Intro to MPI using Python: Parallel Theory & MPI Overview

November 17, 2024

Presented by:

Nicholas A. Danes, PhD

Computational Scientist

Research Computing Group, Mines IT

Preliminaries

- HPC Experience (one of these):
  - Know the basics of
    - Linux Shell
    - Python 3
    - Scientific Computing
  - Active HPC User
    - Mines specific: Wendian, Mio
    - Off-premise: Cloud, NSF Access, CU Boulder Alpine, etc.
  - Previously taken our “Intro to HPC” workshop
    - Offered once per semester

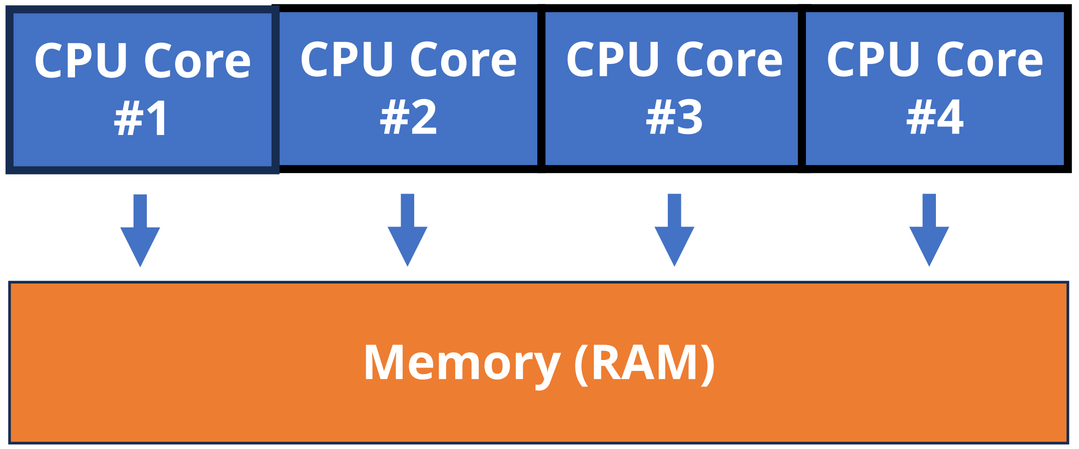
Review of Parallel vs Serial Computing

- When a program uses a single process (“task”) with 1 core (“cpu”), we say it is a **serial computing** program.
- When a program uses multiple cores, we say it is a **parallel computing** program**.**
- Typically, we try to optimize a serial computing program before trying to write it in parallel
- For this workshop, we’re going to assume we are well equipped to deal with the serial code situation

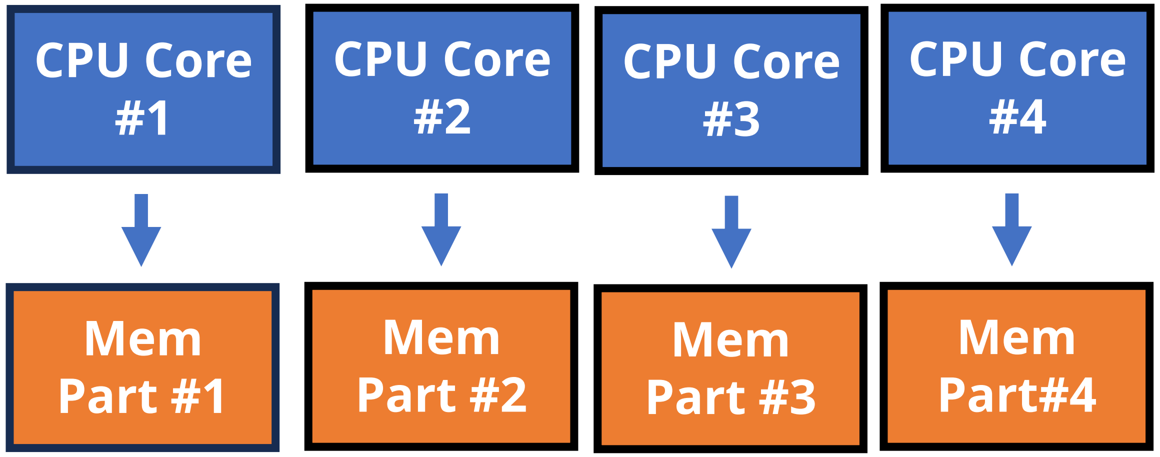
Parallel Programming Models

- Shared vs Distributed Memory Programming
  - Shared (e.g. OpenMP)
    - All CPU cores have access to the same pool of memory
    - Typically, all CPU cores are on the same CPU node
    - Ideal for multi-threaded loops
  - Distributed-memory program (e.g. MPI)
    - Each CPU core is given access to a specific pool of memory, which may or may not be shared
    - A "communicator" designates how each CPU core can talk to another CPU core
    - CPU cores do not have to live on the same CPU node

