OUTLINES

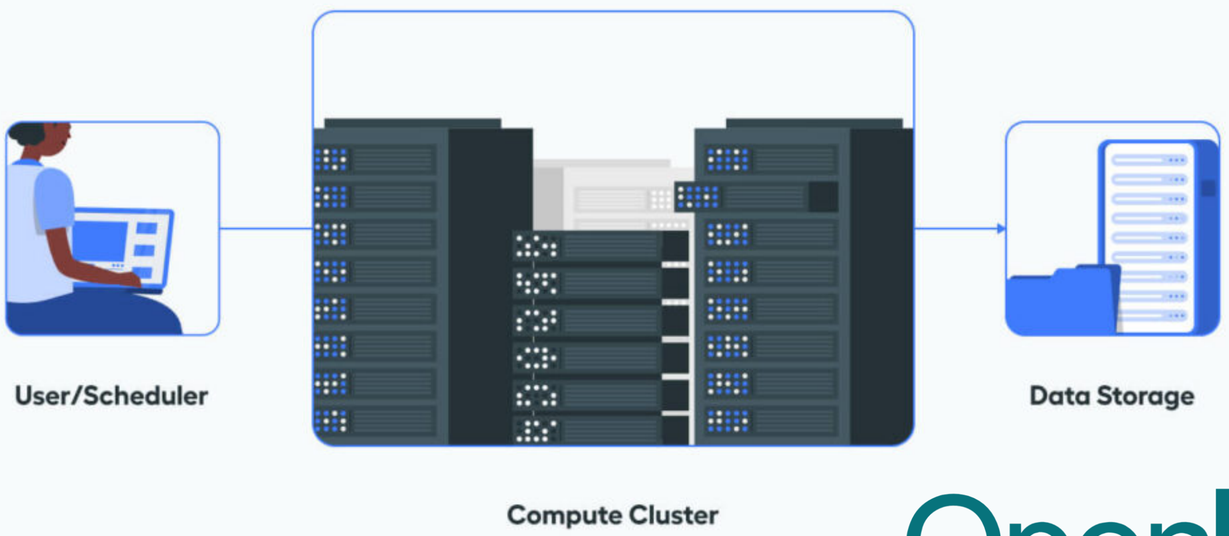
Parallel Programming(CDSC 604)

