Introduction to Parallel Computing

Byoung-Do (BD) Kim, PhD
Associate Chief Research Information Officer
Director, Center for Advanced Research Computing

USC | Advanced Research Computing
Enabling scientific breakthroughs at scale

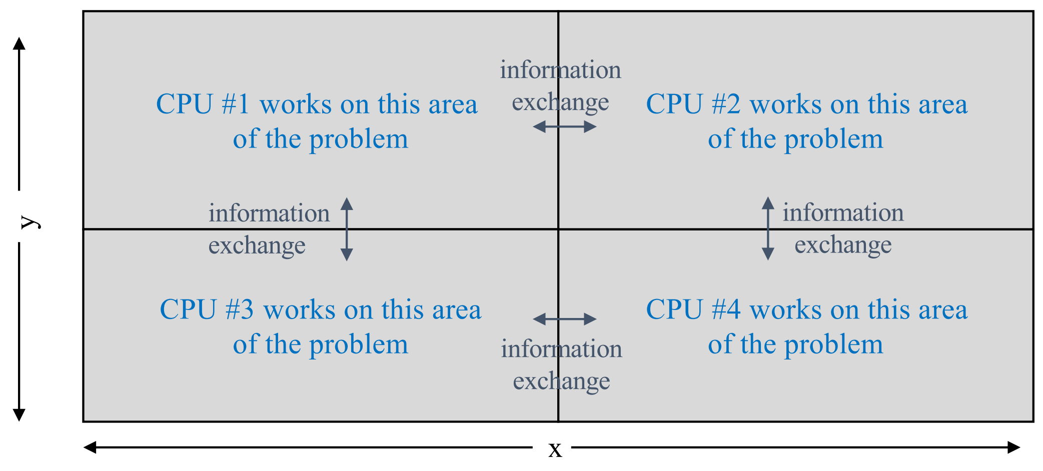
Outline

1. Introduction to Parallel Computing
2. Theoretical background
3. Types of parallel computing systems
4. Programming models
5. Examples
6. Hands-on session

What is Parallel Computing?

- Parallel computing: use of multiple processors or computers working together on a common task.
  - Each processor works on its section of the problem
  - Processors can exchange information

Grid of Problem to be solved

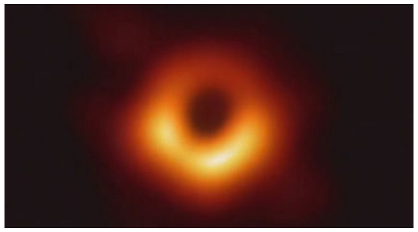

CPU #1 works on this area of the problem
information exchange
CPU #2 works on this area of the problem
information exchange
information exchange
CPU #3 works on this area of the problem
information exchange
CPU #4 works on this area of the problem
y
x

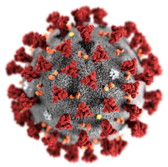
Why Parallel Computing?

- Moore’s Law?
  - Processor speed is no longer double every 18-24 months
  - Multi-core is the norm
- Parallel computing allows one to:
  - solve problems that don’t fit on a single CPU
  - solve problems that can’t be solved in a reasonable time
- We can solve…
  - larger problems
  - faster
  - more cases in a given time

Supercomputing Applications

- Black Hole Simulation
  - Black Hole simulation
  - NASA article
- Formular 1 Racing Car Aerodynamics:
  - F1 aerodynamics
  - Flex wing debate
- Supercomputing vs. COVID-19
  - Drug discovery for COVID-19 using supercomputer
- AI & Supercomputer
  - DeepMind AlphaGo defeated professional Go player 4:1
  - AlphaGo Movie
- Stock Market High-Frequency Trading





Top 500

- List of fastest supercomputers in the world
  - [Top500]
  - [Green500]
  - [List statistics]

IBM-built Summit supercomputer at Oak Ridge National Laboratory

- 148.6 Petaflops (200PF peak)
- 9,216 Power9 CPUs (203K cores)
- 27,648 Nvidia V100 GPUs
- 13MW

