Parallel Computing

Lecture 15, Computational Astrophysics (ASTR660)

Hsiang-Yi Karen Yang, NTHU, 12/22/2022

Announcements

- ***HW6 is due TODAY.*** Late submission within one week will receive ***75%*** of the credit
- The final ***oral presentations (15+5 mins per person)*** are scheduled on ***12/29*** and ***1/5***. The final ***product*** of the project is due on ***1/12***.
- The ***evaluation form*** for the oral presentations has been posted on eLearn. Please check it out and prepare accordingly!

Date	Time	Name	Title
12/29/22	14:20-14:40	吳耕緯	3D N-body with fast multipole method
	14:40-15:00	李佳倫	Effects of AGN Quasar Feedback in Galaxy Clusters
	15:00-15:20	簡嘉成	Improving my magnetohydrodynamic simulation for non-ideal plasma fluid
	15:30-15:50	凌志騰	A gravitational SPH simulation of galaxy collision from scratch
	15:50-16:10	鄭文淇	Simulate impact-driven atmospheric loss using N-body simulation
	16:20-16:40	龔一桓	Ram pressure striping of galaxy cluster: Can a subcluster be assumed as gasless
	16:40-17:00	劉一璠	Simulate the Kelvin-Helmholtz instability by using FLASH code
1/5/23	14:20-14:40	胡英祈	Modeling Dust Polarization in Protoplanetary Disk at (Sub)millimeter Wavelengths
	14:40-15:00	周育如	Multi-gaussian fit, EW, and line ratio for [O I] 5577 and 6300 from DG Tau
	15:00-15:20	考司圖巴	Compare MCMC based fitting with phenomenological models for X-ray Spectra
	15:30-15:50	郭弈翔	Calculate the entropy of a subregion using Python computation
	15:50-16:10	謝明學	simulate the tidal locking phenomenon
	16:20-16:40	張修瑜	Three-body problem in the Moon, Earth, and the Sun.
	16:40-17:00	石郡翰	Using the Mechanism of AIRES to Simulate the Propagation of Cosmic Rays in the universe

Evaluation form for final oral presentations

	Poor				Excellent
CONTENTS OF PROJECT PRESENTATION	**1**	**2**	**3**	**4**	**5**
Were the motivations/goals of the project clearly described and reviewed? .....,	,	,	,	,	,
Did the presenter clearly describe what has been achieved? .....,	,	,	,	,	,
Please evaluate the amount of the efforts* being made. .....,	,	,	,	,	,
Please evaluate the quality of the efforts being made .....,	,	,	,	,	,
Were the scientific results and discussions clearly presented?.....,	,	,	,	,	,
Were the proposed scientific questions addressed by this study? .....,	,	,	,	,	,
Did the presenter provide estimates of the performance** of the numerical method? ,	,	,	,	,	,

***Efforts include achievements and/or solutions for difficulties encountered**

****This includes precision, accuracy, speed, and/or comparison with previous works**

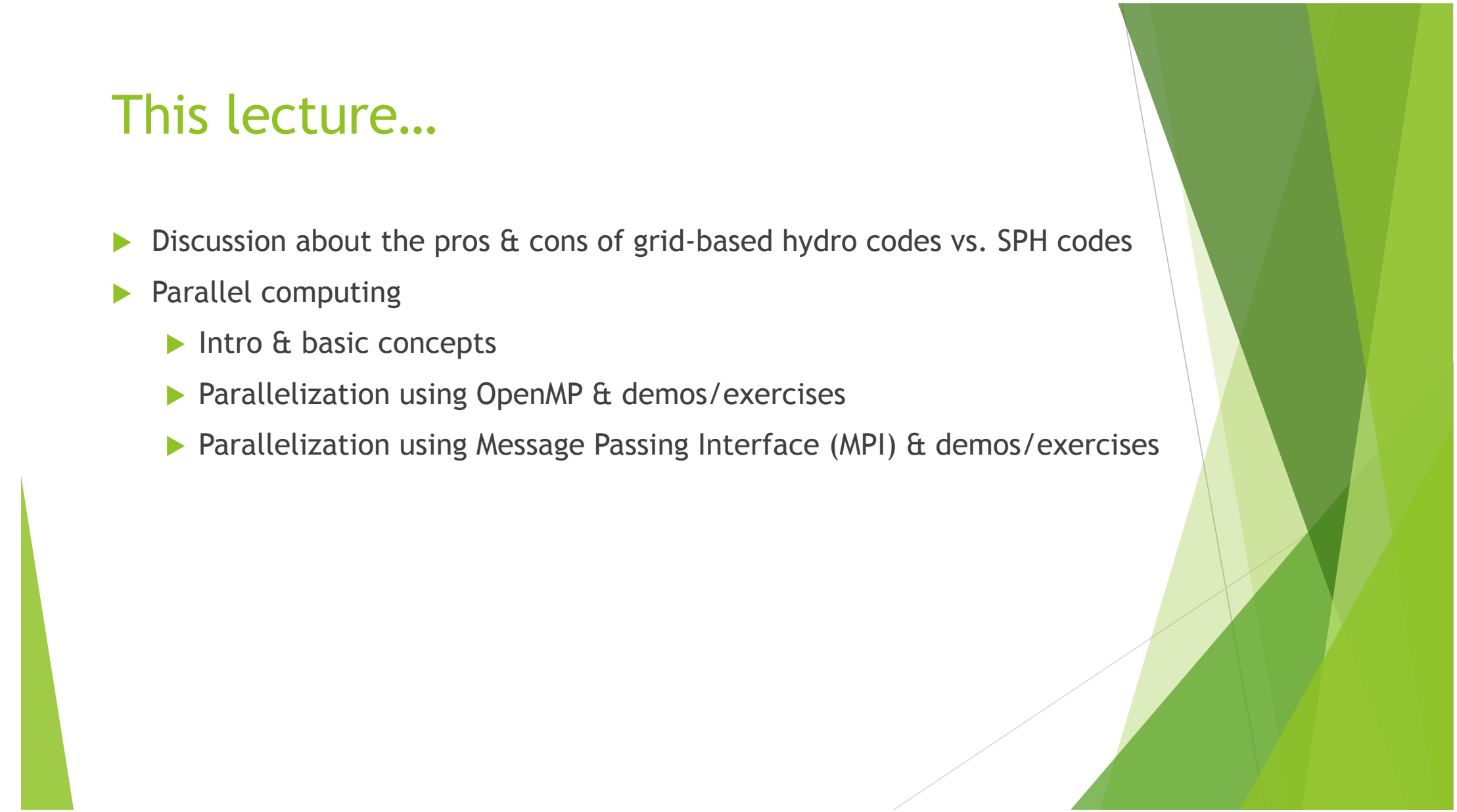
PRESENTATION SKILLS	**1**	**2**	**3**	**4**	**5**
Were the main ideas presented in an orderly and clear manner? .....,	,	,	,	,	,
Did the presentation fill the time allotted? .....,	,	,	,	,	,
Were the visual aids well prepared with clear and understandable figures/text? .....,	,	,	,	,	,
Did the presentation appropriately cite relevant references? .....,	,	,	,	,	,
Did the speaker make good eye contact and maintain the interest of the audience? ...,	,	,	,	,	,
How well did the presenter handle questions from the audience? .....,	,	,	,	,	,

Previous lecture...

- N-body or particle-based simulations are widely used in astrophysics
- Particles could represent a single object (e.g., planets, stars) or a Monte-Carlo sampling of a collection of particles
  - For the latter, need to avoid ***unphysical two-body relaxation*** by using a large N or using a softened force law
- To evolve particles, we need
  - ***Time-integration methods***: leapfrog, Runge-Kutta, symplectic integrators
  - ***Force-integration methods***: PP, PM, P^3M, tree
- ***Smoothed particle hydrodynamics (SPH):*** particle-based, Lagrangian method for hydrodynamics
  - Uses ***SPH kernels*** to compute fluid quantities
  - Uses ***symmetrized SPH equations*** to evolve the fluid quantities
  - Uses ***artificial viscosity*** for shock capturing

This lecture…

- Discussion about the pros & cons of grid-based hydro codes vs. SPH codes
- Parallel computing
  - Intro & basic concepts
  - Parallelization using OpenMP & demos/exercises
  - Parallelization using Message Passing Interface (MPI) & demos/exercises



In-class discussion

Comparisons between grid-based hydro codes and SPH codes

	Grid-based hydro codes	SPH codes
Pros		
Cons		

Comparisons between grid-based hydro codes and SPH codes

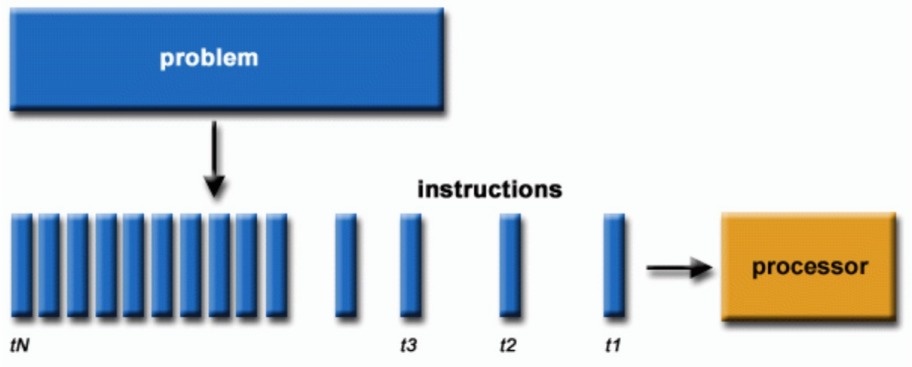
	Grid-based hydro codes	SPH codes
Pros	Good at capturing shocks/CDs	1. Automatic refinement on high density regions 2. Galilean invariant
Cons	Not Galilean invariant (due to truncation errors & numerical diffusion)	1. Difficulty dealing with contact discontinuities (Agertz+07), resulting in unphysical surface tension and suppressed hydro instabilities & turbulence 2. Drawbacks of artificial viscosity (e.g., unable to resolve shocks with high resolution, suppressed cooling, enhanced angular momentum transfer causing break-ups of galaxy disks) (Hosono+16) 3. Poor convergence rate (Hayward+13)

Parallel Computing

Serial vs. parallel computing

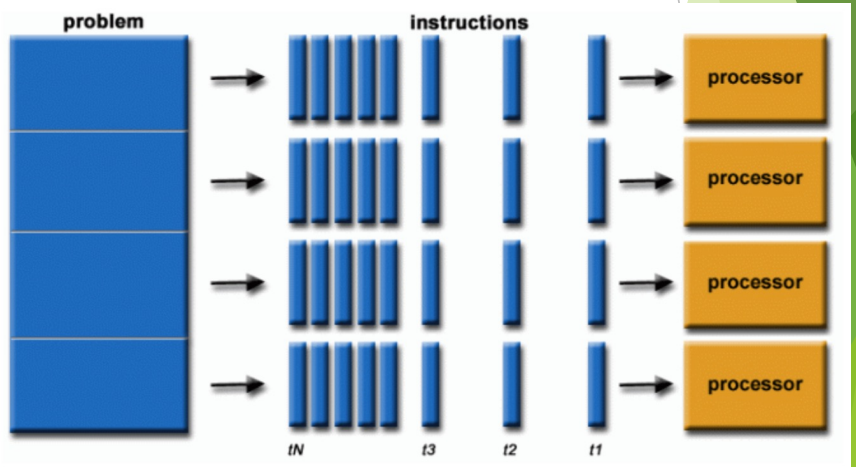
Serial computing

- Instructions in a program are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time