Mesfin Diro Chaka

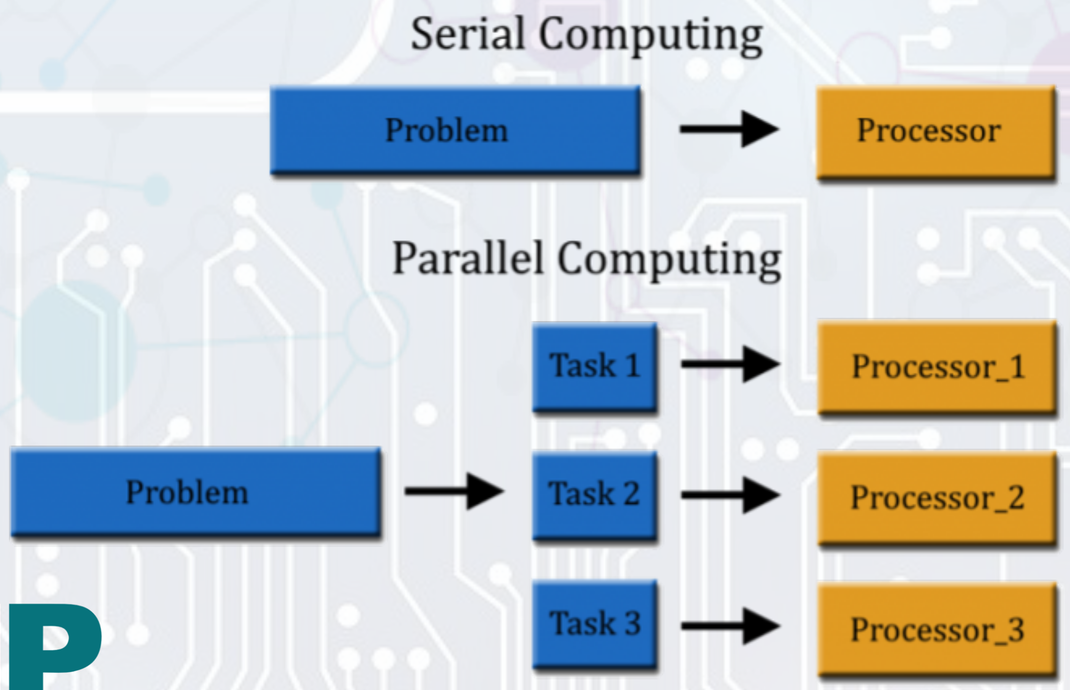
Computational Data Science Program

Addis Ababa University

March, 2024



OpenMP



Parallel computing

Part 1: Lecture 3

Cluster computing: Factors to measure performance of parallel programs

- **Question:**

1. **What do you say if Absolute Speedup (S_{an}) is less than one?**
2. **What do you say if Absolute Efficiency $(E_{an}) = 1$**
3. **What do you say if Absolute Efficiency$(E_{an}) > 1$**
4. **What do you observe by looking at Relative Speedup (S_{rn}) for different value of N**
5. **What do you observe by looking at Relative Efficiency (E_{rn}) for different value of N**

Programming with MPI

Part 1: Lecture3

Introduction to MPI

- In this section we will cover:
  - Introduction to MPI
    - Steps in MPI implementation
    - Installing OpenMPI
    - Testing OpenMPI

Programming with MPI

Part 1: Lecture 3

Message Passing Programming Paradigm

- Parallel program = program composed of tasks(processes) which communicate to accomplish an overall computational goal
- Each processor in a message-passing program runs a sub-program
  - Written in a conferential sequential language
  - All variables are private
  - Communication via special subroutine calls
- MPI refers primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.
- It is a language-independent communications protocol used to program parallel computers.

Programming with MPI

Part 1: Lecture 3

Why Use MPI?

- Message passing now mature as programming paradigm:
  - Well understood
  - Efficient match to hardware
  - Many application
- Full range of desired features:
  - Modularity
  - Access to peak performance
  - Portability
  - Heterogeneity
  - Subgroups
  - Topologies
  - Performance measurement tools

Programming with MPI

Part 1: Lecture 3

Introduction to MPI

- MPI is the first standardized, vendor independent, message passing library.
- MPI closely match the design goals of portability, efficiency, and flexibility
- It become the "industry standard" for writing message passing programs on HPC platforms.
- The first standard for Message passing Interface (MPI-1) is released in 1994 and followed with subsequent revisions.
- MPI Venders that played a significant role in advancing the adoption and optimization of MPI across a wide range of computing platforms:
  - IBM, Intel, TMC, SGI, Seiko, Cray, Convex, Ncube, ….
- Library contributed to the MPI ecosystem:
  - PVM, P4, Zipcode, TCGMSG, Chameleon, Express, Linda, DP(HKU), PM(Japan),
  - AM(Berkeley), FM(HPVM at Illinois)

Programming with MPI

Part 1: Lecture 3

Introduction to MPI

- The message passing model demonstrates the following characteristics:
  - A set of tasks that uses their own local memory during computation.
  - Multiple tasks can reside on the same physical machine as well as across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

Programming with MPI

Part 1: Lecture 3

Reason for using MPI

- Standardization - MPI is the only message passing library that can be considered as a standard. It is supported on virtually by all HPC platforms
- Portability - There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard
- Performance Opportunities - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.
- Functionality - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
- Availability - A variety of implementations are available, both vendor and public domain.

Programming with MPI

Part 1: Lecture 3

MPI Implementations

- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are embedded in the source code.
- The programmer is responsible for determining all parallelism.
- A large number of message passing libraries are available since the 1980s.
- Message Passing usually thought of in the context of distributed memory, parallel computers
- MPI program can be Single program or Multiple program
- However, the same code can be run well on
  - A shared memory parallel computer
  - A network of workstations
  - A collection of heterogeneous (different architecture) processors
  - Even on a single workstation

Programming with MPI

Part 1: Lecture 3

Steps in MPI Implementation

- In manual approach of implementing parallel program, there are steps to follow which fairly guide the programmer to successfully accomplish the job.
- However, there is no single best rule (method) to do manual parallelization.