Limits of Parallel Computing

- Theoretical Upper Limits
  - Amdahl’s Law
- Practical Limits
  - Load balancing
  - Non-computational sections
  - Communication overhead
- Other Considerations
  - time to re-write code

Parallel Computing: Theoretical background

Theoretical Upper Limit to Performance

- All parallel programs contain:
  - parallel sections (we hope!)
  - serial sections (unfortunately)
- Serial sections limit the parallel effectiveness
- Amdahl’s Law states this formally

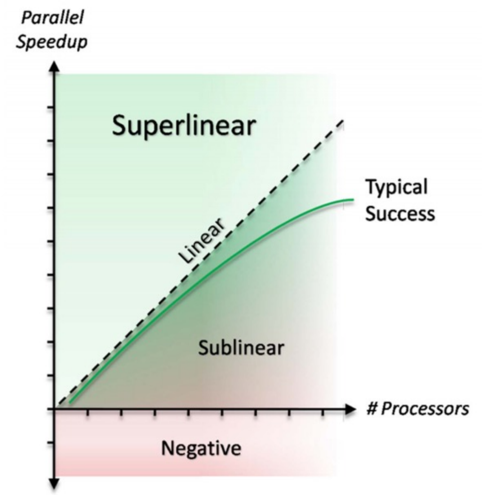
Amdahl's Law

- Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors.

  - Speedup: $S = \frac{T_{serial}}{T_{parallel}}$

  - Effect of multiple processors on speed up: $S = \frac{1}{f_s + \frac{f_p}{N}}$

    where
    - f_s = serial fraction of code
    - f_p = parallel fraction of code
    - N = number of processors

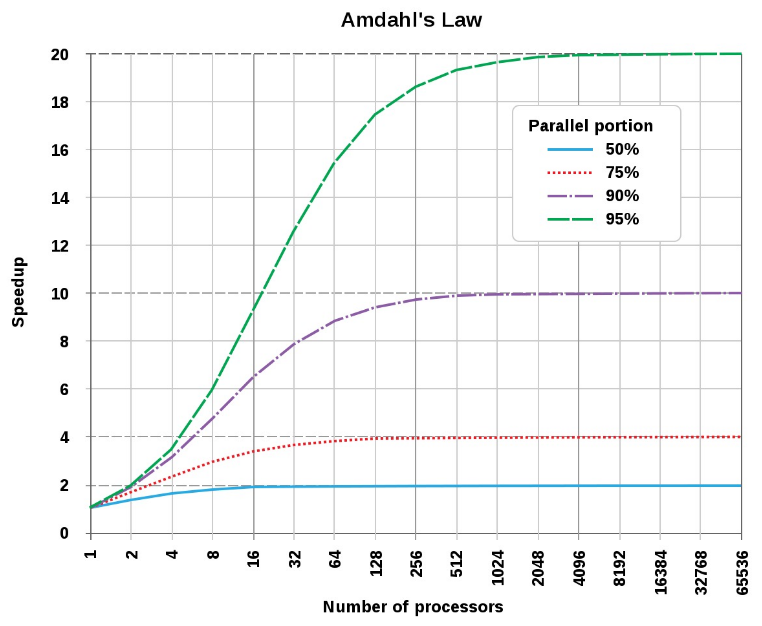


- Amdahl's law in multi-core era: https://research.cs.wisc.edu/multifacet/amdahl/