Shared Memory Parallelism:
1 task, 4 threads



Distributed Memory Parallelism:
4 tasks, 1 thread per task



Overview of MPI

- MPI stands for **m**essage-**p**assing **i**nterface, standard provided as a library for exchanging data (called messages) between objects.
- Different libraries have implemented the MPI standard:
    - OpenMPI
    - MPICH
    - Intel MPI
- Typically used with C, C++ and Fortran
- Objects that can be used to send messages are separated by memory
    - Can be entire CPU nodes, or CPU cores (or even a GPU!)
    - By breaking up by memory of each tasks, a rank can send messages theoretically anywhere as long as there is another layer of network communication
        - MPI most commonly **uses Infiniband** for node-to-node communication
        - Intra-node communication uses CPU architecture
            - Called **vader/BTL** on OpenMPI
- There are many moving parts involving networking for MPI
    - For more information: easybuild_tech_talks_01_OpenMPI_part2_20200708.pdf (open-mpi.org)

Heuristics for writing MPI Programs: Overview

- Typically, MPI programs take a **single program, multiple data (SPMD) model approach**
  - Single program: Encapsulate all desired functions and routines under one program
  - Multiple data: The single program is duplicated with multiple copies of data, and runs on the system each on its own **process.**
- Think about your largest data size and how it can be broken up into smaller chunks
- The multiple processes then can communicate (i.e. share data) using MPI library functions written by the user
- MPI data communication steps should be brought to a minimum, as they can slow down performance ***significantly.***

Common Use Case #1: Perfectly Parallel Computations

- **Perfectly (or trivially) parallel** programs are ones that do not require any MPI communication functions
- MPI is still useful, since it allows the program to run across more than one computer/compute node
- Examples include:
  - Matrix/Vector Addition
  - Markov Chain Monte Carlo (MCMC) Simulations

Common Use Case #2: Domain Decomposition for Partial Differential Equations

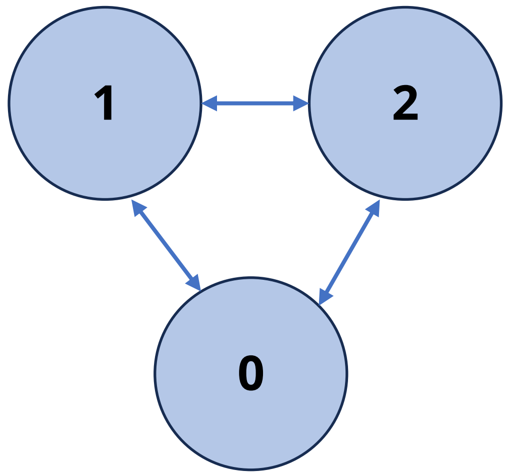
- Solving a spatial partial differential equation
  - Domain is a 1-3D mesh with multiple grid point/cells that can be broken up using **domain decomposition.**
  - Each processor contains a subset of the domain's mesh and solves the numerical problem for the differential equation on that subdomain
  - Derivatives in differential equations typically use finite difference/volume/element approximations, which require knowing values of a function around the evaluated grid point
    - This can require data from other processors
    - MPI can be used to send grid data on the edges of the decomposed domain to the other processors
      - Commonly referred to as **"ghost" cells/nodes/volumes**
    - Popular frameworks provide tracking these grid points within the mesh object
      - parMETIS,SCOTCH, PETSc, Ansys Fluent

Important MPI concepts

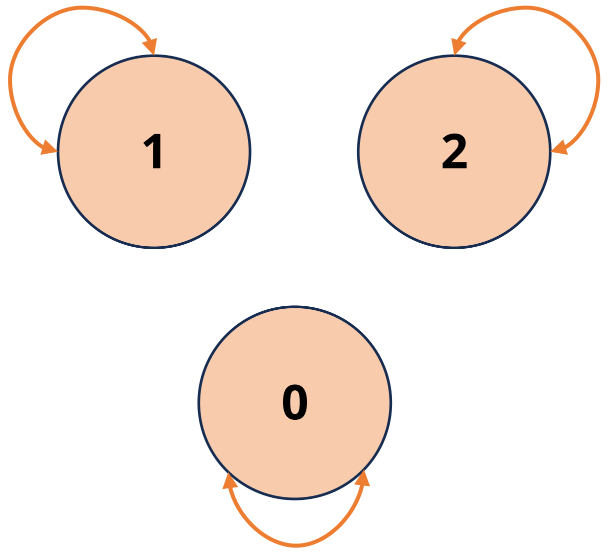
- Initialize – MPI must explicitly started in the code
  - Helps MPI identify what resources were requested
- Rank – How the number of processes are labeled/tracked
  - Common practice: ranks = # of CPU cores requested
  - Other practices: 1 compute node per rank, 1 GPU card per rank
- Size – Total number of ranks
  - In most MPI-only programs, size = number of processors requested
- Finalize – Close MPI within the program