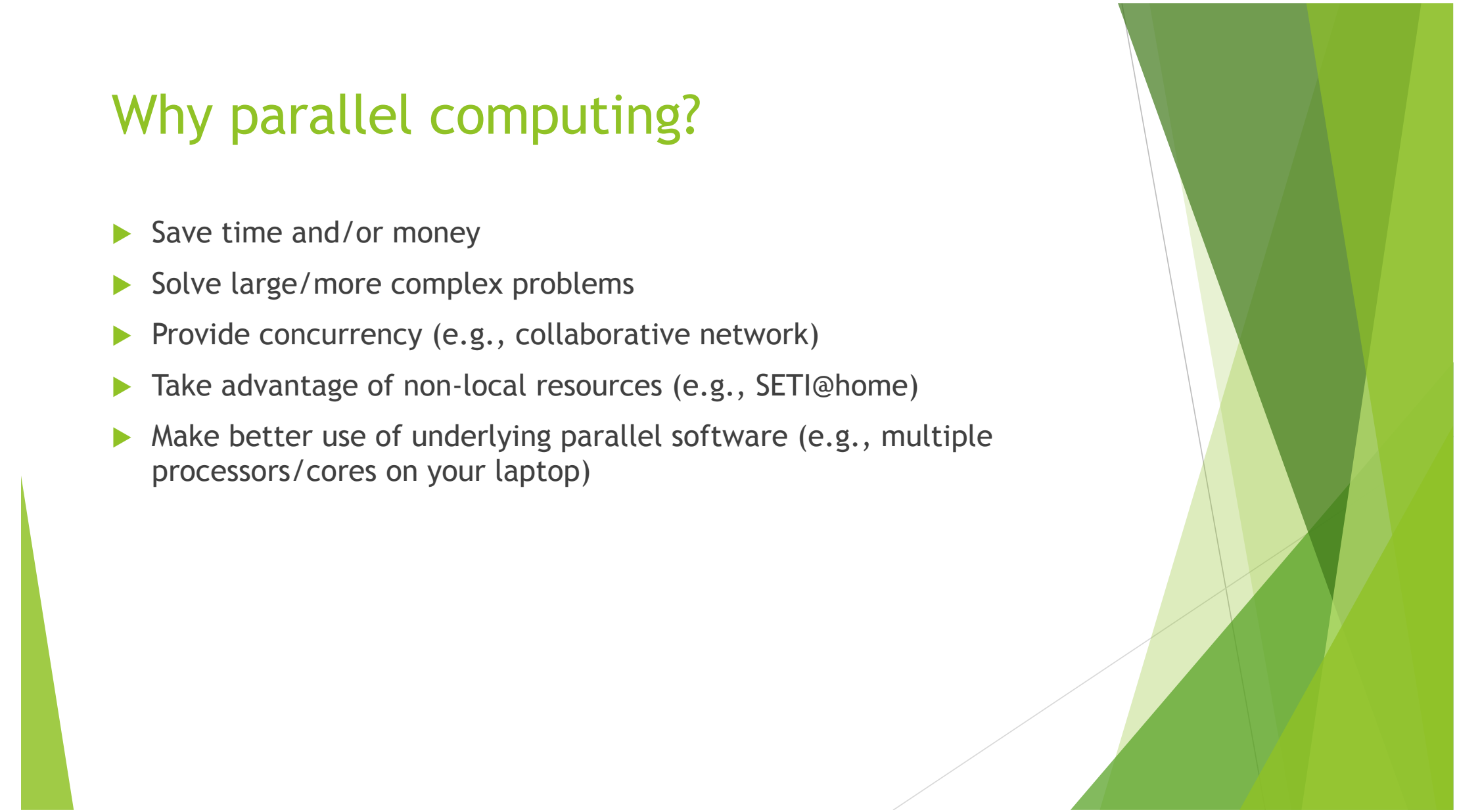
Parallel computing

- Instructions in a program are executed concurrently
- Executed simultaneously on multiple processors



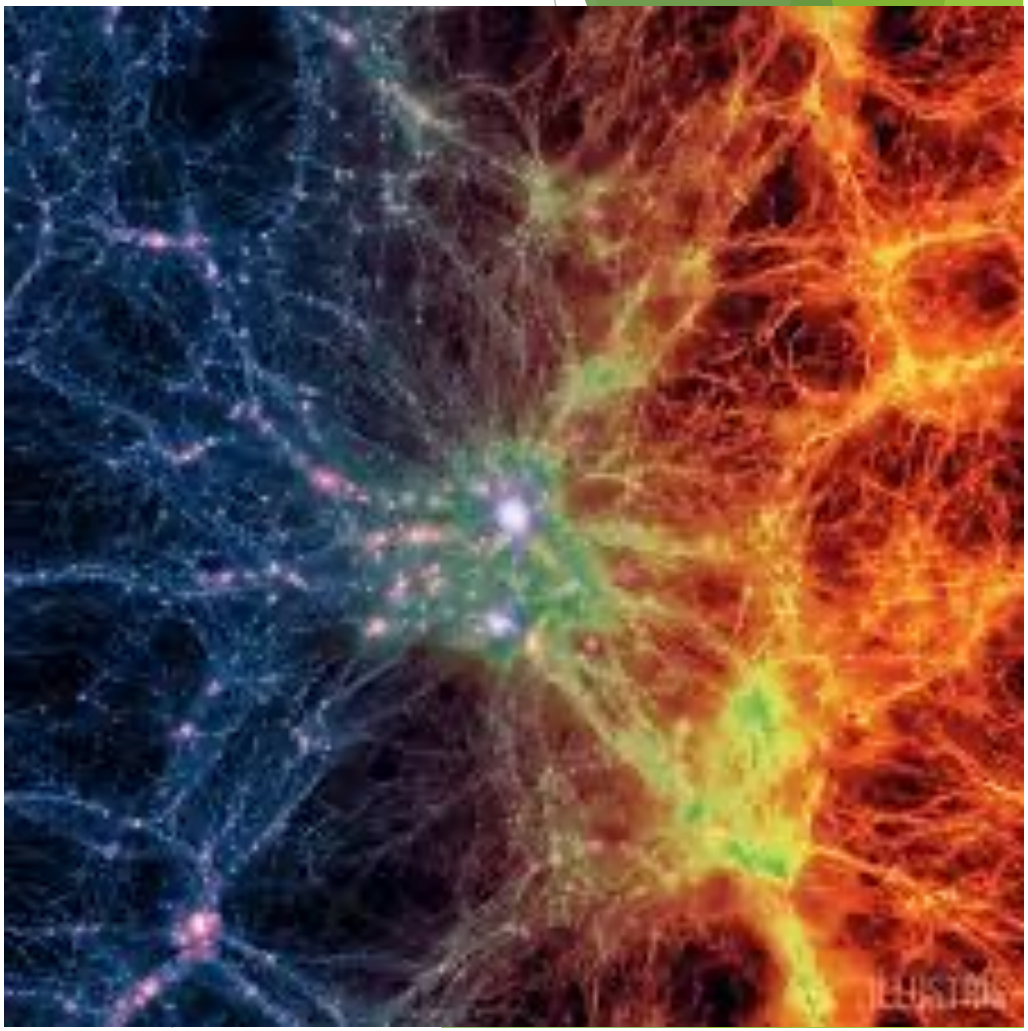
Why parallel computing?

- Save time and/or money
- Solve large/more complex problems
- Provide concurrency (e.g., collaborative network)
- Take advantage of non-local resources (e.g., SETI@home)
- Make better use of underlying parallel software (e.g., multiple processors/cores on your laptop)



Example: cosmological galaxy formation

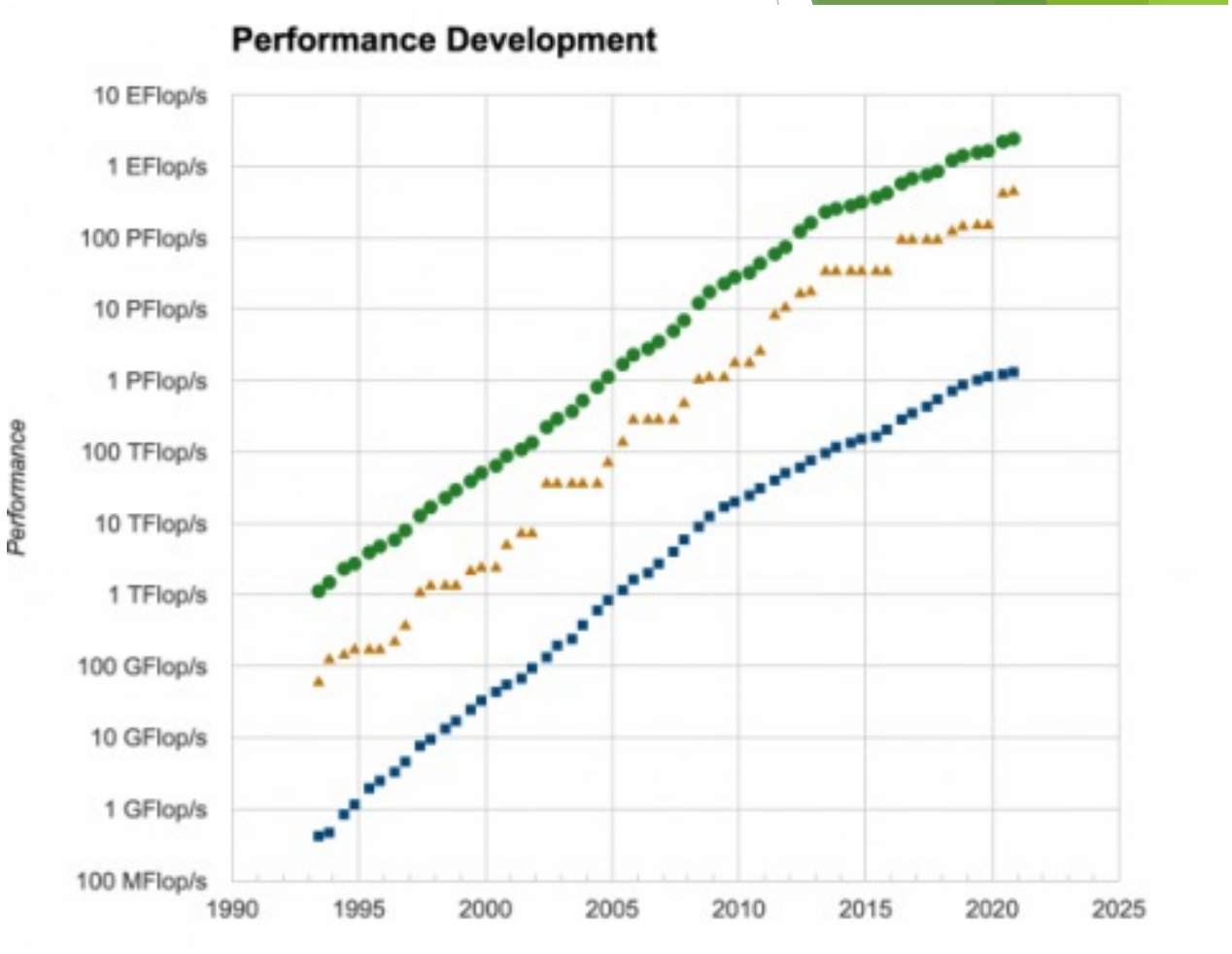
- Illustris/TNG: 3D cosmological hydro/MHD simulation of galaxy formation
- Size: 106.5 Mpc^3, time: 0.3 Myr ~ 13.8 Gyr
- Moving-mesh code AREPO
- Fluid dynamics + dark matter particles + gravity + cooling + star particles + black hole particles
- *Computing time: 1.9 x 10^7 CPU-hours*
  - *~3 months with 8192 CPUs*
  - *~2000 years on 1 CPU!*



Credit: Illustris Collective

Parallelism is the future of computing!

