Intro to MPI using Python: A (Semi)-Deep Dive into mpi4py

November 21, 2024

Presented by:
Nicholas A. Danes, PhD
Computational Scientist
Research Computing Group, IT

Summary of Tuesday's Workshop

- Amdahl's Law (Strong Scaling)
  - When the problem size is fixed, the serial fraction of a parallel code can degrade parallel speedup
- Gustafson's Law (Weak Scaling)
  - The problem size scales linearly with the number of compute resources
- MPI is a C/C++/Fortran bindings library used for parallel communication between processors, compute nodes, and/or accelerators (e.g. GPUs)
  - MPI is also available through Python (mpi4py)
- Overview of MPI concepts
- Lab covering mpi4py environment setup and running over first codes without using MPI communication (embarrassingly parallel)

mpi4py: Pickled Objects vs NumPy Arrays

- When using MPI with C/C++ and Fortran, you need to use a buffer to send and receive messages
  - Common buffer types: integers, floating point, strings, etc.
- With mpi4py, you have two main options:
  - Numpy arrays (more akin to the common buffer types in C/C++)
    - Common data types: `DOUBLE, INT, COMPLEX, FLOAT, CHAR`
  - "Pickled" object
- In Python, any object can be "pickled" into byte stream that can be used with MPI communication functions
  - There is a performance cost for using pickled objects, but can be convenient to rapidly prototype
- Nomenclature differences
  - Pickled MPI functions are **l**ower cased, e.g. `comm.send`
  - NumPy MPI functions are **U**pper cased, e.g. `comm.Send`

mpi4py: Initializing the MPI environment

- Unlike C/C++ and Fortran MPI, mpi4py initializes automatically when you import mpi4py:

```
import mpi4py
```

Similarly, **MPI Finalize is also called internally** when the Python code exits!

For convenience, usually one uses the following syntax

```
from mpi4py import MPI
```

- Define variables for
  - Communicator (typically MPI_COMM_WORLD)

    ```
    comm = MPI.COMM_WORLD
    ```
  - Rank (commonly using variable name rank)

    ```
    rank = comm. rank
    ```
  - Size (commonly using variable name size)

    ```
    size = comm.size
    ```

Point to Point Communication

- Most MPI communication use some form of **send** and **receive** communication calls.
- They are **blocking** calls, which means every **receive** call from a rank needs a **send** call from that rank
- **Tags** are used to differentiate multiple send/receive calls between ranks
- **Pickled object:**

```
comm.send(data, dest=DEST_RANK, tag=TAG_NUMBER)
receiver = comm.recv(source=SOURCE_RANK, tag=TAG_NUMBER)
```

- **NumPy Array:**

```
comm.Send([data, MPI.DATATYPE], dest=DEST_RANK, tag=TAG_NUMBER)
receiver = comm.Recv([data, MPI.INT], source=SOURCE_RANK,
tag=TAG_NUMBER)
```

Lab Problem #1: Basic Parallel Matrix-Vector Multiplication

- Consider an $m\ x\ n$ matrix A being multiplied by a $n\ x\ 1$ vector $\boldsymbol{b}$:

$$A\boldsymbol{b} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

- If we have p processors, how do we split the work up?

Lab Problem #1: Basic Parallel Matrix-Vector Multiplication

- We know each row of A does a dot product with $\boldsymbol{b}$, so each of these dot products can be computed independently.

$$(a_{i1} \quad \cdots \quad a_{in}) \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = a_{i1}b_1 + a_{i2}b_2 + \ldots + a_{in}b_1, \quad i = 1,2,3 \ldots m$$

- Instead using matrix multiplication on the entire matrix A with $\boldsymbol{b}$, we can just do dot products by splitting up the work into submatrices on each processor.

Lab #1: Matrix-Vector Multiplication & Parallel Performance Lab using Strong Scaling

- Explore the mat_vec code and try to understand how the send/receive calls are operating and how the data parallelism is setup.
- Run the code for 1,2,4,8,16,32,64,128 slurm tasks
- Use the post-process.inpy jupyter notebook to analyze the results
- *(If time permits) Additional Exercise:* Do a **weak scaling** analysis!
  - Fix n, m to a size for when $p = 1$(make this smaller than the weak scaling problem, I'd say $m = n = 100$)
  - When $p = 2$, double the size of n and m.. ($m = n = 200$)
  - When $p = 4$, double the size of n and m... ($m = n = 400$)
  - Try this until it doesn't scale anymore!

