# Introduction to Parallel Programming

## 2022 GPU Computing Workshop Series

**_Shiquan Su_**
*CISL HPCD Consulting Services Group*

NCAR
UCAR

**FEBRUARY 3rd, 2022**

# Workshop Etiquette

- Please mute yourself and turn off video during the session.
- Questions may be submitted in the chat and will be answered when appropriate. You may also raise your hand, unmute, and ask questions during Q&A at the end of the presentation.
- By joining today, you are agreeing to UCAR’s Code of Conduct
- Recordings & other material will be archived & shared publicly.
- Feel free to follow up with the GPU workshop team at our office hours, our Slack, or submit support requests to support.ucar.edu
    - Office Hours: Tuesdays, times and connection details TBD

# Workshop Series and Logistics

- Scheduled biweekly through August 2022 (short break in May)

- Sequence of sessions detailed on main webpage
  - Full workshop course description document/syllabus
  - Useful resources for added self-directed learning included

- Registrants may use workshop's Project ID & Casper core hours
  - Please only **submit non-production, test/debug scale jobs**
  - For non-workshop jobs, request an allocation. Easy access startup allocations may be available for new faculty and graduate students.
  - New NCAR HPC users should review our HPC Tutorials page

- Interactive sessions will share code via GitHub and JupyterHub notebooks. More details will be shared prior to these sessions.

# GPU Community Engagement

Below are recommended community resources

- Join NCAR GPU Users Slack and #gpu_workshop_participants

- Consider joining other Slack communities or online spaces
  - OpenACC and GPU Hackathon Slack workspace (NVIDIA managed)
  - If you're excited about Julia, they have a Slack and #GPU channel
  - NCAR GPU Tiger Team for cutting edge updates and future directions
  - Watch Stackoverflow tags for OpenACC, OpenMP, CUDA, or others

- Prepare an application for an upcoming GPU Hackathon

# Overview

- Basic Principles of Parallel Computing
- Terminology of Parallel Applications
- How Parallelization Enable High Performance Computing
- Primary Issues Encountered in Developing a Parallel Application

You don't need to know how to write code to understand these topics, but you will get more out of this topic if you have a specific computation in mind. If you have coded a serial example of your problem, you can
gather profiling information on it to determine the amount of time spent doing different parts of it. It's those time-consuming parts that should be the target of your parallelization efforts.

# Basic Principles of Parallel Computing

# Why we need parallel computing?

- **Scientific Computing**
  - Modern science problems are too big, time to solution is too long, the tasks are too many.
- **High Performance Computing**
  - To stay ahead of the game, scientists are constantly chase high performance, pushing the limit of the hardware.
- **Parallel Computing**
  - Code efficiency quickly reaches the limit of the clock speed. The overall efficiency can not be improved without running multiple instances of the code at the same time, both on device level (threading/GPU) and node level (MPI)

Galaxy Formation

Planetary Movments

Climate Change

Rush Hour Traffic

Plate Tectonics

Weather

- Though parallel programming requires more time and effort than serial programming, parallel computation is the only way to leverage the enormous power of supercomputers like Cheyenne/Derecho.

- Parallel programming is increasingly relevant for all computing platforms; most personal computers (and even cell phones) today include multiple processing cores and require parallel programs to yield the best performance.

# Derecho - Next Generation Computing at NCAR

The flagship of NCAR’s next gen supercomputer Derecho will feature:

- *320,000 AMD Milan processors for parallel computation,*
- *2,500+ 3rd Gen AMD EPYC nodes,*
- *82 4-way A100 SXM GPU nodes with 40GB of device memory*
    - *1,555 GB/s high-bandwidth memory rate*
    - *600 GB/s NVLink GPU interconnect*
- *Total 692 TB system host memory*
- *Derecho will offer* ***3.5x*** *computational capacity vs Cheyenne*

*19.87 peak PetaFlops*

DERECHO

NCAR Supercomputer

*5.34 peak PetaFlops*

# Terminology of Parallel Applications

- **Serial code** uses a single thread of execution working on a single data item at any one time.

- **Parallel code** has more than one concurrent process.
  - Single thread of execution operating on multiple data items simultaneously (vectorized thread on CPUs or block of threads on GPUs)
  - Multiple threads of execution in a single executable
  - Multiple executables or tasks working on the same problem
  - Any combination of the above

- In the context of HPC, a **task** is any distinct stream of instructions and memory address space issued by a parallel code. A task may run in parallel with other tasks and may communicate with each other.

- **MPI** is a common interface standard used to perform communication among tasks explicitly. An MPI process may be called a **rank**.

  **N** tasks running at the same time can be **N** way parallelized.

- **OpenMP** is a programming model for launching a set of tasks.
  - These tasks are called **threads** within a single process. In contrast to MPI, all tasks in OpenMP use the shared memory on a single system, i.e., they must share the same virtual address space. Communication takes place through shared memory.

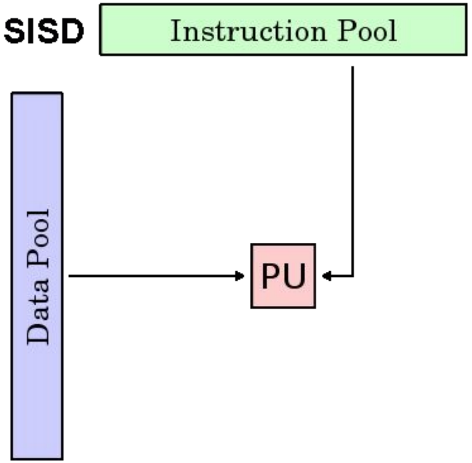
# Flynn’s Taxonomy - Classifying Compute Tasks

Used since 1966, this two-by-two table classifies compute tasks across dimensions of **Instruction Streams** and **Compute Streams**. Each kind of stream can be classified as ***single*** or ***multiple***.

| | Single Data | Multiple Data |
|---|---|---|
| Single Instruction | ● *SISD*<br>● typical CPU thread | ● *SIMD*<br>● vector processors<br>● GPU thread blocks |
| Multiple Instruction | ● *MISD*<br>● possibly set of filters<br>● fault tolerance and redundancies | ● *MIMD*<br>● cluster of nodes<br>● multi-core CPU |