Programming with MPI

Part 1: Lecture 3

Step 1: Understand the Problem and the Program

- Undoubtedly, the first step in developing parallel software is to understand the problem that you wish to solve in parallel.
- If you are starting with a serial program, this necessitates to understand the implementation logic of the existing code (program)

Programming with MPI

Part 1: Lecture 3

Example of Parallelizable Problem: (Approximating PI)

- **Monte Carlo Method for π Approximation:**
  1. **Initialize Variables**
     - **Set the total number of random sample N.**
     - **Initialize the count of points inside the unit circle inside circle to zero**
  2. **Generate random points:**
     - **Repeat the following step N times:**
       - **Generate a random point (x, y) within the unit square $[0,1] \times [0,1]$**
  3. **Determine if the points are inside the Unit Circle:**
     - **For each generated point(x,y):**
       - **If $x^2 + y^2 \leq 1$ increment inside circle by one**
  4. **Compute π Approximation:**
     - **The ratio of points inside the unit circle to the total number of points is proportional to the area of the unit circle to the area of the unit square:**

$$\frac{\text{inside circle}}{N} \approx \frac{\text{Area of Circle}}{\text{Area of Square}} = \frac{\pi \cdot (1^2)}{(1^2)} = \pi$$

     - **Hence, the approximation of π is given by:**

$$\pi \approx \frac{4 \cdot \text{inside circle}}{N}$$

  5. **Output Result:**
     - **Print or return the approximation of π obtained from step 4.**

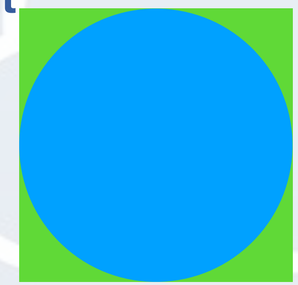
This problem is solved in parallel!

Programming with MPI

Part 1: Lecture 3

Example of Parallelizable Problem: (Approximating PI)

- **Monte Carlo Method for π Approximation:**
  1. **Initialize Variables**
     - **Set the total number of random sample N.**
     - **Initialize the count of points inside the unit circle inside circle to zero**
  2. **Generate random points:**
     - **Repeat the following step N times:**
       - **Generate a random point (x, y) within the unit square where both x and y are between 0 and 1.**
  3. **Determine if the points are inside the Unit Circle:**
     - **For each generated point(x,y) calculate the distance from origin:**
       - **If $x^2 + y^2 \leq 1$ increment inside circle by one**
  4. **Compute π Approximation:**



- **The ratio of the areas of the unit circle and the quarter of the unit square that lies within the circle (since only this quarter of the square contains points that contribute to the approximation of π:**

$$\frac{\text{Area of Circle}}{\text{Area of quarter Square}} = \frac{\pi}{4}$$

- **Hence, the approximation of π is given by:**
- $\pi \approx \dfrac{4 \cdot \text{inside circle}}{N}$

  5. **Output Result:**
     - **Print or return the approximation of π obtained from step 4.**

This problem is solved in parallel!

Programming with MPI

Part 1: Lecture 3

Parallelizable Problem: (Approximating PI) Serial version

```
import random
def approximate_pi(num_samples):
    points_inside_circle = 0
        for _ in range(num_samples):
        # Generate random point coordinates within the unit square
        x = random.uniform(0, 1)
        y = random.uniform(0, 1)
        # Check if the point lies within the unit circle
        if x**2 + y**2 <= 1:
            points_inside_circle += 1
     # Approximate pi using the ratio of points inside the circle to total points
    pi_approx = 4 * (points_inside_circle / num_samples)
    return pi_approx
if __name__ == "__main__":
    num_samples = 1000000  # Number of random samples
    # Compute the approximation of pi
    pi_approx = approximate_pi(num_samples)
    # Output the result
    print("Approximation of pi:", pi_approx)
```

Programming with MPI

Part 1: Lecture 3

Example of a Non-parallelizable Problem:

- Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula: $F(k + 2) = F(k + 1) + F(k)$.
  - This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown above, entail dependent calculations rather than independent ones.
- The calculation of the k + 2 value uses those of both k + 1 and k.
- These three terms cannot be calculated independently and therefore, not in parallelizable.

Programming with MPI

Part 1: Lecture 3

Step 3: Design the parallel Solution

- In order to write programmer directed parallel program we need to identify the program's *hotspots* location, the *bottle necks*, *inhibitors*, and the *algorithm* used.
- Hot Spots
- Knowing the hot spots includes identifying where most of the real work is being done
- The majority of scientific and technical programs usually accomplish most of their work in a few places)
- Profilers and performance analysis tools can help in identifying the hot spots if we have the equivalent serial codes.
- Once identified we need to focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.

