Introduction to
Parallel Programming with MPI

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing, CST
Associate Director, Institute for Computational Science
Assistant Vice President for High-Performance Computing

a.kohlmeyer@temple.edu
Parallel Computing

- Why parallel computing?
  - My problem takes too long to solve
  - My problem is too big to fit into a single computer
  - I have a lot of (small) problems

- Types of parallelism:
  - Bit level parallelism (Vector unit)
  - Instruction level parallelism (Super scalar CPU)
  - Data parallelism (work on different data concurrently)
  - Task parallelism (do different things concurrently)
Types of Parallel Problems

- Embarrassingly parallel vs. mostly serial
- Distributed data vs. replicated data
- Compute bound vs. I/O or memory bound
- Linear scaling vs. high algorithmic complexity
- Fast turnaround required (capability) vs. high throughput required (capacity)

Most problems run on supercomputers are somewhere in the middle, i.e. they can benefit from larger/better hardware, but it is **hard work**
Parallelization Strategies

- Parallelization over data or domains:
  - Each parallel worker operates on subset of data
  - Each parallel worker performs the same operation
  - Replicated data $\Rightarrow$ each process has the full set
  - Distributed data $\Rightarrow$ each process has a part of it
  - Flow of control similar to serial code
  - Efficient with large data sets

- Parallelization over tasks:
  - Each parallel worker performs different operations
  - Load balancing difficult
A High-Performance Problem

A PARALLEL SOLUTION!
Limitations of Parallel Computing

- Amdahl's law: fraction of non-parallel code is theoretical limit to parallel speedup
  => strong scaling benchmarks
- Degree to which tasks/data can be subdivided is limit to concurrency and parallel execution
- Data dependencies: certain operations cannot start before others have been completed
- Communication and synchronization overhead: parallel software needs additional code pass information around and arrange execution
Challenges in Parallel Computing

- Hardware is getting “more parallel”, e.g. advanced vector instructions, multi-core CPUs, GPUs/Accelerators
  => we have to “find” more concurrency

- Parallelization has to become “topology aware”
  => coarse and fine grained parallelization has to be mapped to the topology to reduce memory and i/o contention

- Efficient serial algorithms may not parallelize
  => may need to re-implement code for parallel
Interconnect vs. Performance

- Gigabit ethernet (or 10 Gigabit ethernet):
  - Ubiquitous, affordable (GigE) network (125 MB/s)
  - Uses TCP/IP protocol for reliable communication, high latency (~0.5ms), en-/decoding overhead
- Infiniband, Myrinet
  - 10 GigE “fabric”, but different “encoding” (DDR,QDR)
  - Switched network (single switch, CLOS network)
  - RDMA (remote direct memory access)
  - low communication latency (~1 µs) (4xQDR:12GB/s)
What is MPI?

- A standard, i.e. there is a document describing how the API (= constants & subroutines) are named and should behave; multiple “levels”, MPI-1 (basic), MPI-2 (advanced), MPI-3 (new)
- A library or API to hide the details of low-level communication hardware and how to use it
- Implemented by multiple vendors
  - Open source and commercial versions
  - Vendor specific versions for certain hardware
  - Not binary compatible between implementations
Goals of MPI

- Allow to write software (source code) that is portable to many different parallel hardware. i.e. agnostic to actual realization in hardware
- Provide flexibility for vendors to optimize the MPI functions for their hardware
- No limitation to a specific kind of hardware and low-level communication type. Running on heterogeneous hardware is possible.
- Fortran77 and C style API as standard interface
MPI Program Design

- Multiple and separate processes (can be local and remote) concurrently that are coordinated and exchange data through “messages” => a “share nothing” parallelization

- Best for coarse grained parallelization

- Distribute large data sets; replicate small data

- Minimize communication or overlap communication and computing for efficiency => Amdahl's law: speedup is limited by the fraction of serial code plus communication
Phases of an MPI Program

1) Startup
   - Parse arguments (mpirun may add some)
   - Identify parallel environment and rank in it
   - Read and distribute all data

2) Execution
   - Proceed to subroutine with parallel work
     (can be same of different for all parallel tasks)

3) Cleanup

NOTE: this sequence may be run only once
MPI in C versus MPI in Fortran

The programming interface (“bindings”) of MPI in C and Fortran are closely related (wrappers for many other languages exist).