# Flynn’s Taxonomy - Classifying Compute Tasks

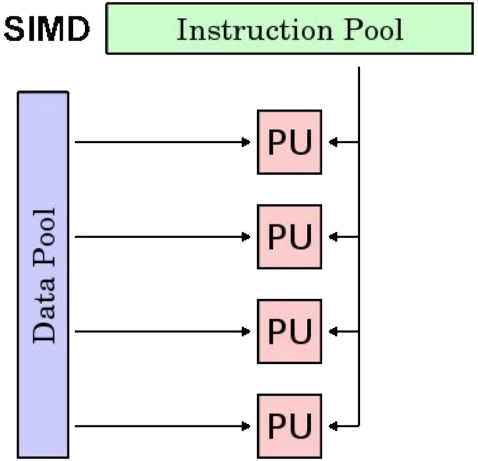
Used since 1966, this two-by-two table classifies compute tasks across dimensions of **Instruction Streams** and **Compute Streams**. Each kind of stream can be classified as ***single*** or ***multiple***.



| | Single Data | Multiple Data |
|---|---|---|
| Single Instruction | • *SISD*<br>• typical CPU thread | • *SIMD*<br>• vector processors<br>• GPU thread blocks |
| Multiple Instruction | • *MISD*<br>• possibly set of filters<br>• fault tolerance and redundancies | • *MIMD*<br>• cluster of nodes<br>• multi-core CPU |

# Flynn’s Taxonomy - Classifying Compute Tasks

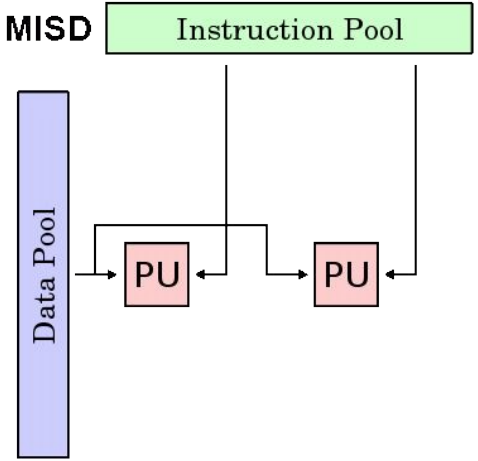
Used since 1966, this two-by-two table classifies compute tasks across dimensions of **Instruction Streams** and **Compute Streams**. Each kind of stream can be classified as ***single*** or ***multiple***.



| | Single Data | Multiple Data |
|---|---|---|
| Single Instruction | ● *SISD*<br>● typical CPU thread | ● *SIMD*<br>● vector processors<br>● GPU thread blocks |
| Multiple Instruction | ● *MISD*<br>● possibly set of filters<br>● fault tolerance and redundancies | ● *MIMD*<br>● cluster of nodes<br>● multi-core CPU |

# Flynn’s Taxonomy - Classifying Compute Tasks

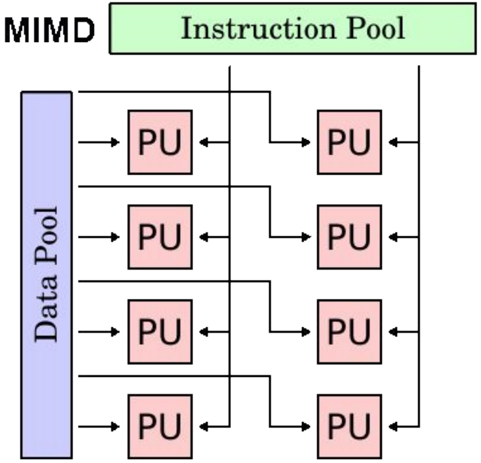
Used since 1966, this two-by-two table classifies compute tasks across dimensions of **Instruction Streams** and **Compute Streams**. Each kind of stream can be classified as ***single*** or ***multiple***.



| | Single Data | Multiple Data |
|---|---|---|
| Single Instruction | ● *SISD*<br>● typical CPU thread | ● *SIMD*<br>● vector processors<br>● GPU thread blocks |
| Multiple Instruction | ● *MISD*<br>● possibly set of filters<br>● fault tolerance and redundancies | ● *MIMD*<br>● cluster of nodes<br>● multi-core CPU |

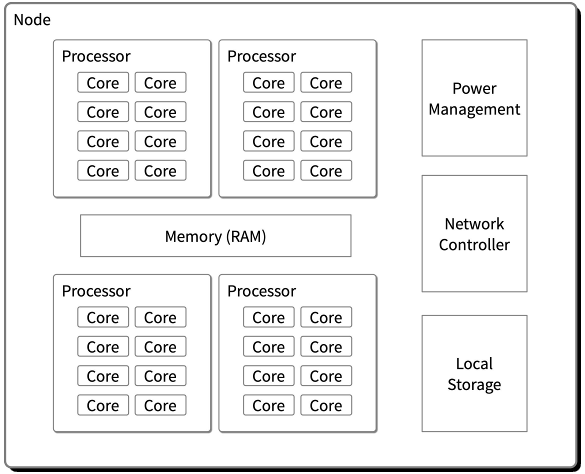
# Flynn’s Taxonomy - Classifying Compute Tasks

Used since 1966, this two-by-two table classifies compute tasks across dimensions of **Instruction Streams** and **Compute Streams**. Each kind of stream can be classified as ***single*** or ***multiple***.



| | Single Data | Multiple Data |
|---|---|---|
| Single Instruction | • *SISD*<br>• typical CPU thread | • *SIMD*<br>• vector processors<br>• GPU thread blocks |
| Multiple Instruction | • *MISD*<br>• possibly set of filters<br>• fault tolerance and redundancies | • *MIMD*<br>• cluster of nodes<br>• multi-core CPU |

A **node** is a standalone physical computer unit with a network connection that typically runs its own instance of the operating system.