Programming with MPI

Part 1: Lecture 3

Step 3: Design the parallel Solution

- In order to write programmer directed parallel program we need to identify the program's *hotspots* location, the *bottle necks*, *inhibitors*, and the *algorithm* used.
- Bottlenecks
  - are the areas that are disproportionately slow, or cause parallelizable work to halt or be deferred. (for example, I/O is usually something that slows a program down)
  - it may be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow area
- Inhibitors refers to issues that affect parallelization.
  - Some inhibitors include data dependency as demonstrated by the Fibonacci sequence above.
  - Finally we should design the parallel algorithm (data parallel or instruction parallel or both)

Programming with MPI

Part 1: Lecture 3

Step 4: Implement the parallel Algorithm

- The next step is to implement the parallel solution using the selected parallel programming model

Programming with MPI

Part 1: Lecture 3

Step 5: Debug and test the Program

- This is the time that we need to sit and observe logical flaws that will lead into a wrong logical conclusion
- We should carefully see all the logic implemented are correct as per the requirement stated in the algorithm design

Programming with MPI

Part 1: Lecture 3

Step 6: Submit the job to the parallel program manager

- This enable you to run your program in the HPC environment.
- We should be able to know how to submit parallel jobs into the HPC resources.

Programming with MPI

Part 1: Lecture 3

Step 7: Analyze the result

- This enable you to know how successful you are

Programming with MPI

Part 1: Lecture 3

Installing OpenMPI

- OpenMPI is Open source implementation of MPI standards for use in cluster environment
- Got the the openMPI download site(https://www.open-mpi.org/software/ompi/v5.0/
- Download the stable version for your machine say openmpi-5.0.2.tar.bz2
- Tar the file
  - tar -xjvf openmpi-5.0.2.tar.bz2
- Go to the folder openmpi-5.0.2
  - cd openmpi-5.0.2

Programming with MPI

Part 1: Lecture 3

Installing OpenMPI

- if you have access to /usr/local (root previlage) you don't need to specify the option —prefix
  - ./configure
  - make
  - make install
- Otherwise
  - Make installation directory if you don't have access to /usr/local
  - ./configure —prefix ~/openmpi
  - make
  - make install

Programming with MPI

Part 1: Lecture 3

Test Program with OpenMPI in Fortran

- Modify the environment variable PATH and LD_LIBRARY_PATH
- For CSHELL users open the file ~/.cshrc and add the following into the file
  - Open file `~/.cshrc` for editing
  - Add the following lines to the file:
    - setenv PATH ${PATH}:BinPath
    - setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:LibPath
  - Close the file and execute the command
    - Source ~/.cshrc
- For Bash Shell open the file ~/.bashrc and add the following into the file
  - export PATH=${PATH}: BinPath
  - export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:LibPath
  - Close the file and execute the command
    - source ~/.bashrc

Programming with MPI

Part 1: Lecture 3

Test Program with OpenMPI in Fortran

```
program sample
   include 'mpif.h'
   integer :: ierr
   call MPI_Init(ierr)
    write(*,*) 'Hello world'
    call MPI_Finalize(ierr)
end program sample
```

Programming with MPI

Part 1: Lecture 3

Test Program with OpenMPI in Fortran

- Compile the program as
  - Mpif90 sample.f90 -o sample.x
- Running the program with N number of processors
  - mpiexec -n 4 sample.x
  - mpiexec —use-hwthread-cpus -n 8 ./sample.x
  - Mpirun —oversubscribe -np 16 ./sample.x

Programming with MPI

Part 1: Lecture 3

Parallelizable Problem: (Approximating PI) MPI version

```
from mpi4py import MPI
import random
def approximate_pi(num_samples):
    # Initialize MPI
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    # Initialize variables to count points inside the unit circle
    local_inside_circle = 0
    # Perform Monte Carlo simulation
    for _ in range(num_samples // size):
        # Generate random points within the unit square
        x = random.random()
        y = random.random()
        # Check if the point lies inside the unit circle
        if x**2 + y**2 <= 1:
            local_inside_circle += 1
    # Gather the count of points inside the unit circle from all processes
    total_inside_circle = comm.reduce(local_inside_circle, op=MPI.SUM, root=0)
    # Master process calculates the final approximation of pi
    if rank == 0:
        pi_approx = (4 * total_inside_circle) / num_samples
        return pi_approx
    else:
        return None
if __name__ == "__main__":
    num_samples = 1000000  # Number of random samples
    # Compute the approximation of pi
    pi_approx = approximate_pi(num_samples)
    # Output the result
    if pi_approx is not None:
        print("Approximation of pi:", pi_approx)
```

THANK YOU!