Collective Communication: Broadcast

- If you want one rank to **send** *the same* data to **each other** rank simultaneously, use the **broadcast** function.

- Pickled Object

```
data = comm.bcast(data, root=sending_rank)
```

- NumPy Array

```
comm.Bcast(data, root=sending_rank)
```

Collective Communication: Scatter

- If you want one rank to **send** *different* data to **each other** rank simultaneously, use the **scatter** function.

- Pickled Object

```
data = comm.scatter(data, root=sending_rank)
```

- NumPy Array

```
comm.Scatter(send_buffer, receive_buffer, root=0)
```

Collective Communication: Gather

- If you want one rank to **receive** data from **all other** ranks simultaneously, use the **gather** function.

- Pickled Object

```
data = comm.gather(data, root=receiving_rank)
```

- NumPy Array

```
comm.Gather(send_buffer, receiving_buffer, root=receiving_rank)
```

Collective Communication: Reduce

- MPI provides a reduce collective communication for common data processing functions:
- Pickled

```
final_data = comm.reduce(data, op=MPI.OPERATION, root=receiving_rank)
```

- Numpy Arrays

```
comm.Reduce(send_buffer, [data, MPI.DATATYPE], op=MPI.OPERATION, root=receiving_rank)
```

MPI.OPERATION common options:

- Add – MPI.SUM
- Max – MPI.MAX
- Min – MPI.Min

Advanced Collective Communication

- Gatherv – Like gather, but allows for variable buffer sizes that the receiving rank obtains
- scatterv – Like scatter, but allows for variable buffer sizes with irregular sizes
- allgather – A combination of gather and broadcast
- All-to-All – A combination of scatter and gather
- All-to-allv – A combination of scatter and gather
- These are highly advanced and will not be covered in this workshop.

Mpi4py: Other Features

- Nonblocking communication
  - isend, irecv
- Barrier
  - MPI.Barrier can block communication until all ranks reach the barrier
- Parallel I/O (MPI-IO)
- One-sided communication
- Wrapping with compiled languages
  - SWIG (C++)
  - F2py (Fortran)

Using mpi4py with other libraries

- F2py – Fortran Wrapper
- Cython – Write python-style code that is converted into C and used as python module
- SWIG – C++ Bindings Wrapper
- PETsc4py – A full linear algebra library that can be used with mpi4py

Final Takeaways

- MPI is the library standard for providing parallel communication between nodes and other compute devices
- Mpi4py provides a way to learn MPI using Python, decreasing time to development for testing
- For best performance, always test your serial case
  - In this workshop, we did not always make the serial case as fast as possible in our examples
  - Using C/C++ (Cython, SWIG) and Fortran (F2py) wrappers are ways to use Python as your “glue” code with MPI, while doing compute-intensive tasks in lower-level languages
- There are more than one ways to implement a problem using MPI
  - When in doubt, use send/receive so you can track what messages are being passed
    - Gather/scatter can be a replacement if you understand how you want your data to move around
  - MPI.Reduce can make simplifying a set of data across multiple ranks quick
  - MPI.Broadcast lets you send data to all ranks quickly
- Always test your performance!
  - Strong and Weak scaling are both tools you can use
    - If you know your problem size is fixed, use *strong scaling* by increasing processors and evaluating run time until efficiency/speed-up trade offs are no longer tolerable
    - If you need to scale your problem to the processor count, use *weak scaling* to do your analysis.

Need Help?

- RC provides consultations on parallel computing
  - We won't write your code for you, but we can help you on the right path
  - Strong/Weak scaling analysis guidance
  - MPI is our preferred library for setting up parallel codes, but we are open to other libraries, including (but not limited to):
    - CUDA
    - Dask
    - Torch.distributed
    - OpenMP
- Get in touch with us here: https://outlook.office365.com/owa/calendar/CIARCTeamServices@mines0.onmicrosoft.com/bookings/

Lab #2

Exercise: Modify Matrix-Vector Multiplication to use gatherv instead of send/receive!

Exit Survey – Please fill out!

- Survey Link: https://forms.office.com/r/JWAfVphbJ3