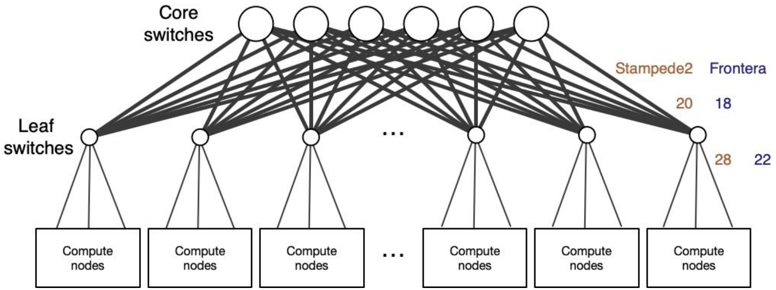
Supercomputer clusters are composed of nodes connected by a communications network.

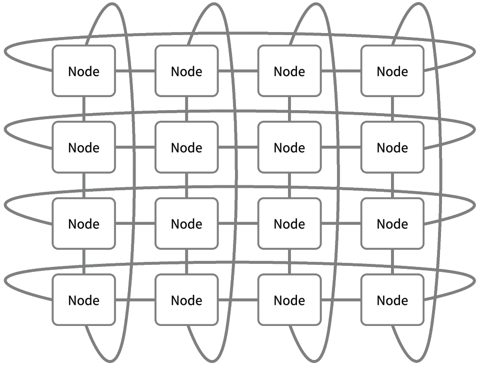
The nodes in a cluster like Cheyenne/Derecho are packaged into units that can be mounted in a dense configuration that provides appropriate power, cooling, and network connections.

A **Cluster** is a collection of nodes that function in some way as a single resource. They may be administered as a unit and provide a uniform environment for tasks to run on the cluster.

On Cheyenne, the software installed on each node is identical, and access from each cluster node to external resources is uniform.

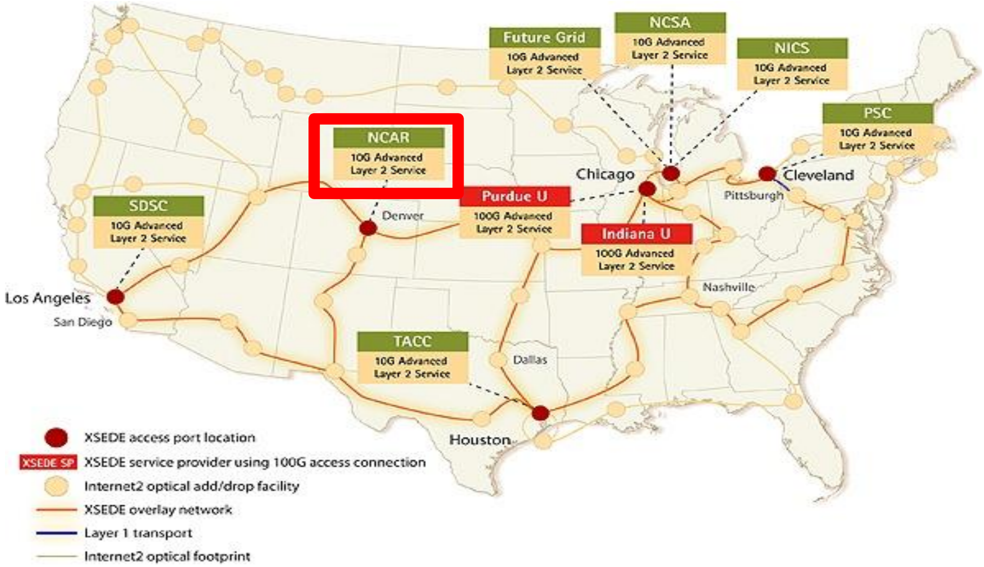
Nodes of a cluster are normally assigned to users by a **Scheduler**, such as PBS or Slurm. An assignment of a set of nodes for exclusive use by a user for a certain amount of time is called a **Job**.





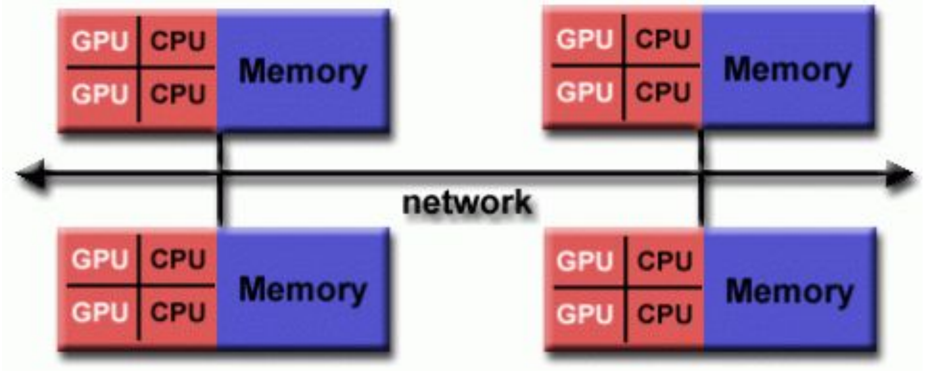
A **Grid** is the software stack and hardware infrastructure designed to handle the technical and social challenges of sharing resources across networking and institutional boundaries. A collaborative grid network allows remote execution of large simulations as well as sending files or sharing virtual environments.

One of the major grids in the US is **XSEDE**, wherein resources are connected by a dedicated high-performance communications network spanning the country.


Future Grid
10G Advanced Layer 2 Service
NCSA
10G Advanced Layer 2 Service
NICS
10G Advanced Layer 2 Service
PSC
10G Advanced Layer 2 Service
NCAR
10G Advanced Layer 2 Service
Purdue U
100G Advanced Layer 2 Service
Indiana U
100G Advanced Layer 2 Service
SDSC
10G Advanced Layer 2 Service
TACC
10G Advanced Layer 2 Service
Chicago
Cleveland
Pittsburgh
Denver
Nashville
Los Angeles
San Diego
Dallas
Houston
XSEDE access port location
XSEDE SP
XSEDE service provider using 100G access connection
Internet2 optical add/drop facility
XSEDE overlay network
Layer 1 transport
Internet2 optical footprint

- In **Distributed Memory** programming, each task owns part of the data, and other tasks must send a message to the owner in order to update that part of the data.

