PHY604 Lecture 27

December 2, 2025

Today’s lecture: MPI and summary

- MPI in fortran and python
- Summary of the class

MPI for distributed memory architecture

- Major limitation of OpenMP: Requires all processors share memory
    - This is not practical if we want to parallelize over 1000’s of processes

- Large supercomputer clusters have distributed memory
    - Need to manage how the work is divided and explicitly send messages from one process to the other as needed

- MPI library is standard for distributed parallel computing

How to use MPI: First get a library (usually available on clusters)

- Intel: https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html
  - Not open source but freely available
  - Optimized for intel architecture
  - For C, C++, fortran

- OpenMPI: https://www.open-mpi.org/
  - Open source
  - More architecture agnostic
  - For C, C++, fortran

- Mpich: https://www.mpich.org/
  - Similar to openmpi, easier to use but potentially lower performance

- mpi4py: https://mpi4py.readthedocs.io/en/stable/
  - For Python

MPI hello world

```fortran
program hello
  use mpi
  implicit none

  integer :: ierr, rank, nprocs

  call MPI_Init(ierr)
  call MPI_Comm_Rank(MPI_COMM_WORLD, rank, ierr)
  call MPI_Comm_Size(MPI_COMM_WORLD, nprocs, ierr)

  if (rank == 0) then
     print *, "Running Hello, World on ", nprocs, " processors"
  endif

  print *, "Hello World", rank

  call MPI_Finalize(ierr)
end program hello
```

MPI hello world

Initiate MPI computation

Determine id for process

Determine number of processes

```
program hello
   use mpi
   implicit none

   integer :: ierr, rank, nprocs

   call MPI_Init(ierr)
   call MPI_Comm_Rank(MPI_COMM_WORLD, rank, ierr)
   call MPI_Comm_Size(MPI_COMM_WORLD, nprocs, ierr)

   if (mype == 0) then
      print *, "Running Hello, World on ", nprocs, " processors"
   endif

   print *, "Hello World", rank

   call MPI_Finalize(ierr)
end program hello
```

Terminate MPI computation

Communicator handle

- Several of the commands we say take as an argument the communicator handle

- Default value is `MPI_COMM_WORLD` which identifies all processes involved in the computation

- We can also identify subsets of processes to, e.g., send messages to

Basic functions:

- `MPI_Init(ierr)`
  - Initiate the MPI computation
  - `ierr` here is optional integer for error handling
- `MPI_Finalize(ierr)`
  - Terminate the MPI computation
- `MPI_Comm_Size(COMM, nprocs, ierr)`
  - Get the number of processes
  - Input: communicator handle
  - Output: number of processes
- `MPI_Comm_Rank(COMM, mype, ierr)`
  - Get the id of the current process
  - Input: communicator handle
  - Output: ID of the process in the group
- `MPI_Send(message,count,datatype,dest,tag,COMM,ierr)`
  - Send a message to a a different process
  - Count: number of elements sent through array
  - Datatype: MPI-specific datatype passed through the array
  - Dest: Destination process
  - Tag: message tag
- `MPI_Recv(message,count,datatype,source,tag,COMM,status,ierr)`
  - Receive message from process
  - Source: Can be (MPI_ANY_SOURCE) or specific process

Some other MPI functions

- `MPI_Scatter`
  - Send data to all other processors
- `MPI_Gather`
  - Return data to root process
- `MPI_Reduce`
  - Reduce all the data via some operator to the root (e.g., to sum it)
- `MPI_Sendrecv`
  - Send and receive in the same command
- `MPI_Bcast`
  - Send the same piece of data to all processes