Important MPI concepts

- Communicator – How ranks know their relation to others
  - "MPI_COMM_**WORLD**" – Every rank knows every other rank
  - "MPI_COMM_**SELF**" – Every rank knows itself
- Communication Types
  - Point-to-Point – *Synchronized* MPI function between ranks
    - Send/Receive – Every send must have a receive
    - Calls can be *blocking* or *non-blocking*
  - Collective - MPI function on all ranks
    - Broadcast – One rank sends data to all other ranks
    - Scatter – One rank sends a chunk of data to each rank
    - Gather – One rank receives data from all other ranks
  - One-sided
    - Not covering this



MPI_COMM_**WORLD**



MPI_COMM_**SELF**

MPI with Python: mpi4py

- mpi4py is a Python library that allows one to use MPI-2 C++ style bindings with Python in an object-oriented way
- Supports various python objects for the buffer interface
  - NumPy Arrays
  - Pickled Objects (lists, dictionaries, etc)
- Documentation: https://mpi4py.readthedocs.io/en/stable/
- We will be using mpi4py for this entire workshop!

mpi4py vs Other Parallel Python Options

- mpi4py alternatives – Also implements the MPI standard in python
  - PyPar: https://github.com/daleroberts/pypar
  - Scientific Python: https://github.com/khinsen/ScientificPython/
  - pyMPI: https://sourceforge.net/projects/pympi/
- Mpi4py.futures: mpi4py.futures — MPI for Python 4.0.1 documentation
  - Based on concurrent.futures (standard Python) to pool workers. Mpi4py futures lets us go across multiple nodes.
- Multiprocessing – spawns multiple processes (called workers) which can distribute work for a function
  - Easier to implement, but limited to single machine/node
  - There are some communication options: multiprocessing — Process-based parallelism — Python 3.12.2 documentation
- Dask – Provides a full parallel job scheduler framework in Python
  - More high-level and communication is more implicit
  - Task-scheduling and works well with Jupyter Notebooks
  - Can used in combination with MPI (DASK-MPI)
  - More details: https://www.dask.org/

Lab #1 (15-20 min):
1. Setting up mpi4py anaconda environment
2. Running our first programs

Today's files:

`/sw/examples/MPI_Workshop_Nov172024.tar.gz`

Basic Parallel Computing Theory

- We use parallelization to improve performance of scientific codes
  - How do we measure that?
  - Can we predict performance based on various factors?
    - Serial performance
    - Hardware
    - Problem size
  - Can we determine how the problem *scales* as we increase compute resources?

Measuring Parallel Performance

Variable	Description
P	Number of processors (“ranks“)
n	Problem size (e.g n is number of mesh cells, etc)
$T_{\{P,max\}}$	Max wall time with P processors
$T_{\{P,avg\}}$	Average wall time across P processors
$T_{\{P,m\}}$	Wall time from the m-th out of P processors
S_P	Speedup with P processors
E_P	Efficiency with P processors
β_P	Load balance with P processors

Speed-up, Efficiency, & Load-Balancing

- **Speed-up**: the ratio of the serial wall time to the parallel (with P processors) wall time