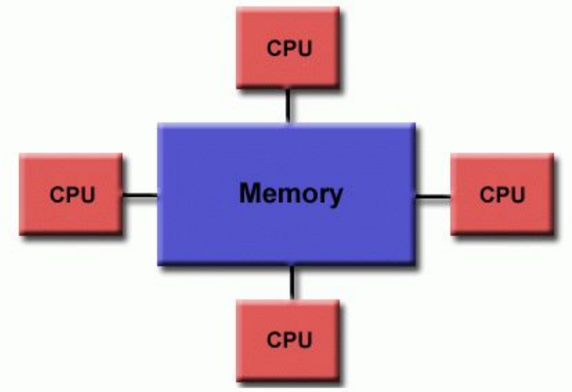
- For Cheyenne/Derecho, there are many nodes & memory associated with one **node** is not directly accessible from another. A distributed memory parallel program has at least one separate executable on each node. Interface standards like **Message-Passing Interface (MPI)** facilitate distributed memory programming for supercomputers.



**Shared Memory** programming implies that all threads of execution within the same parent task can uniformly address the same variables. When threads execute in parallel, however, there is no guarantee for the order of the running threads.

Communication among threads is efficient since any changes to shared memory is immediately visible to all threads, but the programmer must coordinate memory reads and writes so that each thread receives the expected values from memory.

**OpenMP**, a common API for facilitating shared memory programming, includes mechanisms to ensure that the above operations occur in the desired order.

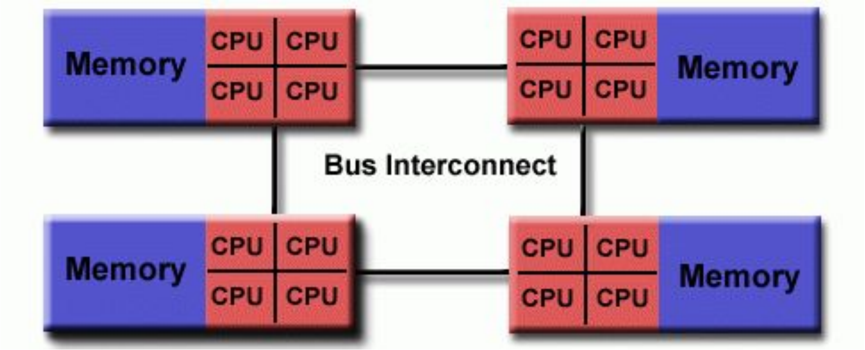


A shared memory computer has multiple cores with access to the same physical memory. The cores may be part of multicore processor chips, and there may be multiple processors within the computer.

If multiple chips are involved, access is not necessarily uniform.

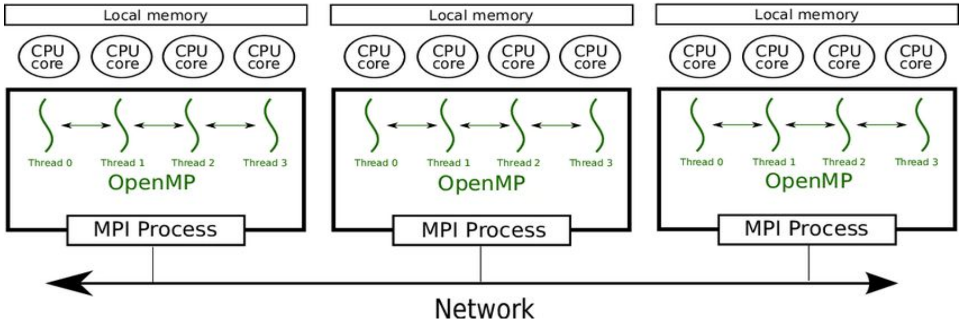
From the perspective of an individual core, some physical memory locations have lower latency or higher bandwidth than others. This situation is called **non-uniform memory access (NUMA).** Nearly all modern computers rely on NUMA designs.

NUMA Architecture

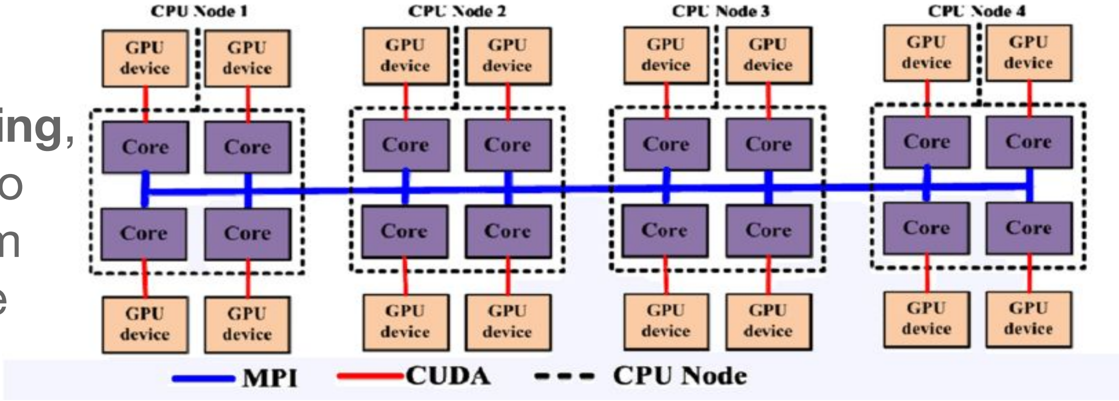


# Hybrid strategy

Using multithreaded tasks designed with shared memory programming to take advantage of multiple cores on a single node while simultaneously using distributed memory strategies to coordinate with tasks on other nodes.



Also known as **hybrid programming**, this technique provides flexibility to the programmer to map parallelism that exists in the program onto the characteristics of the machine.



**Data Parallelism**:

Each parallel worker applies the same operations to a different segment of data.

Each process does the same work on a unique piece of data.

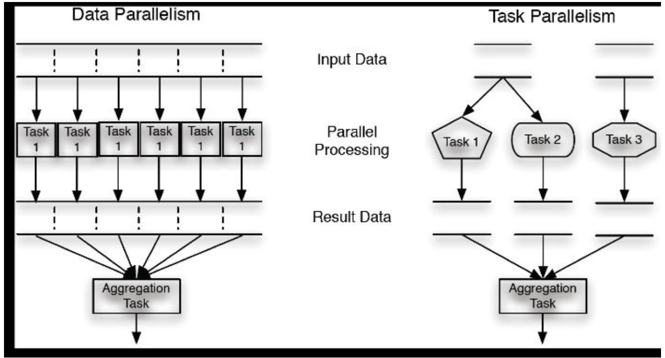
Follow the "owner computes" principle. Divide the data among workers. Each worker computes its own data.