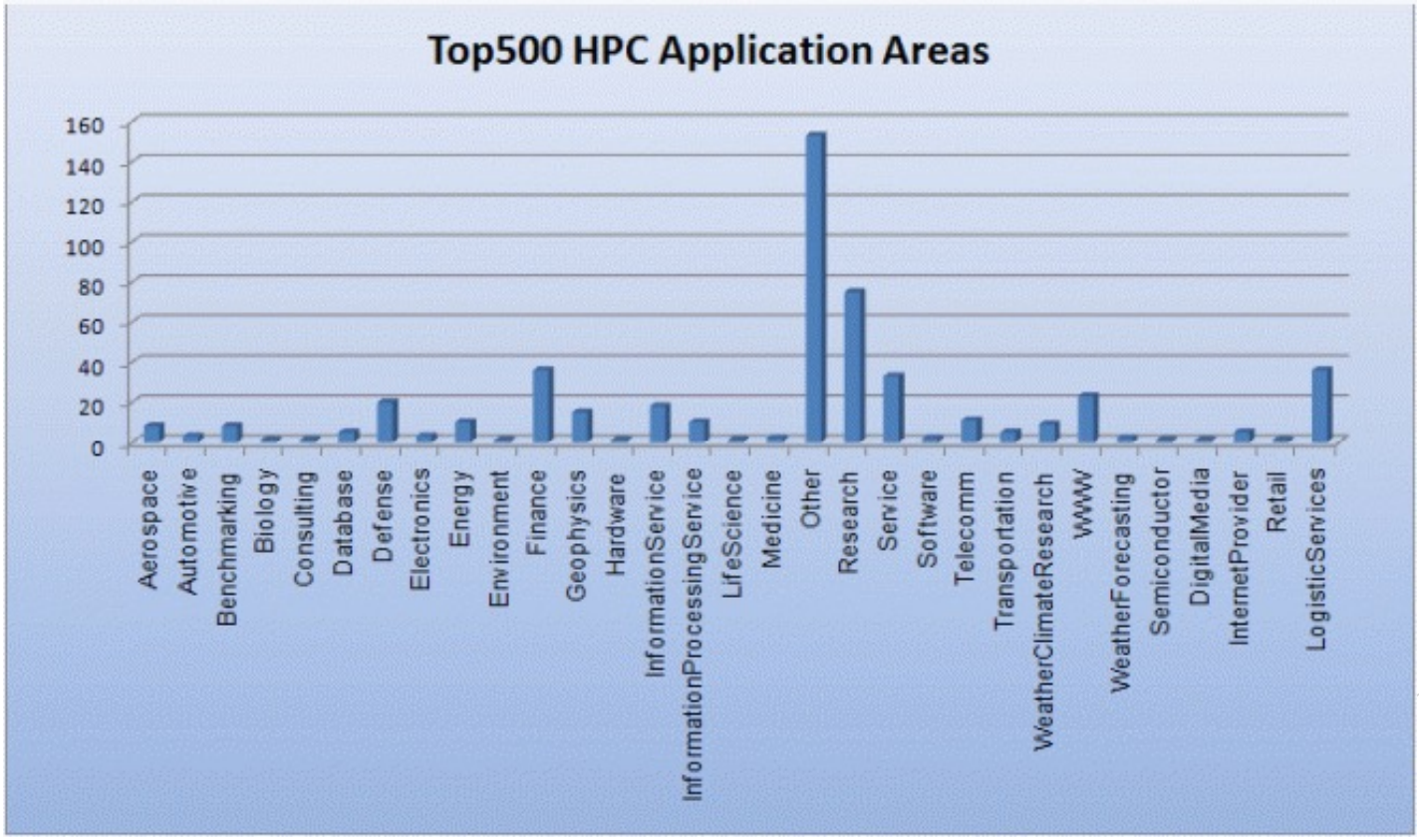
- During the past 20+ years, there has been dramatically (>500,000x) increasing computing performance due to improved architecture
- We are entering the ***exascale*** era (1 Exaflop = 10^{18} FTOPs/sec)!!



Credit: Top500.org

HPC is widely used in all fields!

High performance computing (HPC) is now being used extensively around the world, in a wide variety of applications

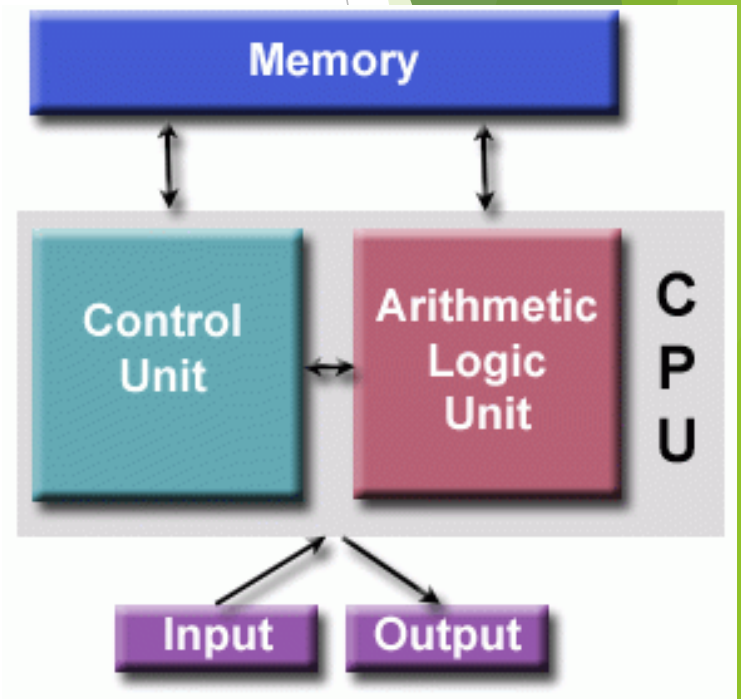


Credit: Top500.org

Concepts & architecture for computing

- Computer architecture used nowadays was first designed by John von Neumann in 1945
- Four main components: memory, control unit, arithmetic/logic units, input/output
- The code unit fetches instructions/data from memory, then sequentially executes the instructions
- Parallel computers still follow this basic design, just multiplied in units

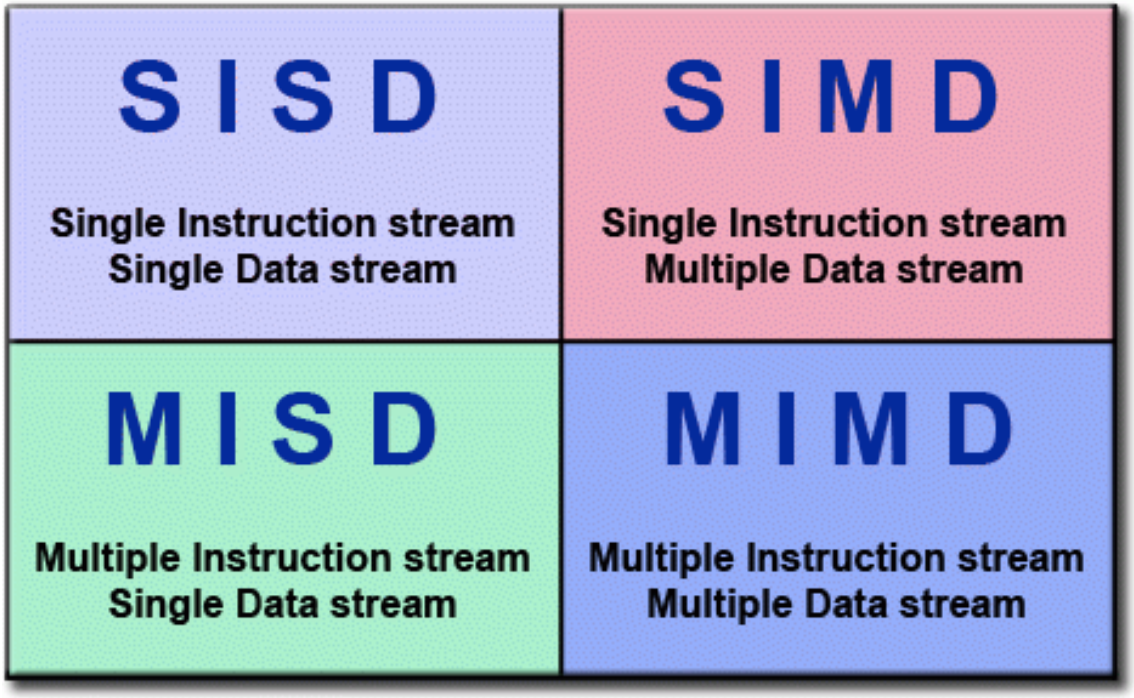
von Neumann computing architecture



Credit: https://hpc-tutorials.llnl.gov

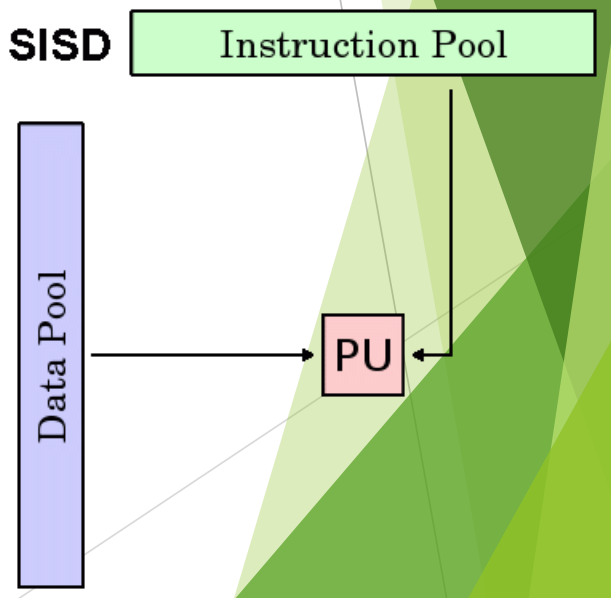
Classification of computing architectures

One of the most widely used classifications is *Flynn's Taxonomy*, in which computing architectures are classified based on whether the "*instruction stream*" and the "*data stream*" is "*single*" or "*multiple*"



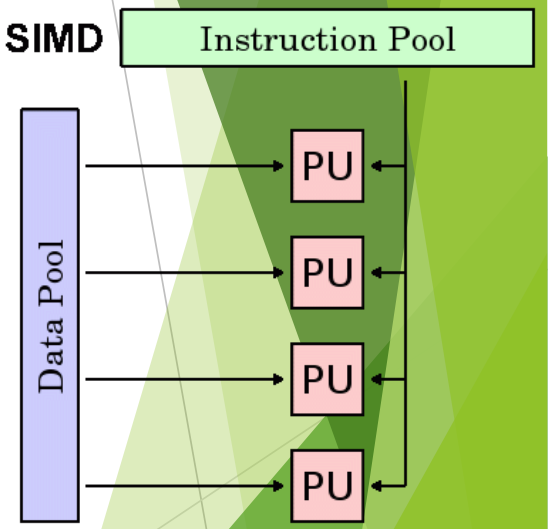
Single Instruction Single Data (SISD)

- This is a *serial* computer
- **Single instruction**: only one instruction stream is executed by the CPU
- **Single data**: only one data stream is used
- Result is *deterministic*
- **Examples**: older computers with single processor/core



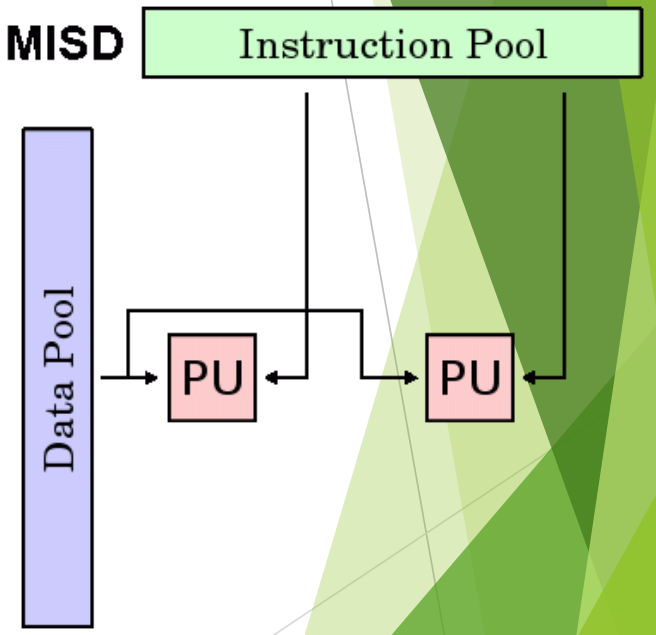
Single Instruction Multiple Data (SIMD)