USC | Advanced Research Computing
Enabling scientific breakthroughs at scale

Illustration of Amdahl's Law

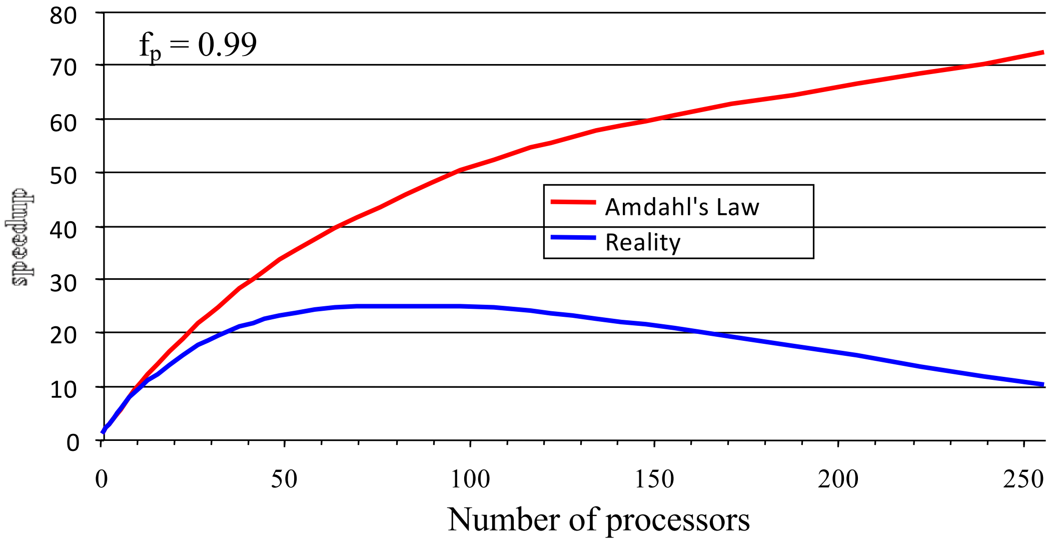
- It takes only a small fraction of serial content in a code to degrade the parallel performance.



From Wikipedia: Amdahl's Law
https://en.wikipedia.org/wiki/Amdahl%27s_law

Practical Limit: Amdahl’s Law vs. Reality

- Amdahl’s Law provides a theoretical upper limit on parallel speedup assuming that there are no costs for *communications*. In reality, communications will result in a further degradation of performance.



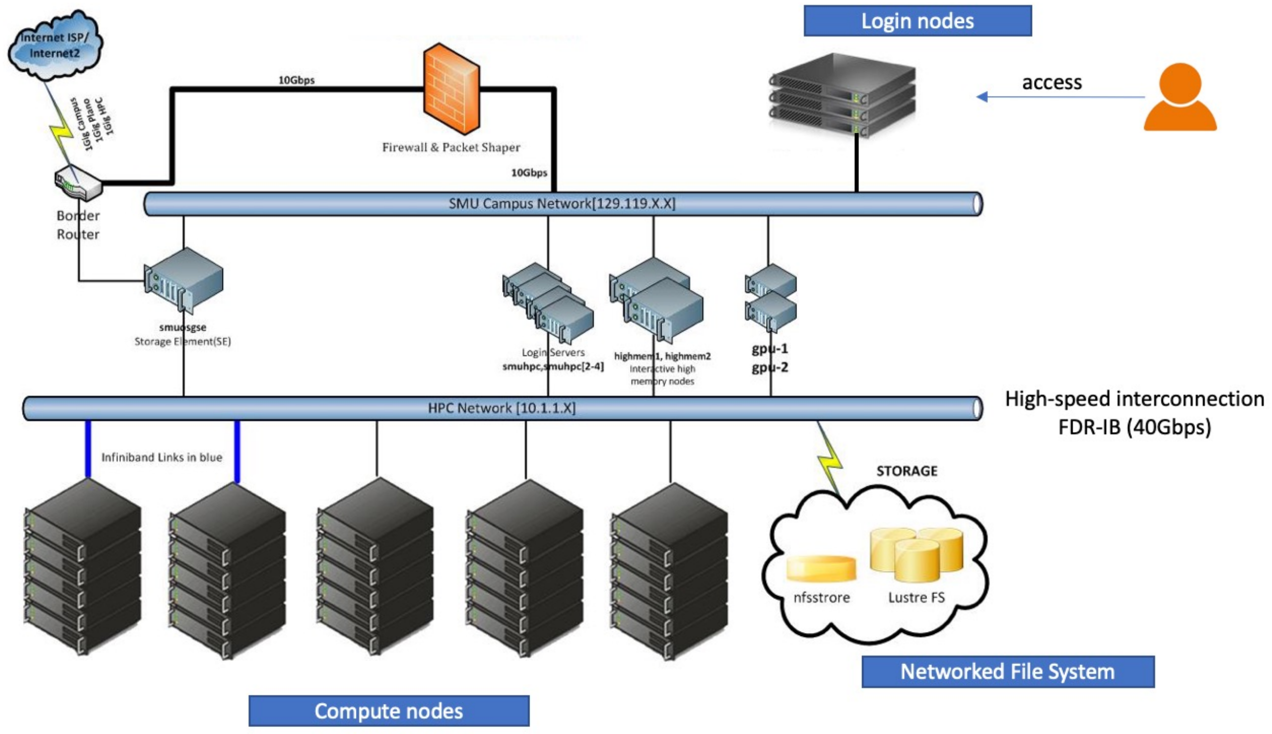
USC | Advanced Research Computing
Enabling scientific breakthroughs at scale