**Functional Parallelism,** also called **task parallelism**:

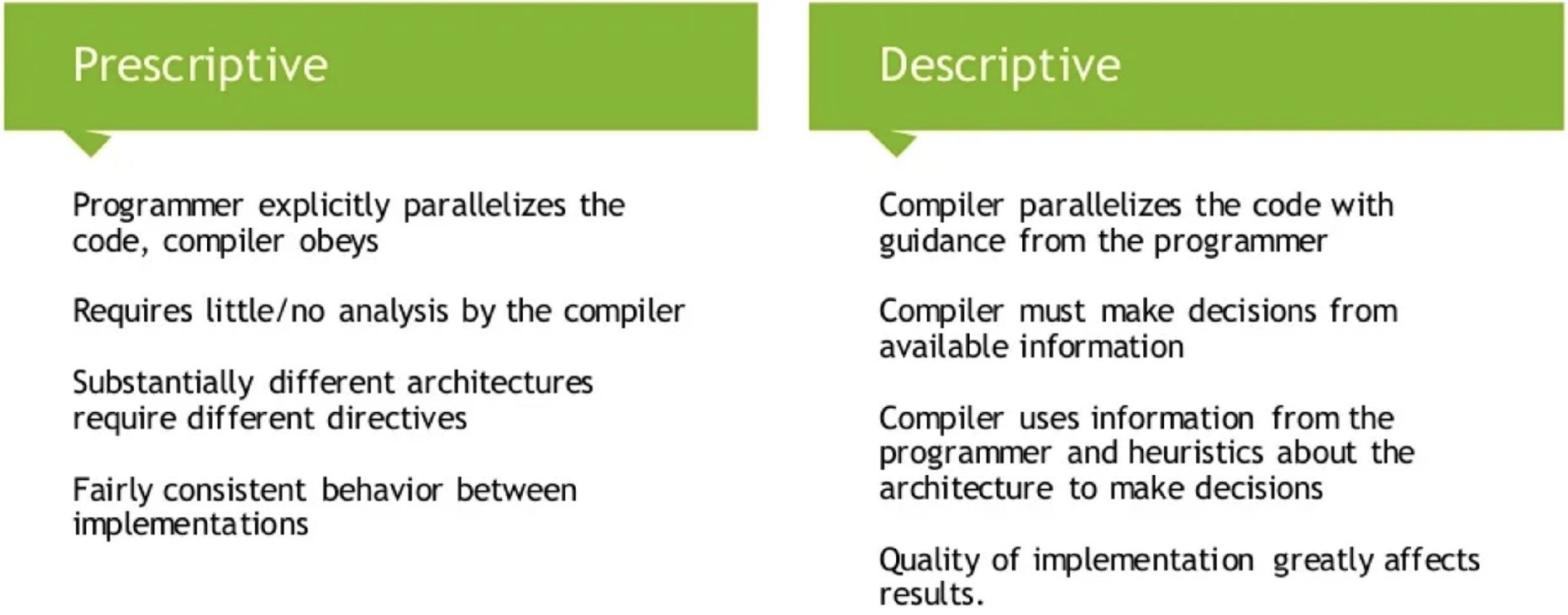
Each parallel worker performs different operations on the data.

Each process performs a different "function" or executes a different code section.

Message-passing libraries (MPI) is the main communication tool among functions.



# Descriptive or Prescriptive Parallelism



## Prescriptive

Programmer explicitly parallelizes the code, compiler obeys

Requires little/no analysis by the compiler

Substantially different architectures require different directives

Fairly consistent behavior between implementations

## Descriptive

Compiler parallelizes the code with guidance from the programmer

Compiler must make decisions from available information

Compiler uses information from the programmer and heuristics about the architecture to make decisions

Quality of implementation greatly affects results.

OpenMP

MPI CUDA

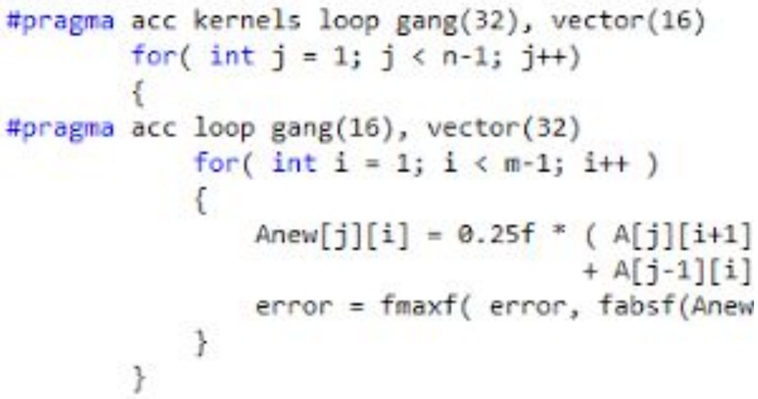
OpenACC

do concurrent std::par

# How Parallelization Enables High Performance Computing

If the scale of your computation allows for shared memory programming, the easiest way to exploit shared memory parallelism is to insert **OpenMP/OpenACC directives** into your code to execute specific loops in parallel.

One advantage is that parallel code may still compile as a serial code; unless the compiler is instructed to honor the OpenMP directives, it will ignore them and produce a serial program. Also common is to explicitly use threads to create a new shared memory parallel program based on an existing serial program. If you choose to create and manage threads explicitly, ensure that access to shared data from different threads is appropriately synchronized.



```
#pragma acc kernels loop gang(32), vector(16)
        for( int j = 1; j < n-1; j++)
        {
#pragma acc loop gang(16), vector(32)
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25f * ( A[j][i+1]
                                     + A[j-1][i]
                error = fmaxf( error, fabsf(Anew
            }
        }
```

**Dependency** is important to parallel programming because they are one of the primary inhibitors to parallelism. One of the common places to find dependency is the **variable dependence**.  See the following two pseudo code sections:

## Difficult to parallelize

Each iteration depends on the next iteration, so two consecutive iterations can't run at the same time.