- A type of *parallel* computer
- **Single instruction**: all processing units execute the same instruction
- **Multiple data**: each processing unit can operate on a different data element
- Best suited for problems with high degree of regularity (e.g., image processing)
- *Synchronous* and *deterministic* execution
- **Examples**: most modern computers, particularly those with graphics processing units (GPUs)



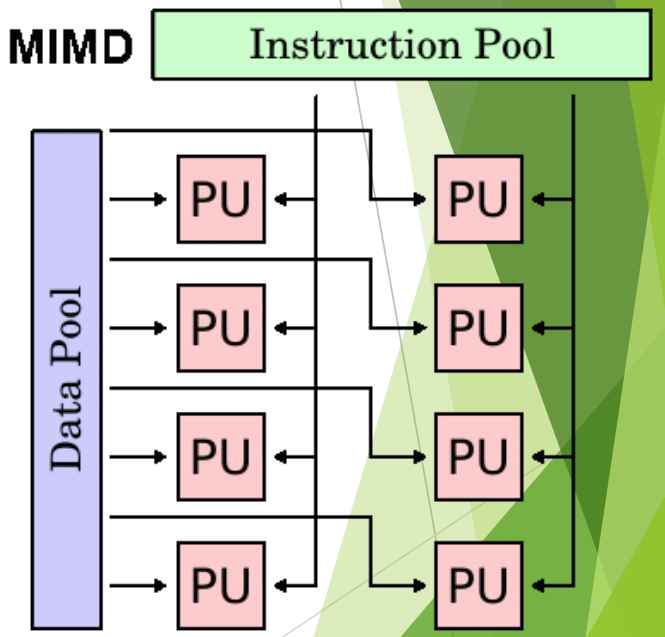
Multiple Instruction Single Data (MISD)

- A type of parallel computer
- **Multiple instruction:** each processing unit operates on data independently via separate instruction streams
- **Single data:** a single data stream is fed into multiple processing units
- Not commonly used

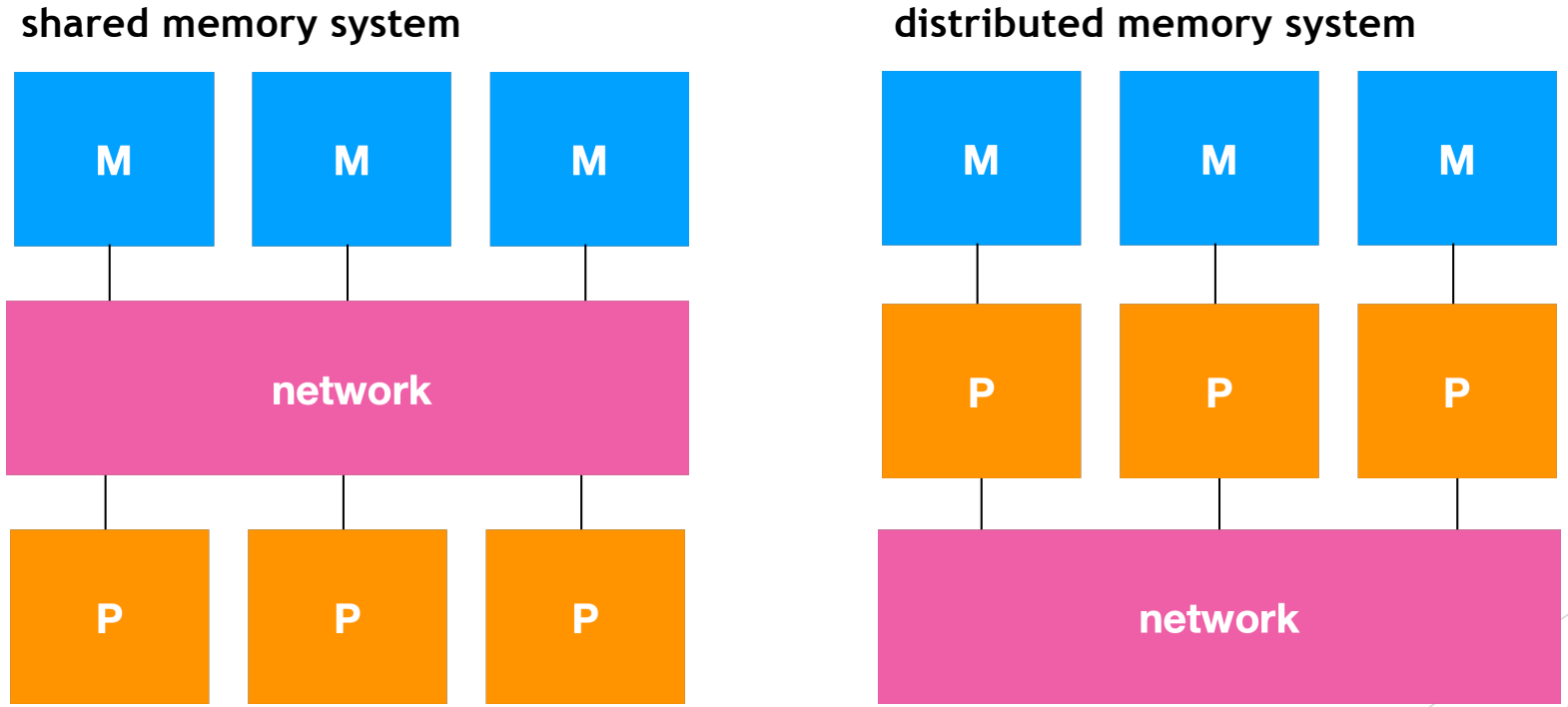


Multiple Instruction Multiple Data (MIMD)

- A type of parallel computer
- **Multiple instruction**: every processor may be executing a different instruction stream
- **Multiple data**: every processor may be working on a different data stream
- Can be synchronous or asynchronous, deterministic or non-deterministic
- **Example**: ***most current computing clusters & supercomputers***



Shared memory vs. distributed memory systems



Shared memory vs. distributed memory systems

	shared memory	distributed memory
Scalability	Harder	Easier
Data mapping	Easier	Harder
Data integrity	Harder	Easier
Performance optimization	Harder	Easier
Automatic parallelization	Easier	Harder

Parallel computing concepts & terminology

- *Speedup*: $S_N \equiv \frac{t_1}{t_N}$
- *Parallel efficiency*: $\epsilon_N \equiv \frac{S_N}{N} = \frac{t_1}{Nt_N}$
- *Parallel overhead*: extra time needed to perform parallel tasks (excluding times for useful works)
  - Examples: initialization, synchronization, data communications ...
- **Embarrassingly (ideally) parallel**: solving many similar, but independent tasks simultaneously; little to no need for coordination

Parallel computing concepts & terminology

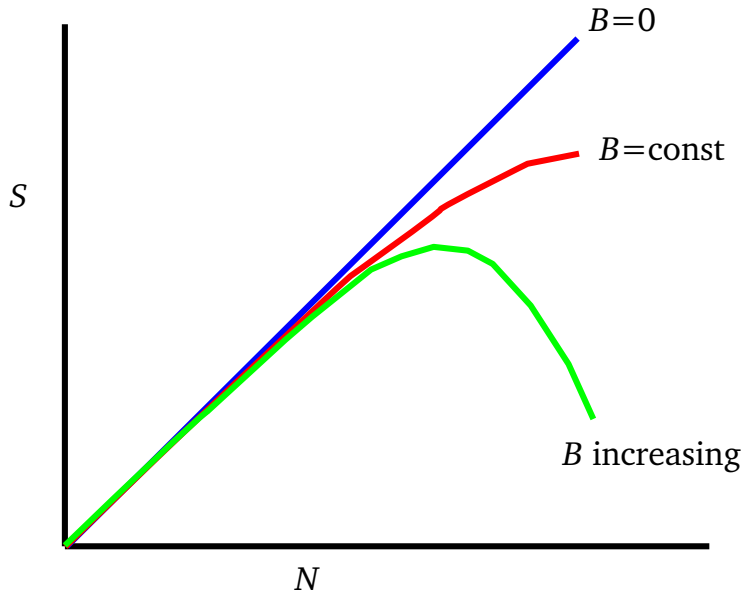
- *Dependency* – causes serialization of tasks
  - Control dependency – must complete task A before task B
  - Data dependency – If Y = f(X), then must compute X before computing Y
- *Load balancing* – keeping everyone busy
  - *Synchronization* – processors must wait for others to catch up before moving on
  - *Barrier* – explicit processor synchronization (e.g., `call MPI_BARRIER(MPI_COMM_WORLD, ierr)`)
  - *Processor starvation* – when a processor does not have enough work to do

Scalability

- Refers to whether a parallel system (hardware and/or software) is able to increase the speedup when adding more resources
- Factors:
  - Hardware (e.g., memory-cpu bandwidths, network communications)
  - Parallel overhead
  - Algorithm (Amdahl's law)
- Amdahl's law:

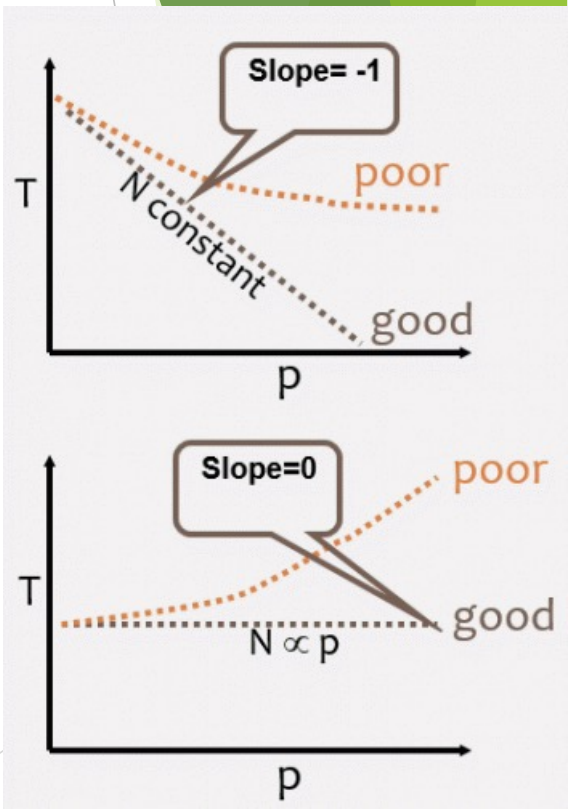
$$S = \frac{N}{BN + (1 - B)}$$

B = % of algorithm that is serial for N processors



Scalability

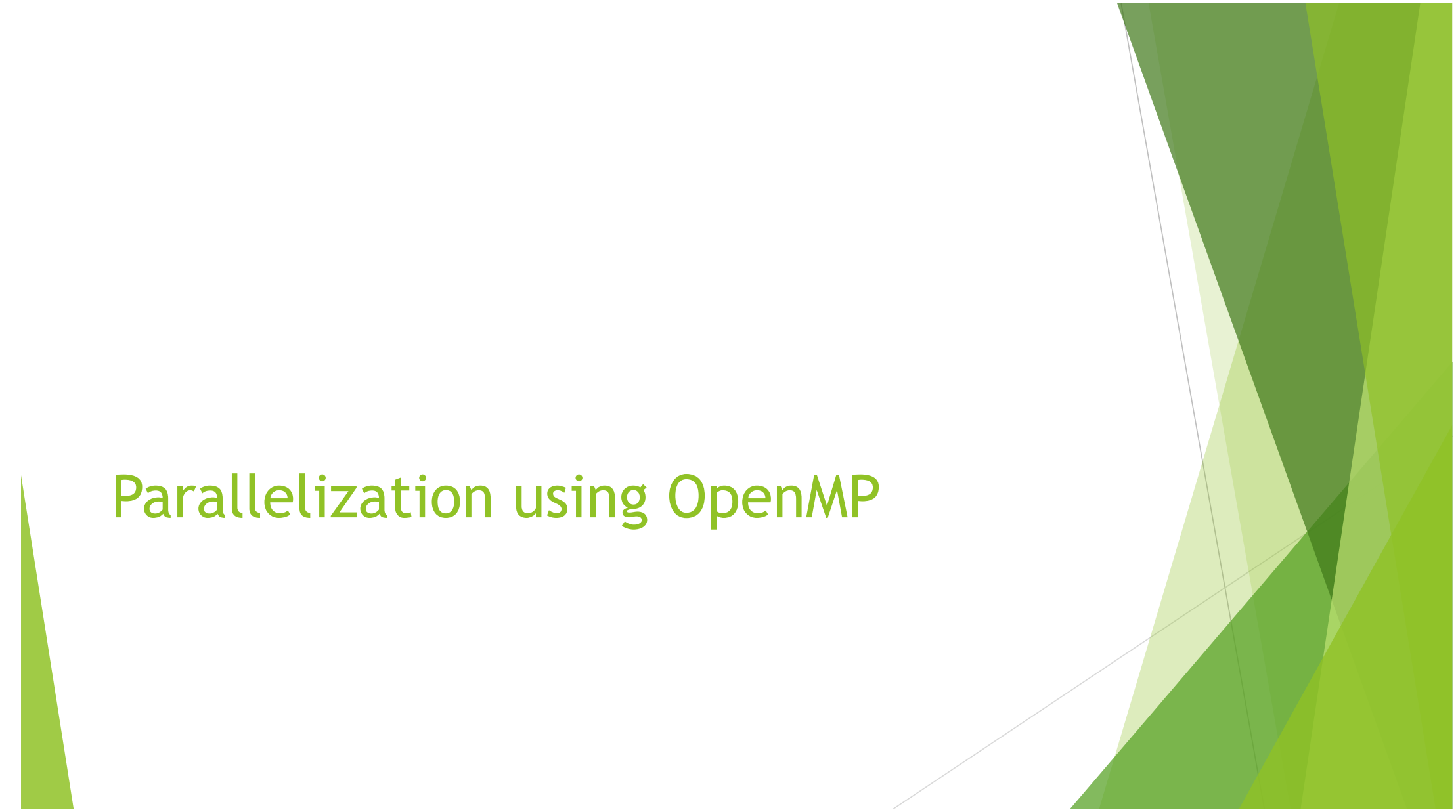
- ***Strong scaling***
  - Given ***fixed problem size***, how does the computing time (T) scale with more processors (P)
  - Perfect scaling means problem is solved in 1/P time
  - This measures how much faster we can run the same problem
- ***Weak scaling***
  - Given ***fixed problem size per processor*** (total size proportional to P), how does the computing time scale with P
  - Perfect scaling means problem runs in same time as single processor run
  - This measures how large a problem we can run in same amount of time



Parallel programming implementations

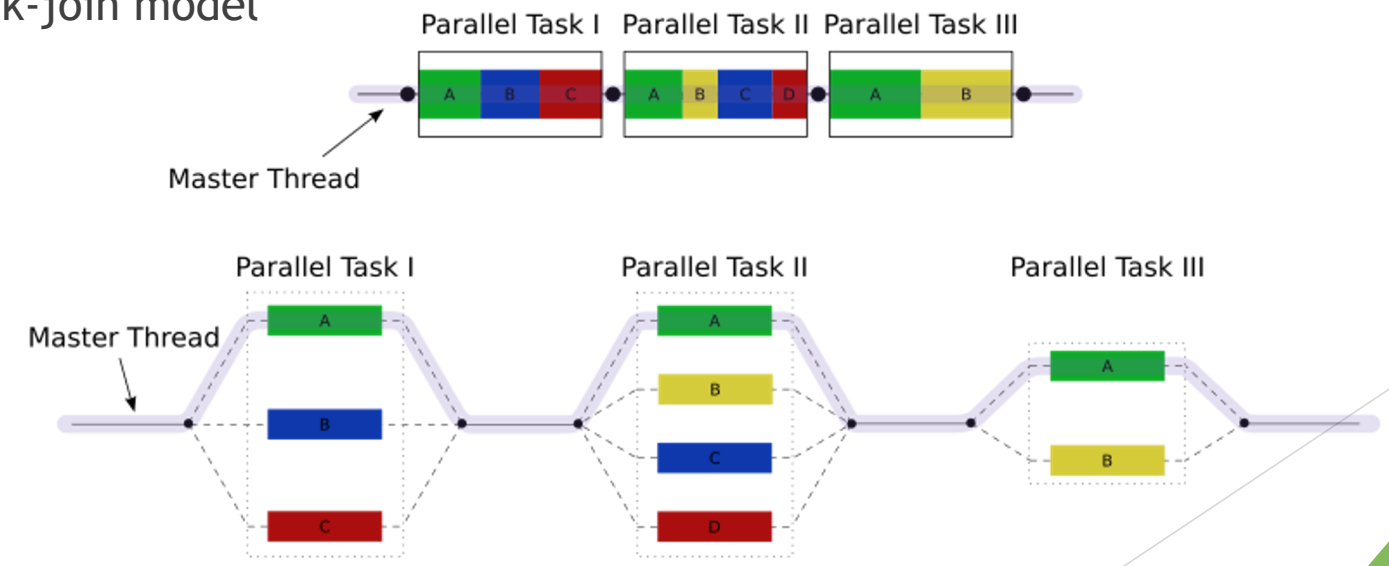
- Automatic parallelization (by compiler)
- Threads on shared-memory systems (e.g., OpenMP)
- Message passing on distributed-memory systems (e.g., MPI)
- Hybrid models

Parallelization using OpenMP



OpenMP – Open Multi-Processing

- Multi-threaded, shared-memory, directive-based parallelism
  - "Directive" or "pragma" tells the compiler how to process the program (e.g., #include, #ifdef)
- Intra-node parallelism
- Fork-join model



Credit: wikipedia

OpenMP – Open Multi-Processing

- Supported languages: C, C++, Fortran, Python (using Cython)
- Three main API (application program interface)
  - Compiler directives (e.g., `!$omp parallel` in Fortran)
  - Runtime library routines (e.g., `omp_get_num_threads()`)
  - Environment variables (e.g., `OMP_NUM_THREADS`)
- This lecture only overs a very brief introduction – see Supplemental Information for details

The “parallel” construct

- This is the fundamental OpenMP construct defining a code black (a *parallel region*) to be executed by *multiple threads*
  - A team of threads is created when entering a parallel region
  - The original thread becomes the master thread (with *thread ID* 0)
  - All threads (including master) execute the same code block
  - An implied barrier is put at the end of a parallel region
- **Example:** Hello World!

```
program hello

    implicit none

    !$omp parallel
    print *, " hello world"
    !$omp end parallel

end program
```

Demo: Try running Hello World using OpenMP

- Connect to CICA and load the Fortran compiler:

```
module load gcc
```

- Go to your exercise directory on CICA:

```
cd astr660/exercise
```

- Copy the template files to the current directory:

```
cp -r /data/hyang/shared/astro660CompAstro/L15exercise ./
```

- Enter the openmp directory and see what files are there:

```
cd L15exercise/openmp
ls
```

Demo: Try running Hello World using OpenMP

- Compile the hello.f90 program:

```
gfortran –o hello –fopenmp hello.f90
```

- Execute the program and see the result:

```
./hello
```

- Vary the environment variable $OMP_NUM_THREADS, e.g.,

```
export OMP_NUM_THREADS=4
```

- Run the program again and see how the result changes

Thread ID

- omp_get_thread_num(): get the thread ID
- omp_get_num_threads(): get the current number of threads
- **Demo:** thread_id.f90 - did you get the desired result?

```
program hello

    use OMP_LIB

    implicit none
    integer :: nthreads, tid

!$omp parallel
    tid = omp_get_thread_num()
    print*, 'Hello World from thread = ', tid

    if (tid .eq. 0) then
      nthreads = omp_get_num_threads()
      print*, 'Number of threads = ', nthreads
    endif
!$omp end parallel

end program hello
```

*These variables are **shared** among different threads!!*
⇒ *This would result in the "**race condition**" in which the processors race to write to memory!!*
⇒ *To fix this, one way is to use:*
!$omp parallel private(nthreads, tid)

Data-sharing attribute clauses

- Variables in a parallel region can be either ***shared*** or ***private***
  - Shared variables: visible/modifiable to all threads
  - Private variables: each thread has its own copy
- ***Data-sharing attribute clauses*** are used to control the visibility of selected variables
  - E.g., `!$omp parallel private (var)`
- Most variables are shared by default
- Loop iteration variables are private by default

Data-sharing attribute – “private” clause

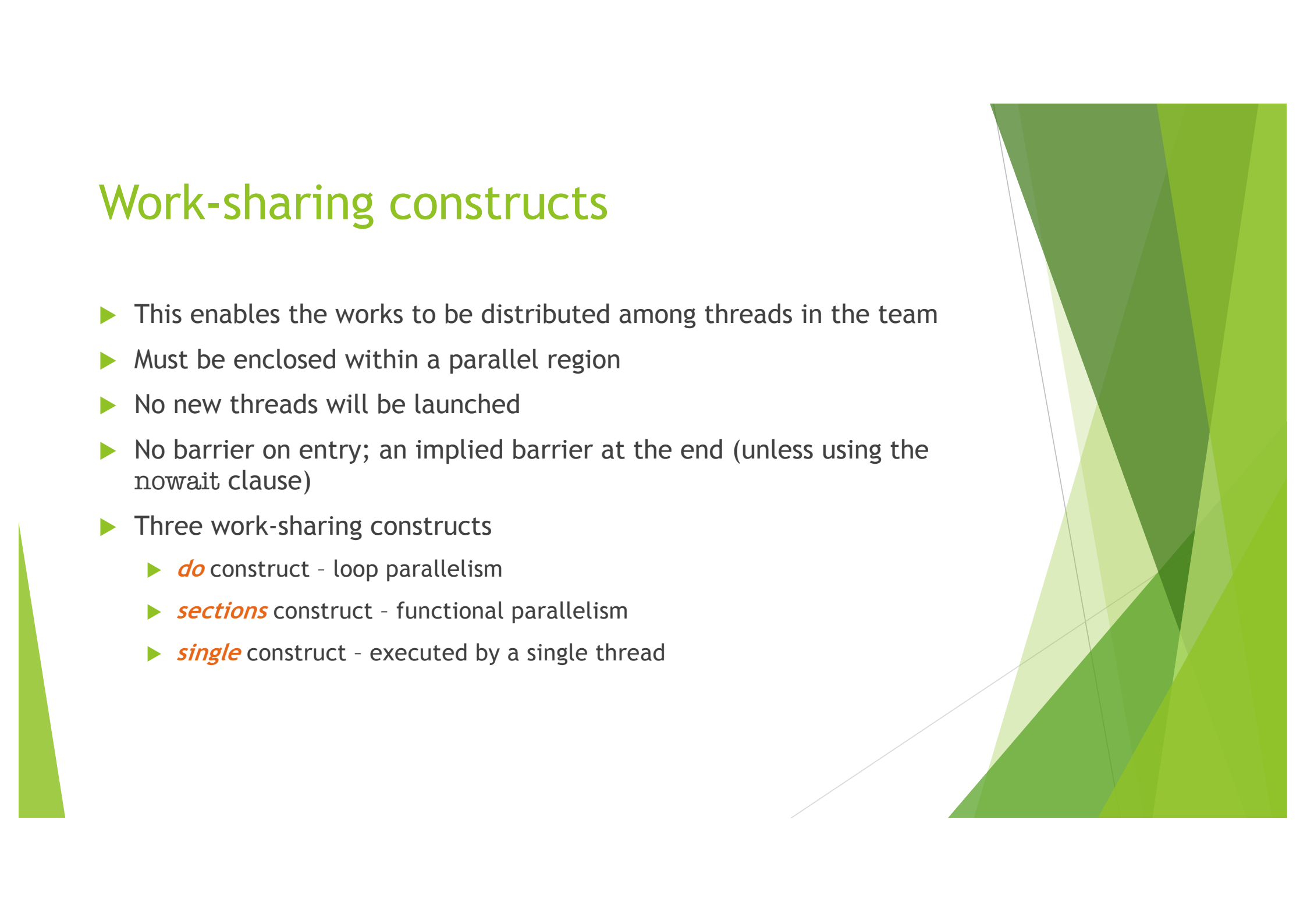
- The private(var) clause creates a new local copy of var for each thread
- Demo: private.f90 – what is the result of the tmp variable?

```
program private

    use OMP_LIB

    implicit none
    integer, parameter :: n = 10
    integer :: i, tmp

    tmp = 0

!$omp parallel do private(tmp)
    do i = 1, n
        tmp = tmp + 1
    enddo
!$omp end parallel do

    print*, 'tmp = ', tmp

end program private
```

tmp is not initialized!
⇒ *To initialize from a shared variable, use the “**firstprivate**” clause*
⇒ *Use the **default(none)** clause so that the compiler would complain if variable extents are not defined*