$$S_P = \frac{T_{\{1,max\}}}{T_{\{P,max\}}}$$

- When $S_P = P$**,** the speed-up is **ideal.**
- **Efficiency:**

$$E_P = \frac{S_P}{P}$$

- When $E_P = 1$**,** the efficiency is **ideal.**
- **Load-balancing:**

$$\beta_P = \frac{T_{\{P,avg\}}}{T_{\{P,max\}}}$$

When $\beta_P = 1$**,** the efficiency is **ideal.**

Basic Parallel Computing Theory: Amdahl’s Law

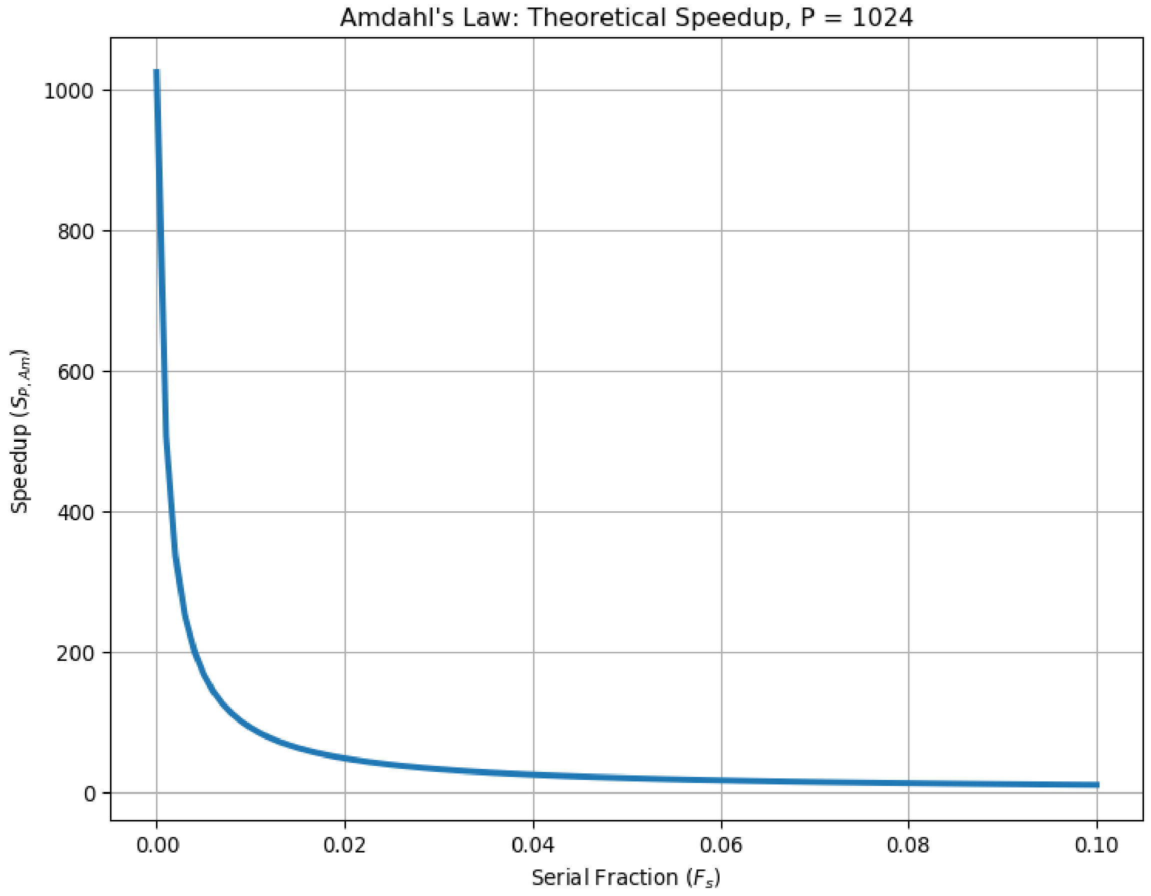
- In 1967, Gene Amdhal proposed a way to predict how much a code can scale due to a serial bottleneck [4].
  - Amdhal’s Law can be summarized with the following equation relating to speedup:

$$S_{P,Am} = \frac{1}{F_s - \frac{1 - F_s}{P}}$$

Where F_s is the *theoretical* serial fraction, the proportion of the runtime of a code that is run with only 1 processor.

Basic Parallel Computing Theory: Amdahl's Law

- Amdahl's law shows a a severe constraint to parallel scalability if a large portion of your code is in serial.
- Plot on the right shows Amdahl's Law with $P = 1024$ processors
    - If the serial fraction is about 0.5% of the runtime, then we see about a 167 times speedup, implying a 167/1024 ~ 16.3% parallel efficiency.
    - If the serial fraction is about 10% of the runtime, then the speedup drops to about 10, 10/1024 ~ about 0.97% parallel efficiency.
- **Main takeaway:** Amdahl's Law states that minimizing the time a code spends in serial is crucial for scaling up your parallel program.