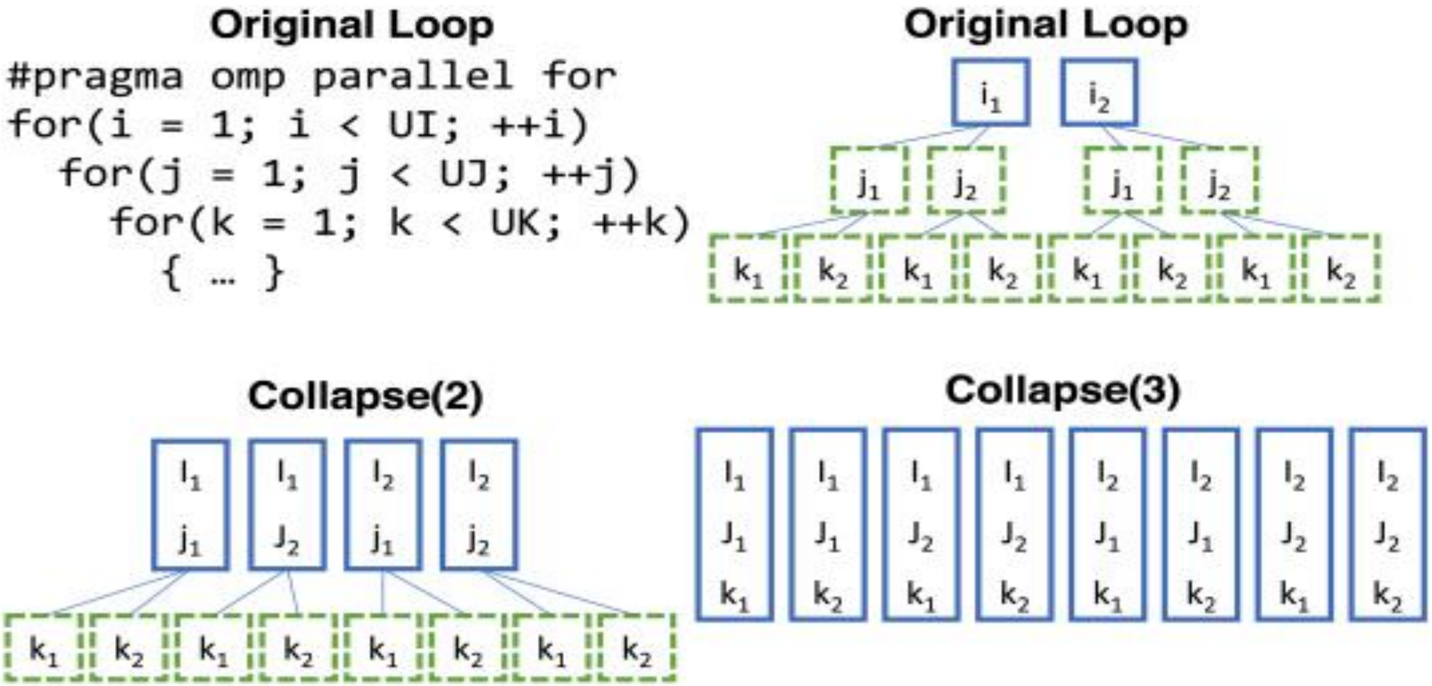
```
DO J = MYSTART, MYEND
    A(J) = A(J-1) * 2.0
END DO
```

## Easy to parallelize

Each iteration is independent. Multiple iterations can run at the same time.

```
  task 1      task 2
  ------      ------
  X = 2        X = 4
   . .          . .
Y = X**2    Y = X**3
```

**Exploiting Parallelism** is essentially the most important thought process behind parallel programming. One of the best places to exploit parallelism is the loop structure. We can flatten or **Collapse Nested Loops** to generate more parallelism to allow more threads working at the same time.



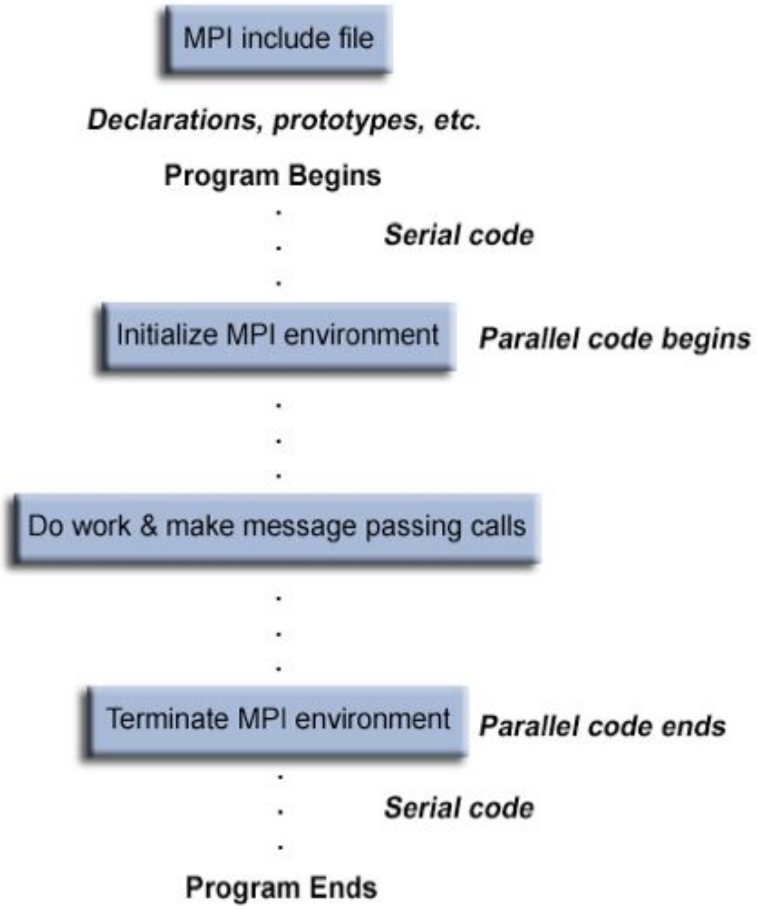
Original Loop

```
#pragma omp parallel for
for(i = 1; i < UI; ++i)
  for(j = 1; j < UJ; ++j)
    for(k = 1; k < UK; ++k)
      { … }
```

**Distributed Memory** parallel programs are much more difficult to write as simple modifications of serial programs. The communication between nodes need to be managed by coordinating **MPI** commands.