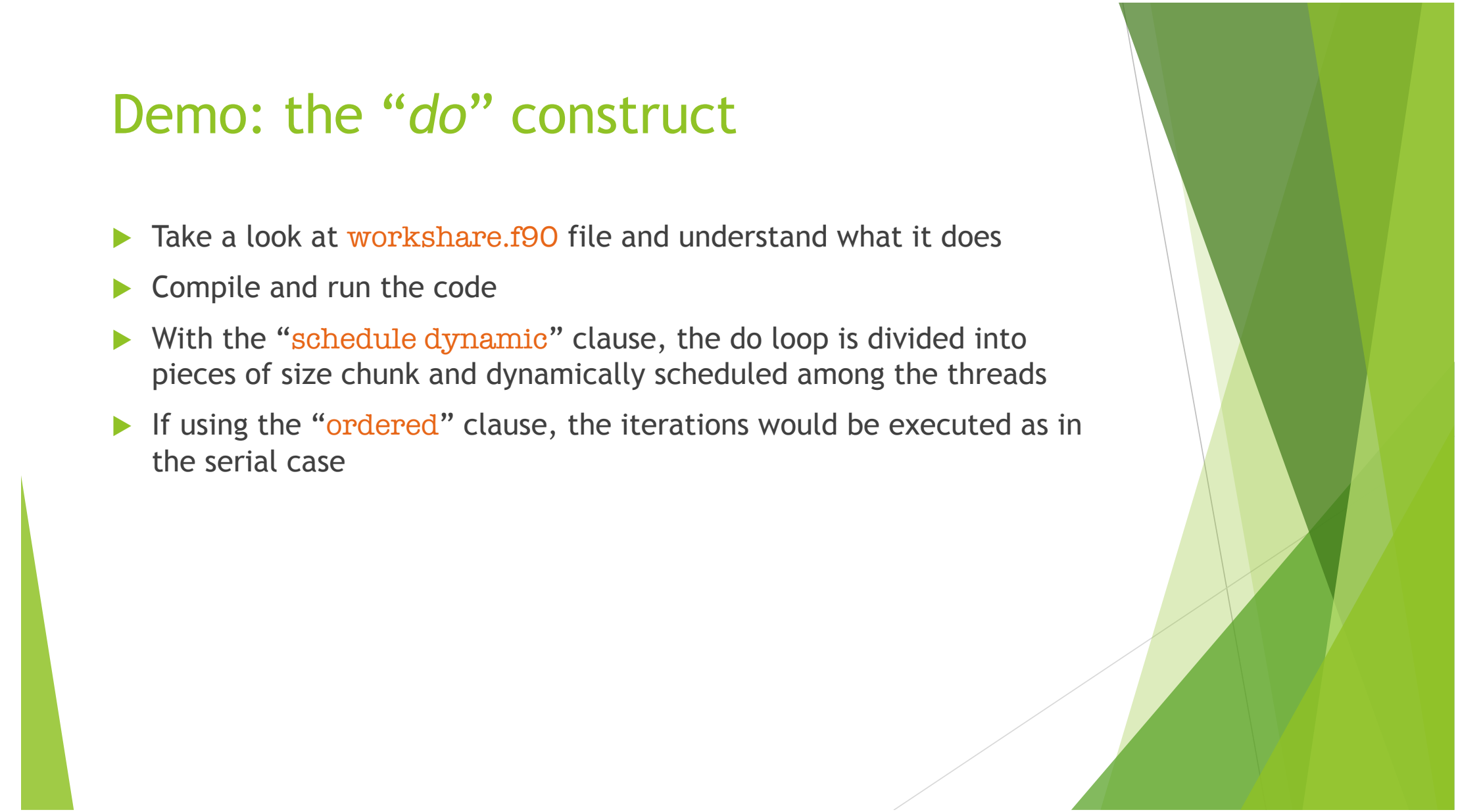
Work-sharing constructs

- This enables the works to be distributed among threads in the team
- Must be enclosed within a parallel region
- No new threads will be launched
- No barrier on entry; an implied barrier at the end (unless using the `nowait` clause)
- Three work-sharing constructs
  - *do* construct – loop parallelism
  - *sections* construct – functional parallelism
  - *single* construct – executed by a single thread



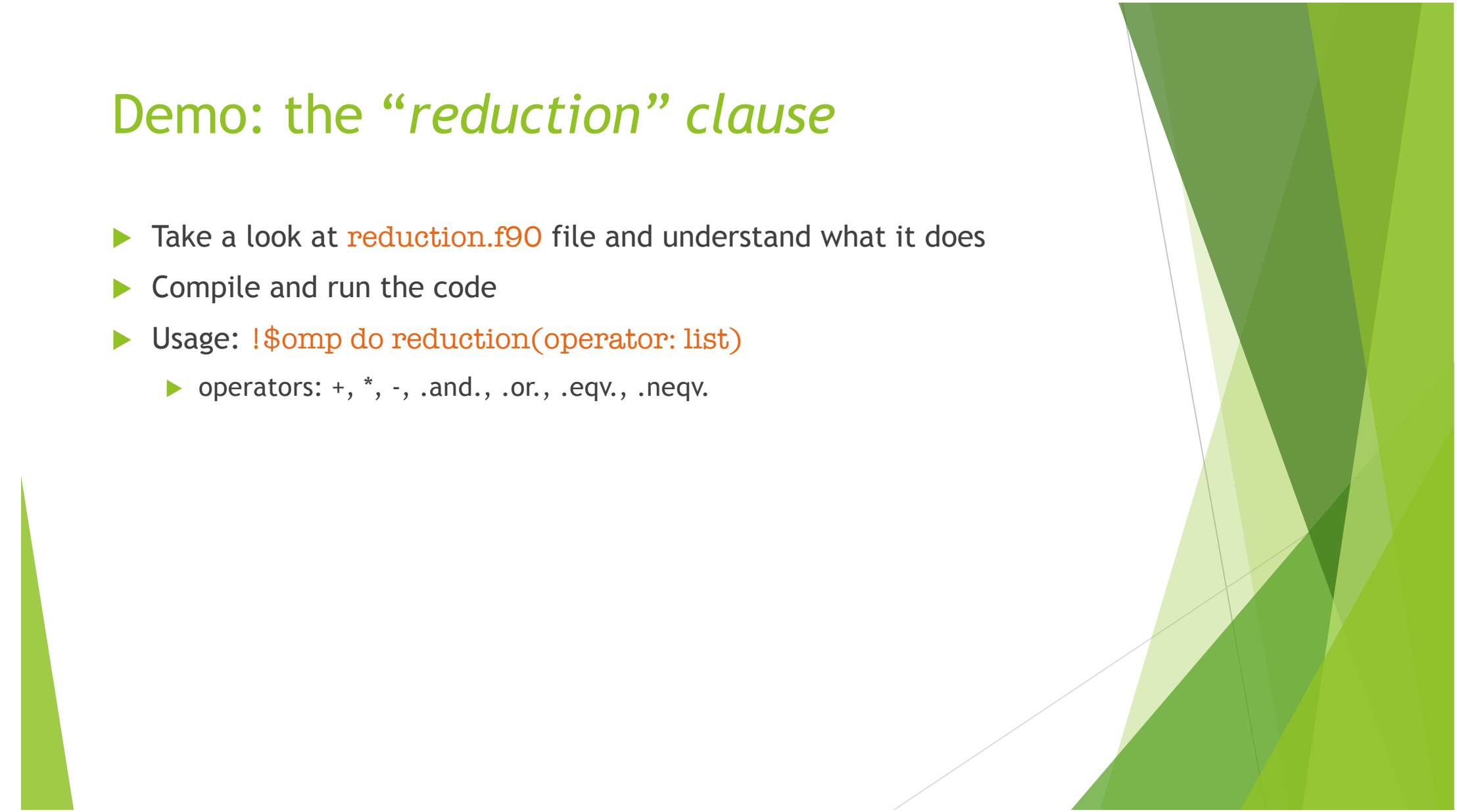
Demo: the "*do*" construct

- Take a look at `workshare.f90` file and understand what it does
- Compile and run the code
- With the "`schedule dynamic`" clause, the do loop is divided into pieces of size chunk and dynamically scheduled among the threads
- If using the "`ordered`" clause, the iterations would be executed as in the serial case



Demo: the "*reduction*" *clause*

- Take a look at reduction.f90 file and understand what it does
- Compile and run the code
- Usage: !$omp do reduction(operator: list)
  - operators: +, *, -, .and., .or., .eqv., .neqv.



The "*sections*" construct

- The "***sections***" construct assigns different blocks of work to different threads
- Each section is computed by one thread; each thread may compute more than one section
- Threads <-> sections mapping is indeterministic
- Execution order of different section is indeterministic

```
program sections

    use OMP_LIB

    implicit none
    integer :: i
    integer, parameter :: n = 100
    real :: a(n), b(n), c(n), d(n)

    ! initialization
    do i = 1, n
        a(i) = i * 1.0
        b(i) = a(i)
    enddo

!$omp parallel shared(a,b,c,d) private(i)

!$omp sections

!$omp section
    do i = 1, n
        c(i) = a(i) + b(i)
    enddo

!$omp section
    do i = 1, n
        d(i) = a(i) + b(i)
    enddo

!$omp end sections nowait

!$omp end parallel

end program sections
```

The "*single*" construct

- The "***single***" construct specifies a code block to be executed ***by a single thread***
  - Which thread to execute this code block is unspecified
  - It has an implied barrier at the end of this construct
- It is similar to the "***master***" construct, which specifies a code block to be executed only by the ***master*** thread
  - No implied barrier at the end
- Often used when dealing with code that are not thread safe (e.g., file I/O)

```
program single

    use OMP_LIB

    implicit none
    integer :: nt, tid, i

!$omp parallel

    nt  = omp_get_num_threads()
    tid = omp_get_thread_num()
    if (tid .eq. 0) then
        print*, 'Number of threads = ', nt
    endif
    print*, 'Outside the single construct from thread ', tid

!$omp single
    print*, 'Inside the single construct from thread ', tid
!$omp end single nowait

!$omp end parallel

end program single
```

Synchronization constructs – *"critical"*

- Specify a code block to be executed by ***one thread at a time***
  - All threads will attempt to execute the CRITICAL region, but a thread must wait until the previous thread exists the CRITICAL region
- **Demo**: in critical.f90, fix the data race problem using the "critical" construct

wrong due to data race

```
program critical

    use OMP_LIB

    implicit none
    integer, parameter :: n = 100
    integer :: i, tmp

    tmp = 0

!$omp parallel do
    do i = 1, n

        tmp = tmp + 1

    enddo
!$omp end parallel do

    print*, 'tmp = ', tmp

end program critical
```

Synchronization constructs – "*critical*"

- Specify a code block to be executed by ***one thread at a time***
  - All threads will attempt to execute the CRITICAL region, but a thread must wait until the previous thread exists the CRITICAL region
- **Demo**: in critical.f90, fix the data race problem using the "critical" construct

correct

```
program critical

    use OMP_LIB

    implicit none
    integer, parameter :: n = 100
    integer :: i, tmp

    tmp = 0

!$omp parallel do
    do i = 1, n
!$omp critical
        tmp = tmp + 1
!$omp end critical
    enddo
!$omp end parallel do

    print*, 'tmp = ', tmp

end program critical
```

