cores-----
Pi value estimation: 3.139797
Parallel execution time: 0.007430 seconds
----140 cores-----
Pi value estimation: 3.141024
Parallel execution time: 0.003797 seconds
----160 cores-----
Pi value estimation: 3.141008
Parallel execution time: 0.003627 seconds
----190 core-----
Pi value estimation: 3.140679
Parallel execution time: 0.003867 seconds
----200 cores-----
Pi value estimation: 3.146057
Parallel execution time: 0.002509 seconds
----400 cores-----
Pi value estimation: 3.157088
Parallel execution time: 0.001155 seconds
----600 cores-----
Pi value estimation: 3.154779
Parallel execution time: 0.001713 seconds
----700 cores-----
Pi value estimation: 3.160340
Parallel execution time: 0.001634 seconds
----800 cores-----
```

```
Pi value estimation: 3.161212
Parallel execution time: 0.001741 seconds
^C
[ls565@login02 OS_project]$
```

Lets plot it using python

```
import matplotlib.pyplot as plt
import numpy as np

number_of_cores = [1, 12, 16, 32, 94, 120, 140, 160, 190, 200, 400, 600,
700, 800]
parallel_execution_times = [0.250100, 0.021159, 0.015876, 0.008559,
0.005756, 0.007430, 0.003797, 0.003627, 0.003867, 0.002509, 0.001155,
0.001713, 0.001634, 0.001741]
serial_execution_time = parallel_execution_times[0]

# calculating speedup
speedup = [serial_execution_time / time for time in
parallel_execution_times]

P = 0.996  # 99.6% parallelizable code
theoretical_amdahl_speedup = [1 / ((1 - P) + P / N) for N in
number_of_cores]

plt.plot(number_of_cores, speedup, label='Actual Speedup', marker='o')
plt.plot(number_of_cores, theoretical_amdahl_speedup, label='Theoretical
Speedup (Amdahl\'s Law)', marker='x')

plt.xlabel('Number of Cores')
plt.ylabel('Speedup')
plt.title('Speedup vs Number of Cores (Amdahl\'s Law)')
plt.legend()
plt.grid(True)
plt.show()
```



There you have it!!!!!!!

- ★ As you can see that the performance increased up to 400 cores and after that it decreased because of the overhead.
- ★ Theoretical speed is shown by the orange plot.
- ★ You might notice sudden drops between 100 - 200 cores, that is because the cluster might have other factors which can contribute to load like multiple users using at that time.
- ★ Also note that since there is no serial part in my code, I am taking 99.6% as parallelizable.
- ★ This study shows that after a certain number of cores, the overhead takes over and the performance will be decreased. So we have to find the sweet spot where our performance is at the maxima.

References

- *NJIT's HPC: https://hpc.njit.edu/*
- *PSC youtube: MPI video https://www.youtube.com/watch?v=Jt2LdbHU97g*
- *MPI slides: https://www.psc.edu/resources/training/mpi-workshop/*
- *TACC youtube:*
    - *CVW: https://cvw.cac.cornell.edu/parallel/intro/index*
- *SDSC youtube parallel concepts:* *Parallel Computing Concepts*
- *https://www.open-mpi.org/video/?category=general*
- *Amdahl's Law:*
    - *https://www.researchgate.net/figure/The-shape-of-a-typical-speed-up-graph-and-of-Amdahls-law-see-Equation-26-p-opt-is_fig6_37412792*
    - *https://www.researchgate.net/publication/37412792_Parallelization_of_population-based_evolutionary_algorithms_for_combinatorial_optimization_problems*
- https://mauriciocely.github.io/blog/2020/08/05/estimating-pi-using-the-monte-carlo-method/
- Open MPI community: https://www.open-mpi.org/papers/
- MPI Forum for discussions and help: https://www.mpi-forum.org/