In distributed memory programs, often only one or a few tasks will be doing I/O.

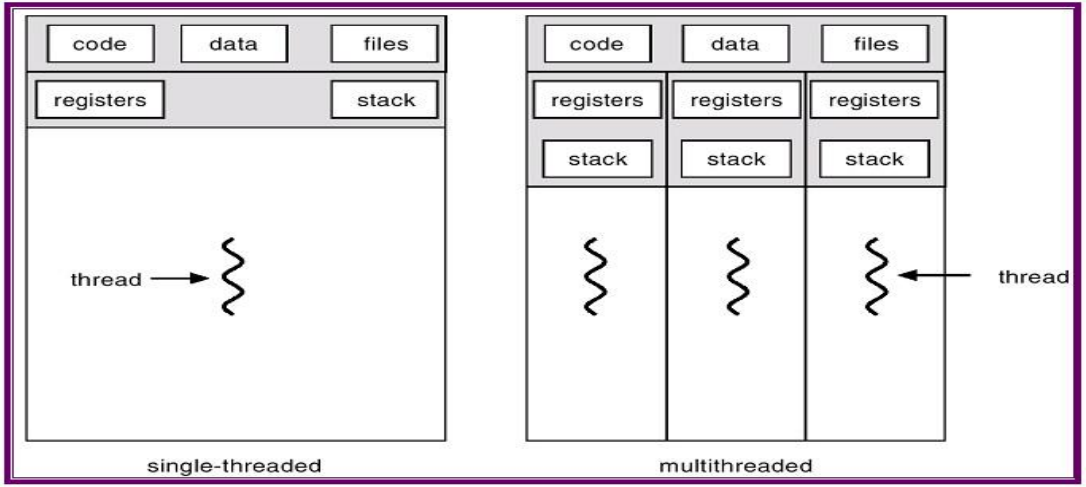
The tasks responsible for I/O may need to distribute and collect data from the other tasks.


MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work & make message passing calls
Terminate MPI environment
Parallel code ends
Serial code
Program Ends

- A program can also be parallelized by **taking advantage of language features, extensions, or libraries** that are already capable of parallel computation.
- Some higher-level programming languages support distributed arrays. Languages that support distributed arrays take care of scattering and gathering blocks of arrays as necessary.
- Computational libraries, such as **ScaLAPACK, FFTW, PETSc, and the Intel oneAPI Math Kernel Library (oneMKL)**, offer distributed memory parallel algorithms.
- If possible, use existing libraries and software features that support parallel computation. These tools are designed to handle common parallelization needs in a general way, and they are subject to extensive testing.
- Leverage vendor support (**Descriptive Parallelism**), and community collaboration (**Hackathons**).

# Primary Issues Encountered in Developing a Parallel Application

- In parallel computing, **efficiency** is the proportion of simultaneously available resources utilized by a computation. By definition, proper HPC code aims for the highest possible efficiency. For computationally-intensive code, we usually focus on whether every processor is always performing useful work for the algorithm.

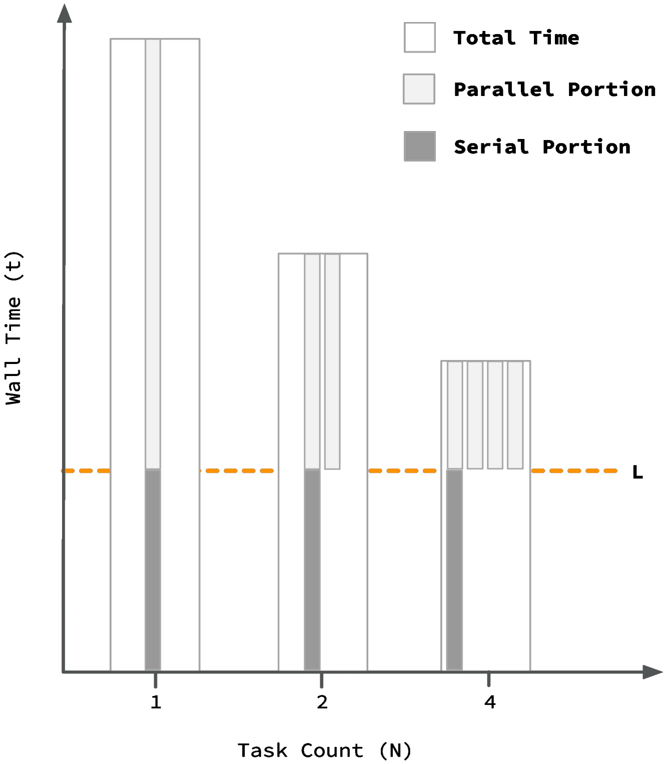


- A serial program running alone seriously under utilizes a cluster node and will waste your allocation.

- The ratio of **FLoating-point OPerations per Second (FLOPS)** to the peak theoretical performance is a common way to report overall efficiency for parallel code.

- The **Peak Theoretical Performance** is calculated with the assumption that each processor core performs every possible floating-point operation during each clock cycle.

# Amdahl's Law



In terms of speedup, S, Amdahl's Law states:

$$S = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{P_1 + L}{(P_1 / N + L)} = \frac{N \cdot (P_1 + L)}{(P_1 + N \cdot L)} = \frac{N{\cdot}P_1 + N{\cdot}L}{P_1 + N{\cdot}L}$$

L = Wall Time of Linear/Serial Portion
P = Wall Time of Parallel Portion
N = number of task count.