Synchronization constructs – "*barrier*"

- Synchronize all threads in a parallel region
  - Threads reaching this barrier earlier will wait until all other threads have reached the same barrier
- **Demo**: in barrier.f90, fix the data race problem using the "barrier" construct

wrong due to data race

```
program barrier

    use OMP_LIB

    implicit none
    integer, parameter :: n = 10000
    integer :: i, tmp, result

    tmp = 0

!$omp parallel
    do i = 1, n
!$omp critical
        tmp = tmp + 1
!$omp end critical
    enddo

!$omp master
    result = tmp
    print*, 'result = ', result
!$omp end master

!$omp end parallel

end program barrier
```

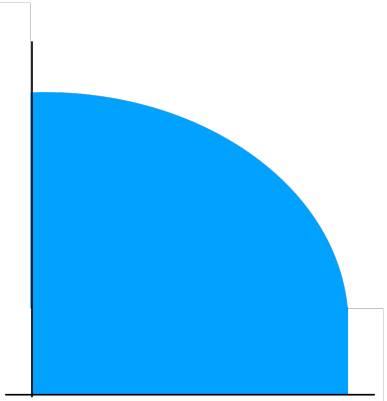
Synchronization constructs – "*barrier*"

- Synchronize all threads in a parallel region
  - Threads reaching this barrier earlier will wait until all other threads have reached the same barrier
- **Demo**: in barrier.f90, fix the data race problem using the "barrier" construct

correct

```
program barrier

    use OMP_LIB

    implicit none
    integer, parameter :: n = 10000
    integer :: i, tmp, result

    tmp = 0

!$omp parallel
    do i = 1, n
!$omp critical
        tmp = tmp + 1
!$omp end critical
    enddo
!$omp barrier
!$omp master
    result = tmp
    print*, 'result = ', result
!$omp end master

!$omp end parallel

end program barrier
```

Exercise: parallelize the pi calculation using OpenMP



$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

- To get the bonus credit, please submit the code and screenshot to the TAs by the start of next class (12/29/2022)

serial version

```
1  program pi
2
3      use OMP_LIB
4
5      implicit none
6      integer :: i
7      real    :: x, dx , sum, t1, t2
8      integer,parameter  :: n = 10000000000
9
10     dx = 1./float(n)
11
12     t1 = omp_get_wtime()
13     do i=0,n-1
14         x = (i+0.5)*dx
15         sum = sum + 4.0/(1.0+x*x)
16     enddo
17     t2 = omp_get_wtime()
18     print *, "Pi   =", dx*sum
19     print *, "Time =", (t2-t1)
20
21 end program pi
```

Parallelization using MPI

Message Passing Interface (MPI)

- Parallelization by ***passing messages*** among different processors
- Works on shared-memory, distributed-memory, or hybrid systems
  - Transparent to users – users don't need to distinguish inter-node and intra-node communication
- Single program multiple data (SPMD) model
  - All processes execute the same program
  - Each process gets a unique ID called "***rank***" (similar to thread ID in OpenMP)
  - But unlike OpenMP, users need to ***call the MPI routines*** to explicitly transfer data among different ranks
- Two types of communication
  - ***Point-to-point communication***
  - ***Collective communication***
- Can be used together with OpenMP – hybrid parallelism

Message Passing Interface (MPI)

- Supported languages: C, C++, Fortran
  - Other languages are also supported as long as they can interface with these C/C++/Fortran libraries (e.g., mpi4py)
- Many implementations (e.g., OpenMPI, MPICH, MVAPICH, Intel MPI)
  - MPICH: http://www.mpich.org/
  - OpenMPI: http://www.open-mpi.org
  - Both websites have tutorials available
- Compilation for Fortran codes
  - include 'mpif.h' or use mpi
  - Compile using mpif90 (for Fortran codes) or mpicc (for C) – implementation dependent
- Running the program (implementation dependent)
  - E.g., mpirun –np 48 a.out
  - E.g., mpiexec –np 48 a.out
  - E.g., srun –n 48 a.out

Minimal MPI

- `MPI_Init()`: initiates use of MPI
- `MPI_Finalize()`: finalize use of MPI
- `MPI_Comm_size(MPI_Comm comm, int size)`: return the number of processes in communicator comm
- `MPI_Comm_rank(MPI_Comm comm, int rank)`: return the rank of calling process in communicator comm

- Initially, all processes belongs to the communicator ***MPI_COMM_WORLD***
  - One could assign the processes to different groups (e.g., see http://mpitutorial.com/tutorials/introduction-to-groups-and-communicators)

Demo: MPI Hello World

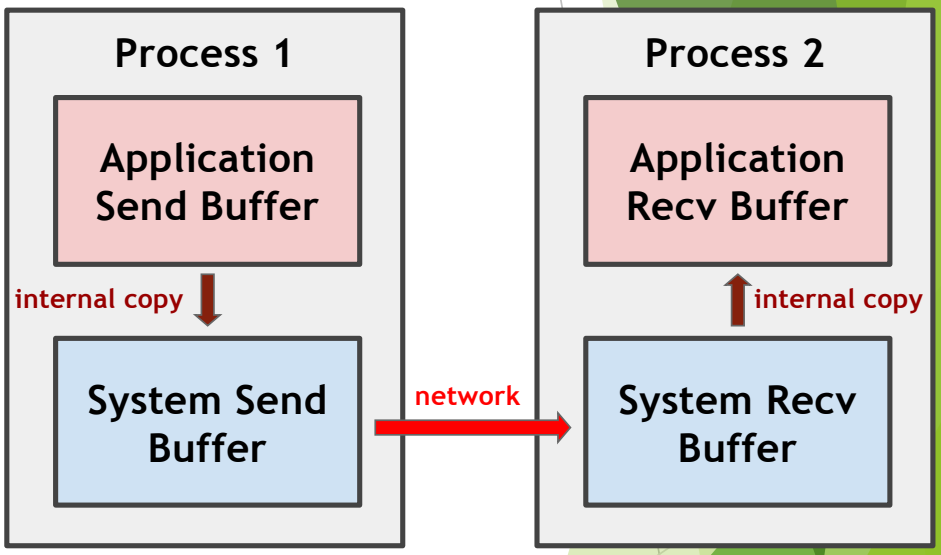
- module load gcc
- module load openmpi
- mpif90 –o hello hello.f90
- mpirun –np 2 ./hello

[your exercise directory]/L15exercise/mpi/hello.f90

```
1  program hello
2      implicit none
3      include 'mpif.h'
4      integer :: rank, size ,ierror
5
6      call MPI_INIT(ierror)
7      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
8      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
9      print *, "hello world from rank", rank, '/', size
10     call MPI_FINALIZE(ierror)
11
12 end program hello
13
```

Point-to-point communication

- To transfer data from process 1 to process 2
  - Process 1 performs a "*send*" operation
  - Process 2 performs a "*receive*" operation (denoted as "*recv*")
- Done by calling
  - MPI_Send(void msg, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - MPI_Recv(void msg, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status status)



Demo: MPI send/recv

- Take a look and understand how it works
- Compile and run the code:
  - mpif90 -o talk talk.f90
  - mpirun -np 2 ./hello
- Try running on different number of processes and see how the result changes

talk.f90

```
1  program hello
2      implicit none
3      include 'mpif.h'
4      integer :: rank, size ,ierror
5      integer :: status(MPI_STATUS_SIZE)
6      integer :: tag, i
7      integer :: msg, msg_sum
8
9      call MPI_INIT(ierror)
10     call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
11     call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
12
13     msg_sum  = 0
14     tag      = 1
15     msg      = rank
16
17     if (rank .eq. 0) then
18         msg_sum = msg
19         do i = 1, (size-1)
20             call MPI_Recv(msg, 1, MPI_INTEGER, i, tag, MPI_COMM_WORLD, status, ierror)
21             msg_sum = msg_sum + msg
22         enddo
23     else
24         call MPI_Send(msg, 1, MPI_INTEGER, 0, tag, MPI_COMM_WORLD, ierror)
25     endif
26
27     if (rank .eq. 0) print *, "sum = ", msg_sum
28     call MPI_FINALIZE(ierror)
29
30 end program hello
31
```

Blocking vs. non-blocking communications

- Standard send/recv functions are "***blocking***"
  - ***MPI_Send (blocking send):*** return only after the send buffer can be safely reused
  - ***MPI_Recv (blocking recv):*** return only after the recv buffer contains the complete received message
- On the other hand, ***non-blocking*** send/recv functions ***return immediately*** after the function call
  - ***MPI_Isend (non-blocking send)*** and ***MPI_Irecv (non-blocking recv)***
  - Communications proceed in the background
  - Unsafe to access the send/recv buffers after calling the non-blocking functions => require calling `MPI_Wait` or `MPI_Test` to test the completion of communication
  - Advantages: (1) allow overlapping communication (which may improve performance), (2) avoid deadlock

Example: blocking communication

blocking.f90

```
tag      = 1
sendbuf    = (rank+1)*10
targetRank = mod(rank+1,2)

! rank 0: send first and then receive using blocking transfer
if (rank .eq. 0) then
    call MPI_Send(sendbuf, 1, MPI_INTEGER, targetRank, tag, &
                  MPI_COMM_WORLD, ierror)
    call MPI_Recv(recvbuf, 1, MPI_INTEGER, targetRank, tag, &
                  MPI_COMM_WORLD, status, ierror)
! rank 1: receive first and then send using blocking transfer
else
    call MPI_Recv(recvbuf, 1, MPI_INTEGER, targetRank, tag, &
                  MPI_COMM_WORLD, status, ierror)
    call MPI_Send(sendbuf, 1, MPI_INTEGER, targetRank, tag, &
                  MPI_COMM_WORLD, ierror)
endif

print*, "Rank ", rank, ": Send ", sendbuf, ", Recv ", recvbuf
```

Example: deadlock (case 1)

```
tag      = 1
sendbuf  = (rank+1)*10
targetRank = mod(rank+1,2)

! both ranks send first and then receive using blocking synchronous transfer
call MPI_Ssend(sendbuf, 1, MPI_INTEGER, targetRank, tag, &
               MPI_COMM_WORLD, ierror)
call MPI_Recv(recvbuf, 1, MPI_INTEGER, targetRank, tag, &
              MPI_COMM_WORLD, status, ierror)
```

- ***MPI_Ssend (blocking synchronous send):*** return only after a matching recv has been posted (i.e., "***handshaking***" between send and recv)

Example: deadlock (case 2)

```
tag       = 1
sendbuf   = (rank+1)*10
targetRank = mod(rank+1,2)

! both ranks receive first and then send using blocking (asynchronous) transfer
call MPI_Recv(recvbuf, 1, MPI_INTEGER, targetRank, tag, &
              MPI_COMM_WORLD, status, ierror)
call MPI_Send(sendbuf, 1, MPI_INTEGER, targetRank, tag, &
              MPI_COMM_WORLD, ierror)
```