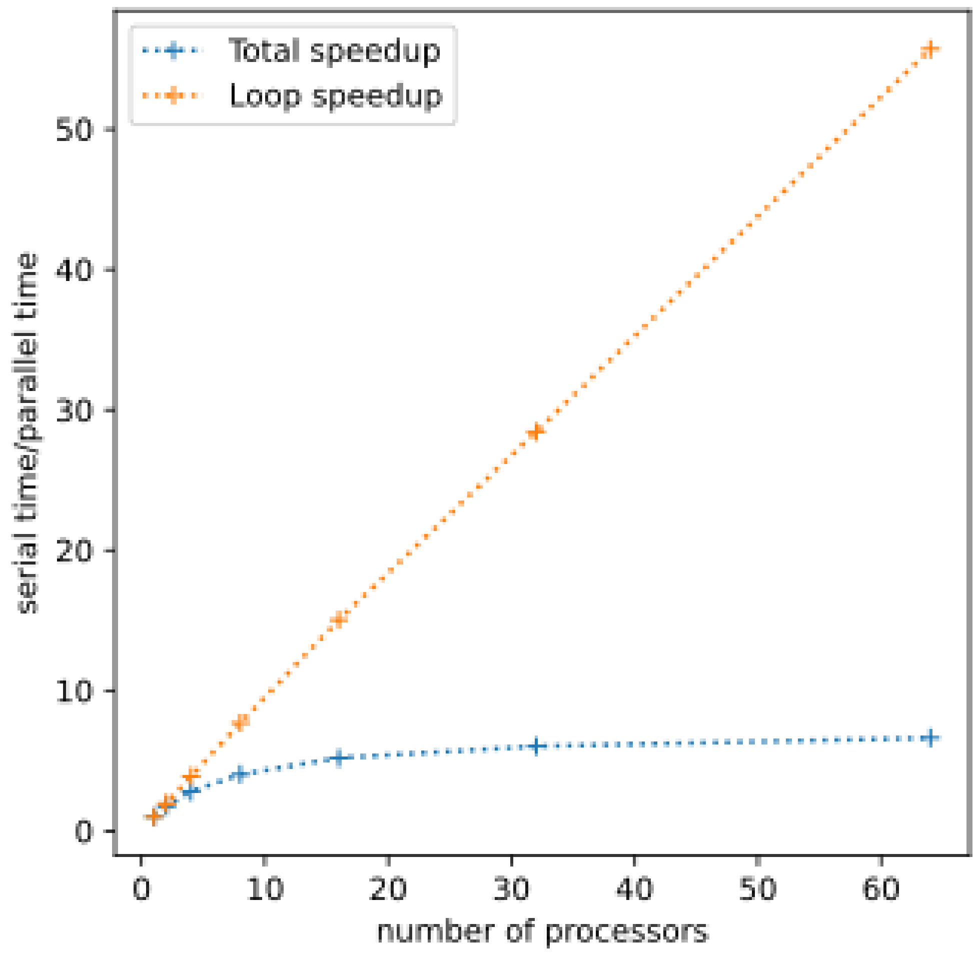
Let’s do some useful parallelization

- Example: Numerical integration of a function with the trapezoid rule

- User will enter the limits of integration and number of subintervals

- We will spread the sum over subintervals over processors using MPI

Speedup for trapezoid integration over 100 million subintervals



MPI in Python: Same ideas, similar syntax

```
from mpi4py import MPI
comm = MPI.COMM_WORLD # Communicator
rank = comm.Get_rank()# Process rank
size = comm.Get_size()# Number of processes

comm.Send(arr, dest=1)
comm.Recv(arr, source=0)
```

MPI in python: Capitalized versus uncapitalized methods

- Capitalized, e.g.,:

```
comm.Send(arr, dest=1)
comm.Recv(arr, source=0)
```

  - Closer to the C MPI methods, lower level, faster
  - Do not work with all python objects (dicts, lists, classes)
  - Use on, e.g., numpy arrays

- Uncapitalized, e.g.,:

```
comm.send({"x": 1}, dest=1)
data = comm.recv(source=0)
```

  - Higher-level, less efficient, python-friendly methods
  - Works with python objects (using pickle)

Today’s lecture: MPI and summary

- Parallel computing with MPI

- Summary of the course

Some main takeaways from what we covered:

- Floating point arithmetic:
  - No floating point calculation is exact
  - Nominally equivalent mathematical expressions can lead to very different errors
- Good-enough programming practices:
  - Use version control!
- Numerical integration/differentiation:
  - Trade continuous operators for discrete function evaluations
  - More points = more accuracy (limed by roundoff errors!), but we can also be smart about the algorithm we use to make the most out of the points we have/can calculate
  - Adaptive methods can estimate errors, guide convergence of numerical parameters, and make the most of relatively few calculations

Some main takeaways from what we covered:

- Interpolation:
  - Keep in mind the difference between interpolation and fitting
  - Limited use for noisy data
  - Cubic splines are the most popular

- Root finding:
  - Many problems throughout the course could be related to finding the root of a function
  - Accomplished via iterative methods

- Ordinary differential equations:
  - One of the powers of computation
  - Can solve a wide range of problems with some simple(ish) methods (e.g., 4th order Runge-Kutta)
  - Adaptive methods are useful here too

Some main takeaways from what we covered:

- Linear algebra:
  - Solving systems of linear equations came up often throughout the class
  - We can efficiently solve linear problems, especially if we take into account special structure in the problem
  - Can use iterative root-finding methods to generalize to nonlinear algebra
- Fast Fourier transforms
  - Useful for signal processing and solving PDEs
  - Very efficient algorithms have been developed
- Curve fitting:
  - Keep in mind the difference between curve fitting and interpolation
  - General least squares with well-chosen basis functions is a versatile method
  - Beware of overfitting

Some main takeaways from what we covered:

- Partial differential equations
  - This is a huge topic, we just scratched the surface
  - The technique used depends on the type of PDE (unlike ODEs)
  - Numerical stability is a crucial concern

- Monte Carlo and stochastic methods:
  - Many physical processes involve some element of randomness
  - Can also be used in a variety of other areas when deterministic methods fail
  - Stochastic methods can be mixed with deterministic ones, e.g., genetic algorithms

- Neural networks
  - Allows computers to do pattern recognition in problems where we have an incomplete or unsophisticated physical model, but a lot of data
  - More flexibility in the model gives better description of training set
  - Bigger training set gives better description of unknown data

Some main takeaways from what we covered:

- Parallel computing
  - Needed to take advantage of advances in computational power
  - Approach is based on the hardware and the limiting parts of your program

After class tasks

- If you have turned in HWs late, send me an email when you have finished turning them is so I can grade them

- Remember final projects are due Dec. 16, presentations are Dec. 16, 2025, 2:15pm-5:00pm (in B131)
    - Send me an email if you have any questions about the final projects