Algorithm design/programming practice:

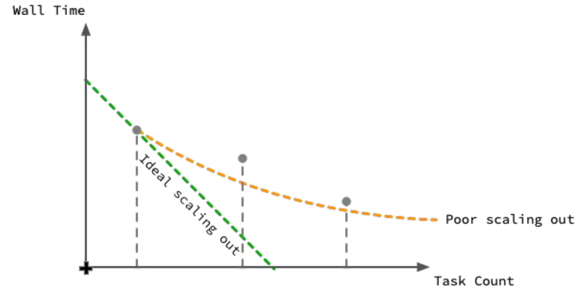
Good: $P >> L$

Bad: $P << L$

# Evaluating parallelization effort and outcome: **Scaling Behavior**

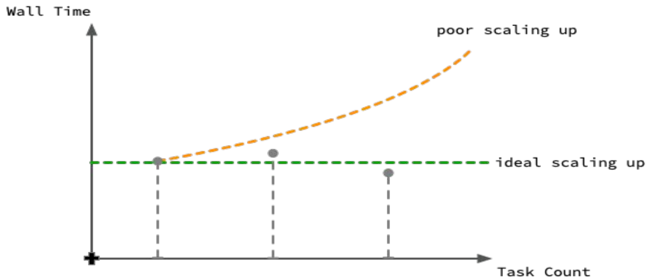
## **Strong Scaling**:

Fix the size of the problem, finish the job on as many resources as possible with shortest wall time.



## **Weak Scaling**:

Fix the workload on each task, finish a job as large as possible with as many resources as possible in the same amount of wall time.



**Hybrid Case**: Iterative optimization/evolution problem, weak scaling on the system size, strong scaling within time step.

# Thinking Big

Scaling up used to mean using 8, 16, or 32 cores, but now scaling up means hundreds, thousands, or even hundreds of thousands of cores!

This requires different thinking because many programs that scale acceptably on smaller machines will not perform well when scaled to large machines.

For traditional domain science codes. A parallelization strategy is to split each iteration of an outer loop into separate computations and distribute these computations among **Parallel Tasks**. Although the outermost parallelizable loop in the program is one of the most important sites for scrutiny, simply spreading it across tasks may not be the best design strategy for scalable parallel programs due to data dependency and messaging overhead.

Dividing data and computations over tasks is **Domain Decomposition**; this is a type of data parallelization, as discussed in this topic. If we think of the data arranged in N-dimensional space, the outermost loop might be traverse one of these dimensions. In that case, parallelizing on that loop involves slicing the domain into strips one (or a few) data elements wide.

In small clusters with problems of modest size, nearly all of the time is spent in computation. With computer systems like Cheyenne/Derecho that have thousands of nodes, the computation may not be the only time-consuming part of the job. Attention also needs to be paid to **I/O (both input and output)** at the startup and shutdown of the job.

Ultimately I/O needs to be parallelized too. Furthermore, systems programmers are constantly working to decrease startup times for MPI and other software infrastructure. Issues like these are bound to assume greater importance in the exascale era.

# Additional Learning Resources

We only covered a small amount of parallel programming concepts. Feel free to explore recommended material below for further details and more in depth discussions.

- LLNL's Introduction to Parallel Computing webpage
- Self-paced courses from NCSA and UIUC, hpc-training.org
- XSEDE repository of online learning resources & course catalog

To get a head start on GPU concepts…

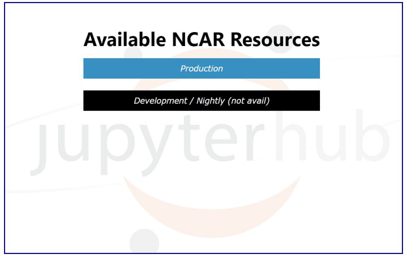
- Document summary of Resources and NVIDIA Documentation
- UCAR curated collection of external learning resources, GDrive
- Other training courses offered and archived online

# THE END

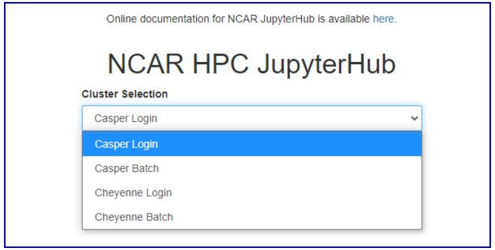
# Questions?

# Additional resources in NCAR

# JupyterHub service on NCAR HPC resources:



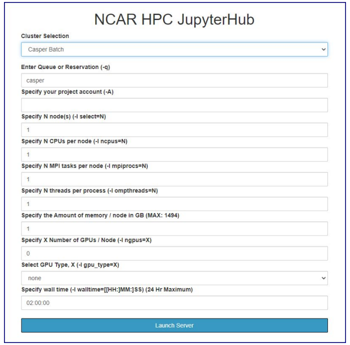
Available NCAR Resources



Online documentation for NCAR JupyterHub is available here.

NCAR HPC JupyterHub

Cluster Selection: Casper Login



NCAR HPC JupyterHub

Cluster Selection: Casper Batch

Enter Queue or Reservation (-q): casper

Specify your project account (-A):

Specify N node(s) (-l select=N): 1

Specify N CPUs per node (-l ncpus=N): 1

Specify N MPI tasks per node (-l mpiprocs=N): 1

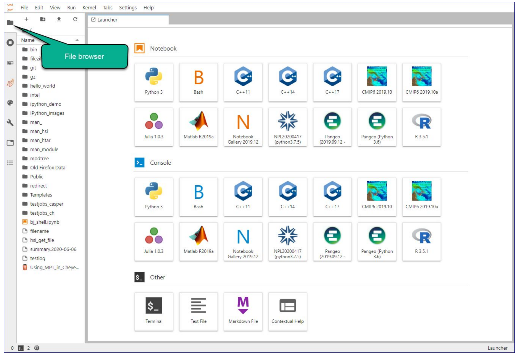
Specify N threads per process (-l ompthreads=N): 1

Specify the Amount of memory / node in GB (MAX: 1494): 1

Specify X Number of GPUs / Node (-l ngpus=X): 0

Select GPU Type, X (-l gpu_type=X): none

Specify wall time (-l walltime=[[HH:]MM:]SS) (24 Hr Maximum): 02:00:00



- Parallel in Python through different engines: DASK, SPARK
- Container effort on WRF and CESM
- Exascale computing, Leveraging community effort, Exascale computing projects, such as HEFFTE, support by SPACK
- Supercomputing in commercial, AWE, Cloud bursting
- GPU programming trainings, community support, and Hackathons