Amdahl's Law Limitations

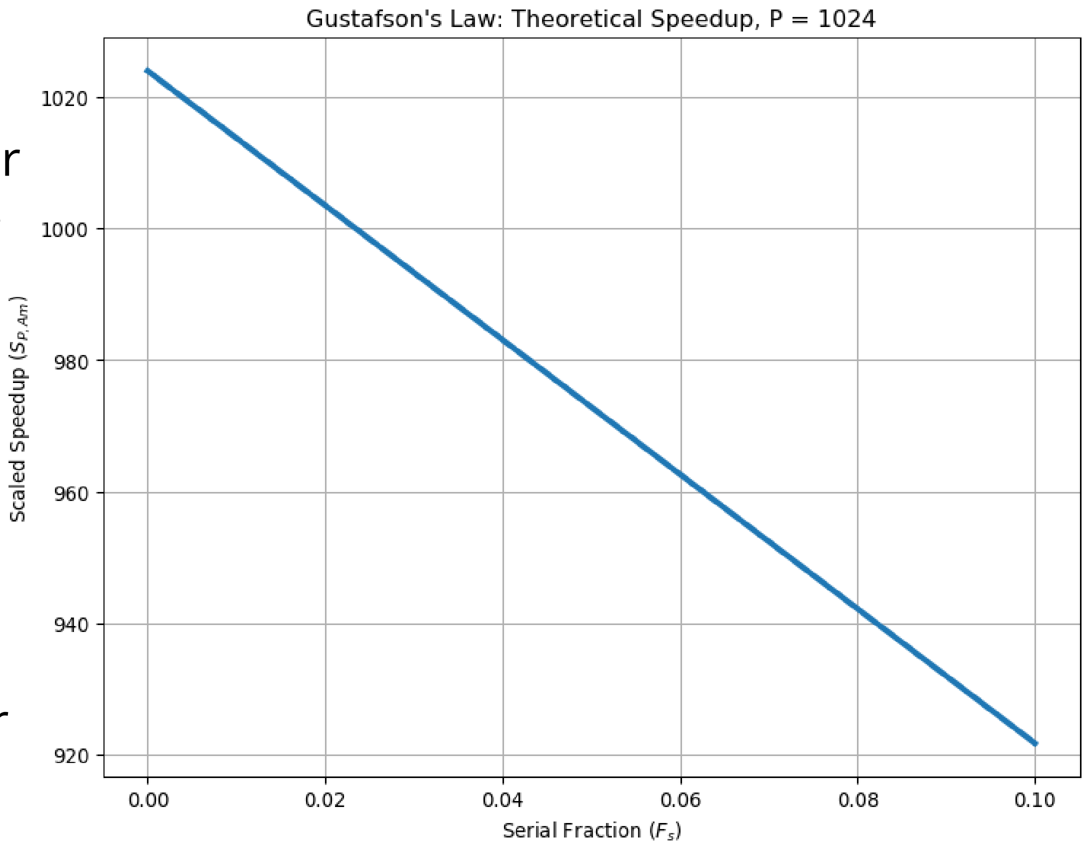
- Amdahl's Law makes many assumptions about your compute situation
  - Doesn't account for hardware limitations
    - CPU configuration (cache, memory, etc)
    - Disk performance (read/write speeds, etc)
  - The fraction of the code spend in *parallel* could also depend on the number of processors, i.e.
    $$1 - F_s = F_P = F_P(P)$$
  - It assumes that your problem size is fixed
- In practice, when performing a benchmark with increasing number of processors with a **fixed problem size**, we call this **Strong Scaling**.

Gustafson’s Law

- In response, John Gustafson argued that the assumptions from Amdahl’s Law for was not appropriate for all parallel workloads [4].
    - In particular, the serial time spent by the processor was *not* independent of the number of processors
        - More processors used on a CPU means the cores will compete for memory bandwidth
- As an approximation, Gustafson approximated speedup by assuming the parallel part of the program is linearly proportional to the number of processors:

$$S_{P,Gu} = P + (1 - P)F_s$$

- This equation is often referred to as *scaled speedup.*
- When one increases the problem size with the number of processors linearly, we call this **weak scaling.**



Lab #2 (15-20 min): Calcuating pi in parallel using Leibiniz’s formula

Today’s files:

```
/sw/examples/MPI_Workshop_Nov172024.tar.gz
```

References

- [1] https://www.cs.uky.edu/~jzhang/CS621/chapter7.pdf
- [2] https://www.youtube.com/watch?v=pDBIoil-LTk
- [3] https://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf
- [4] Gustafson, John L. “Reevaluating Amdahl’s law.” Communications of the ACM 31, no. 5 (1988): 532-533: http://www.johngustafson.net/pubs/pub13/amdahl.htm
- [5] https://xlinux.nist.gov/dads/HTML/singleprogrm.html