Practical Limit: Amdahl’s Law vs. Reality

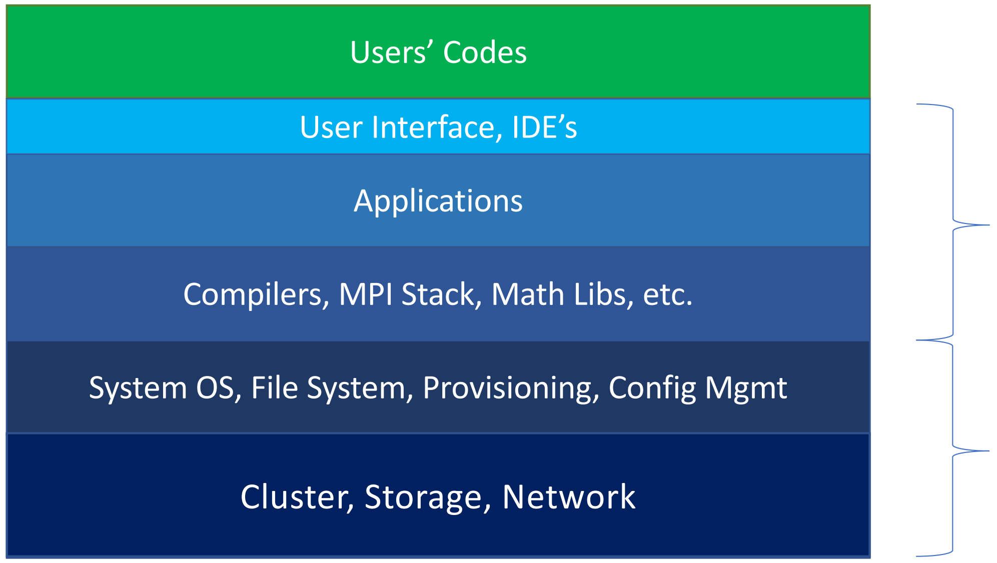
- In reality, the situation is even worse than predicted by Amdahl’s Law
  - Scheduling (shared processors or memory)
  - Communications
  - I/O
- Writing effective parallel applications is difficult!
  - Load balance is important
  - Communication can limit parallel efficiency
  - Serial time can dominate
- Is it worth your time to rewrite your application?
  - Do the CPU requirements justify parallelization?
  - Will the code be used just once?