**MPI in C:**
- Use '#include <mpi.h>' for constants and prototypes
- Include only once at the beginning of a file

**MPI in Fortran:**
- Use 'include “mpif.h”' for constants
- Include at the beginning of each module
- All MPI functions are “subroutines” with the same name and same order and type of arguments as in C with return status added as the last argument
How much MPI do we need?

- A fully functional MPI program can be written using only 6 MPI functions:
  - MPI_INIT
  - MPI_COMM_SIZE
  - MPI_COMM_RANK
  - MPI_BCAST or MPI_SEND
  - MPI_REDUCE or MPI_RECV
  - MPI_FINALIZE
MPI Startup / Cleanup

- Initializing the MPI environment:
  - **CALL MPI_INIT(STATUS)**
  - Status is integer set to MPI_SUCCESS, if operation was successful; otherwise to error code

- Releasing the MPI environment:
  - **CALL MPI_FINALIZE(STATUS)**

**NOTES:**
- **All** MPI tasks have to call **MPI_INIT & MPI_FINALIZE**
- **MPI_INIT** may only be called once in a program
- No MPI calls allowed outside of the region between calling **MPI_INIT** and **MPI_FINALIZE**

Institute for Computational Molecular Science
Communicator Size and Process Rank

- A “communicator” is a label identifying a group of processors that are ready for parallel computing with MPI.
- By default the `MPI_COMM_WORLD` communicator is available and contains all processors allocated by `mpirun`.
- **Size**: How many MPI tasks are there in total?
  - `CALL MPI_COMM_SIZE(comm, size, status)`
  - After the call the integer variable `size` holds the number of processes on the given communicator.
- **Rank**: What is the ID of “me” in the group?
  - `CALL MPI_COMM_RANK(comm, rank, status)`
  - After the call the integer variable `rank` holds the ID or the process. This is a number between 0 and `size-1`. 
“Hello World” a la MPI

Fortran

PROGRAM hello
  INCLUDE ‘mpif.h’
  INTEGER :: ierr, rank, size
  CALL MPI_INIT(ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
  PRINT*, 'I am ', rank, ' of ', size
  CALL MPI_FINALIZE(ierr)
END

Important: call MPI_INIT before parsing arguments
Sending Data With MPI

- To exchange data between processes we need to tell MPI some information about the data
  - Where does it come from? Where does it go to? => which rank is sending or receiving the data
  - Of what type (integer, real, ...) is the data? => use constants `MPI_INTEGER`, `MPI_REAL`
  - How many elements are to be sent? => even a single variable is a list with 1 element
  - What communicator is used? => what (sub) group of processors is participating
    => `MPI_COMM_WORLD` is defined by default
Calling MPI_BCAST

MPI_BCAST(buffer, count, type, sender, comm, err)
  buffer:     buffer with data
  count:      number of data items to be sent
  type:       type (=size) of data items
  sender:     rank of sending processor of data
  comm:       group identifier, MPI_COMM_WORLD
  err:        error status of operation

• NOTES:
  • buffers must be large enough (can be larger)
  • Data type must match (MPI does not check this)
  • all ranks that belong to the communicator must call this
  • Program waits until all processors are ready to communicate => synchronization point.
Calling MPI_REDUCE

MPI_REDUCE(in,out,count,type,op,receiver,comm,err)

in: data to be sent (from all)
out: storage for reduced data (on receiver)
count: number of data items to be reduced
type: type (=size) of data items
op: reduction operation, e.g. MPI_SUM
receiver: rank of sending processor of data
communicator: group identifier, MPI_COMM_WORLD
err: error status or MPI_SUCCESS

• NOTES:
  • A reduction is to combine information from multiple processors to one receiving processor
  • Reduction operations can be: MPI_SUM, MPI_MAX, MPI_MIN, MPI_LAND, MPI_LOR, and many more
The PI program

Algorithm:

\[
\int_{0}^{1} \frac{4}{1 + x^2} \, dx = 4(\arctan(1) - \arctan(0)) = \pi
\]

Integrate, i.e determine area under function numerically using slices of \( h \times f(x) \) at midpoints
External MPI Resources

Here are some links to tutorials and literature

- CI-Tutor at NCSA: http://www.citutor.org/
- MPI reference and mini tutorial at LLNL: http://computing.llnl.gov/tutorials/mpi/
- Designing and Building Parallel Programs, by Ian Foster: http://www.mcs.anl.gov/~itf/dbpp/
- MPI standards: http://www.mpi-forum.org/
- OpenMPI: http://www.open-mpi.org