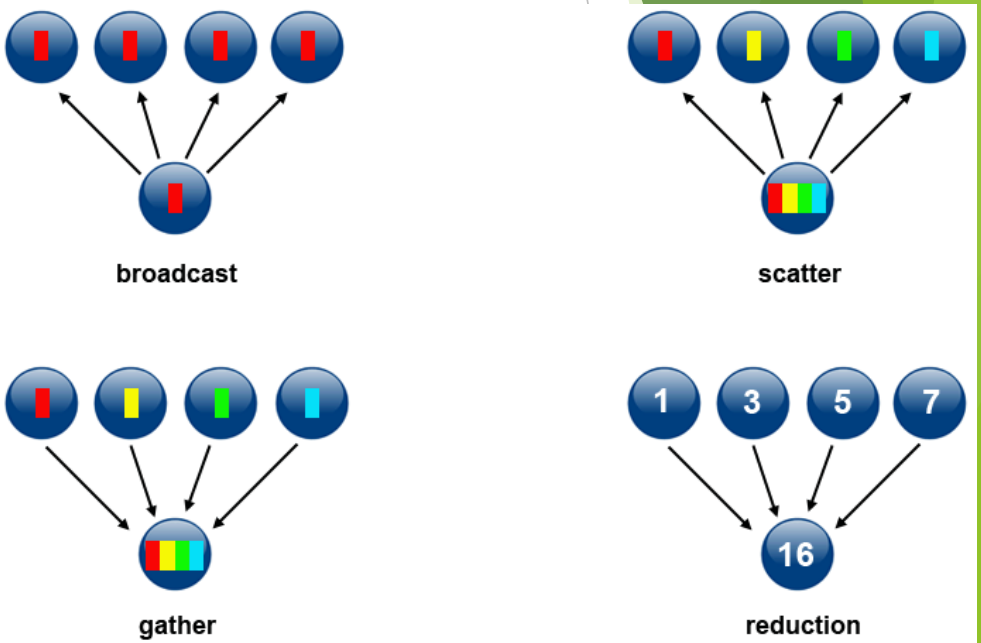
Example: non-blocking communication

```
tag      = 1
sendbuf    = (rank+1)*10
targetRank = mod(rank+1,2)

! both ranks receive first and then send using non-blocking transfer
call MPI_Irecv(recvbuf, 1, MPI_INTEGER, targetRank, tag, &
               MPI_COMM_WORLD, request(1), ierror)
call MPI_Isend(sendbuf, 1, MPI_INTEGER, targetRank, tag, &
               MPI_COMM_WORLD, request(2), ierror)
call MPI_Waitall(nreq, request, status, ierror)
```

- By using non-blocking functions, this example will not cause a deadlock

Collective communication

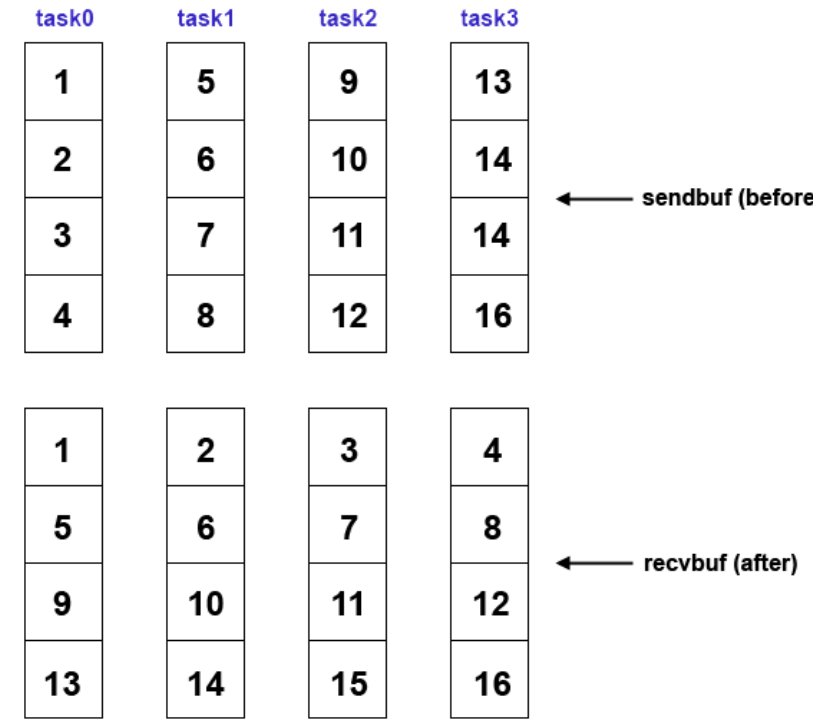
- Types of collective operations
  - *Synchronization* (e.g., MPI_Barrier)
  - *Data movement* (e.g., MPI_Bcast, MPI_Scatter_MPI_Alltoall)
  - *Collective computation* (e.g., MPI_Reduce)



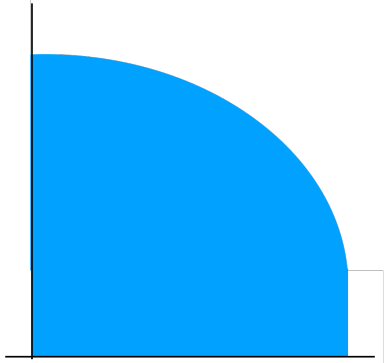
Collective communication

- MPI_Allgather(): first collect data from all ranks and then distribute them to all ranks
- MPI_Allreduce(): first reduce data from all ranks and then distribute them to all ranks
- MPI_Alltoall(): all ranks send messages to all ranks

- See supplemental information for more complete descriptions!



Challenge: parallelize the pi calculation using MPI (additional bonus credit 0.5)



$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

- To get the bonus credit, please submit the code and screenshot to the TAs by the start of next class (12/29/2022)

serial version

```
1  program pi
2
3      use OMP_LIB
4
5      implicit none
6      integer :: i
7      real    :: x, dx , sum, t1, t2
8      integer,parameter  :: n = 10000000000
9
10     dx = 1./float(n)
11
12     t1 = omp_get_wtime()
13     do i=0,n-1
14         x = (i+0.5)*dx
15         sum = sum + 4.0/(1.0+x*x)
16     enddo
17     t2 = omp_get_wtime()
18     print *, "Pi   =", dx*sum
19     print *, "Time =", (t2-t1)
20
21 end program pi
```

Parallel computing -- summary

- Parallelism is the future of computing!
- Flynn's taxonomy (types of computing architecture): SISD, SIMD, MISD, ***MIMD***
- Memory architecture: ***shared-memory*** and ***distributed-memory*** systems
- Terminology: speedup ($S_N \equiv \frac{t_1}{t_N}$), parallel efficiency ($\epsilon_N \equiv \frac{S_N}{N}$), parallel overhead, embarrassingly parallel, (control/data) dependency, load balancing
- Scalability: the ability for a system to increase speedup with more processes (P)
  - ***Strong scaling***: for ***fixed problem size***, how does the computing time scale with P
  - ***Weak scaling***: for ***fixed problem size per processor***, how does computing time scale with P

Parallel computing -- summary

- *OpenMP (open multi-processing)*
  - *Multi-thread*, *shared-memory*, *directive-based* parallelism (e.g., `!$omp parallel`)
  - Fork-join model by launching multiple threads
  - *Work-sharing constructs*: do, sections, single
  - *Synchronization constructs*: critical, barrier
  - Think carefully about whether the variables are *shared* or *private* to avoid *race conditions*!!
- *MPI (message passing interface)*
  - Parallelism by *passing messages* among different processes (identified by their *ranks*)
  - Users *call the MPI routines* to tell the program how to transfer data among ranks
  - *Point-to-point communication*: MPI_Send, MPI_Recv, MPI_Ssend, MPI_Isend, MPI_Irecv
  - *Collective communication*: MPI_Barrier, MPI_Bcast, MPI_Gather, MPI_Scatter, MPI_Alltoall...
  - Think carefully about whether the function calls are *blocking* or *non-blocking* to avoid *deadlocks*!!

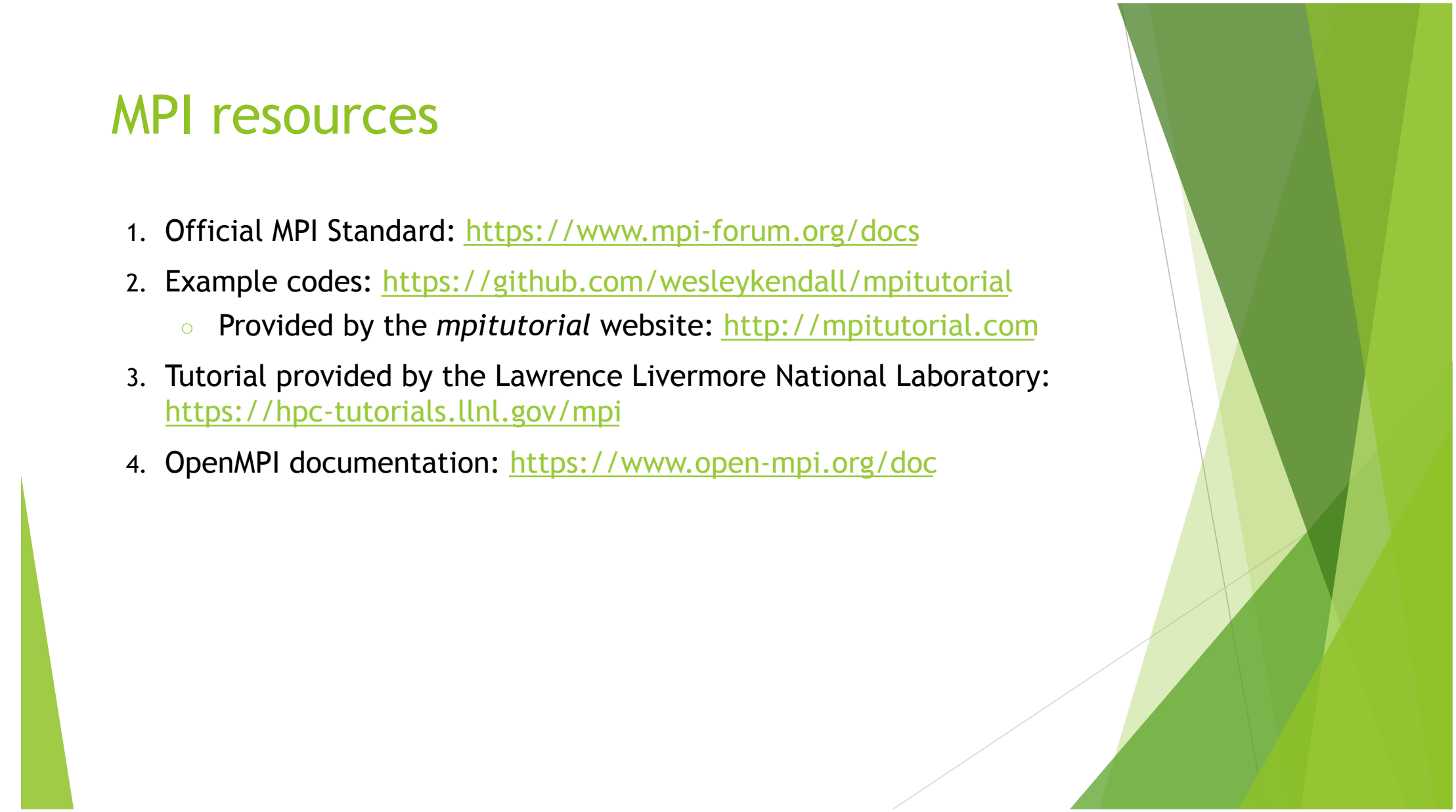
Supplemental Information

OpenMP resources

1. Official specifications: https://www.openmp.org/specifications
2. Official example codes:
   1. https://www.openmp.org/wp-content/uploads/openmp-examples-5.0.0.pdf
   2. https://github.com/OpenMP/Examples
3. Tutorial provided by the Lawrence Livermore National Laboratory: https://hpc-tutorials.llnl.gov/openmp
4. Tutorial provided by IBM (for C): https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.cbclx01/tuoptppp.htm

MPI resources

1. Official MPI Standard: https://www.mpi-forum.org/docs
2. Example codes: https://github.com/wesleykendall/mpitutorial
   - Provided by the *mpitutorial* website: http://mpitutorial.com
3. Tutorial provided by the Lawrence Livermore National Laboratory: https://hpc-tutorials.llnl.gov/mpi
4. OpenMPI documentation: https://www.open-mpi.org/doc



References & acknowledgements

- LLNL HPC Tutorial: https://hpc-tutorials.llnl.gov
- Course materials of Computational Astrophysics from Prof. Kuo-Chuan Pan (NTHU)
- Course materials of Computational Astrophysics from Prof. Hsi-Yu Schive (NTU)
- Course materials of Computational Astrophysics and Cosmology from Prof. Paul Ricker (UIUC)
- “Computational Physics” by Rubin H. Landau, Manuel Jose Paez and Cristian C. Bordeianu
- “Scientific Computing – An Introductory Survey” by Michael T. Heath