Parallel System Architecture

HPC in a Nutshell: Cluster System Architecture



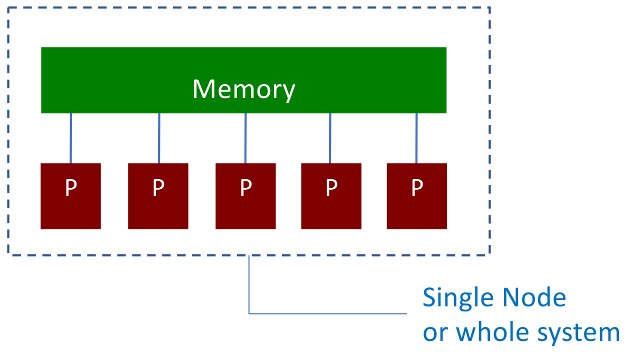
HPC in a Nutshell: SW Stack

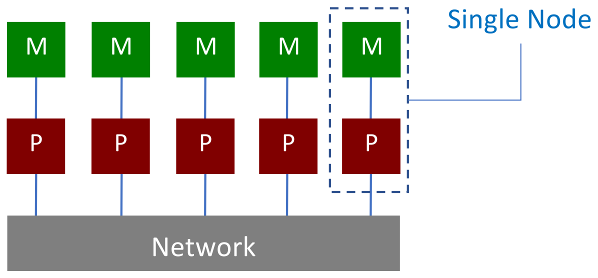

Users' Codes
User Interface, IDE's
Applications
Compilers, MPI Stack, Math Libs, etc.
System OS, File System, Provisioning, Config Mgmt
Cluster, Storage, Network

User-facing
SW stack

System
Administration

Shared vs. Distributed Memory





Shared memory:

- Single address space.
- All processors have access to a pool of shared memory.

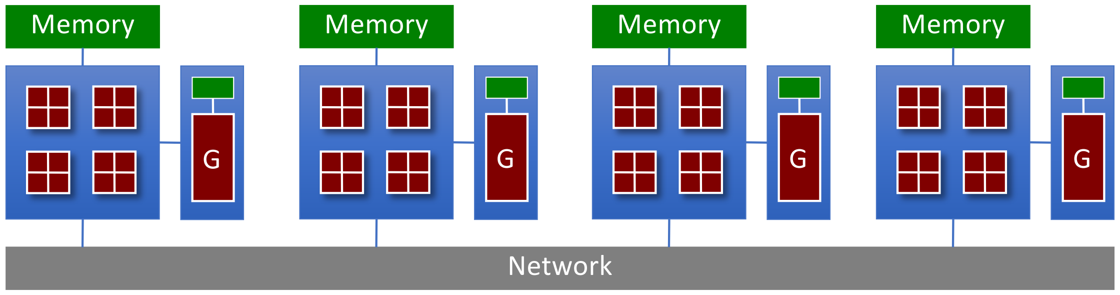
- Methods of memory access :
  - Bus, Crossbar
- Programming model: OpenMP

Distributed memory:

- Each processor has its own local memory.
- Must do message passing to exchange data between processors.

- Methods of memory access :
  - Various topological interconnection
- Programming model: MPI

Multicore with Accelerators



- A limited number of processors N have access to a common pool of shared memory
- To use more than N processors requires data exchange over a network
- Communication details increasingly complex
  - Cache access vs. Main memory access
  - Quick Path / Hyper Transport socket connections
  - Node to node connection via high-speed network
- Load balancing critical for performance
- Requires specific libraries and compilers (CUDA, OpenCL, ACC, etc.)
- Huge performance boot is possible

Parallel Programming Models

Parallel Programming Models

- Data Parallelism
  - Each processor performs the same task on different data

- Task Parallelism
  - Each processor performs a different task on the same data (or on different data)

- Most applications fall between these two

Data Parallel Programming Example

- One code will run on 2 CPUs
- Program has array of data to be operated on 2 CPUs, array is split into two parts.

```
program:
…
if CPU=a then
   low_limit=1
   upper_limit=50
elseif CPU=b then
   low_limit=51
   upper_limit=100
end if
do I = low_limit,
upper_limit
   work on A(I)
end do
...
end program
```

CPU A

```
program:
…
low_limit=1
upper_limit=50
do I= low_limit,
upper_limit
   work on A(I)
end do
…
end program
```

CPU B

```
program:
…
low_limit=51
upper_limit=100
do I= low_limit,
upper_limit
   work on A(I)
end do
…
end program
```

Task Parallel Programming Example

- One code will run on 2 CPUs
- Program has 2 tasks (a and b) to be done by 2 CPUs

```
program:
…
initialize
...
if CPU=a then
   do task a
elseif CPU=b then
   do task b
end if
….
end program
```

CPU A

```
program:
…
initialize
…
do task a
…
end program
```

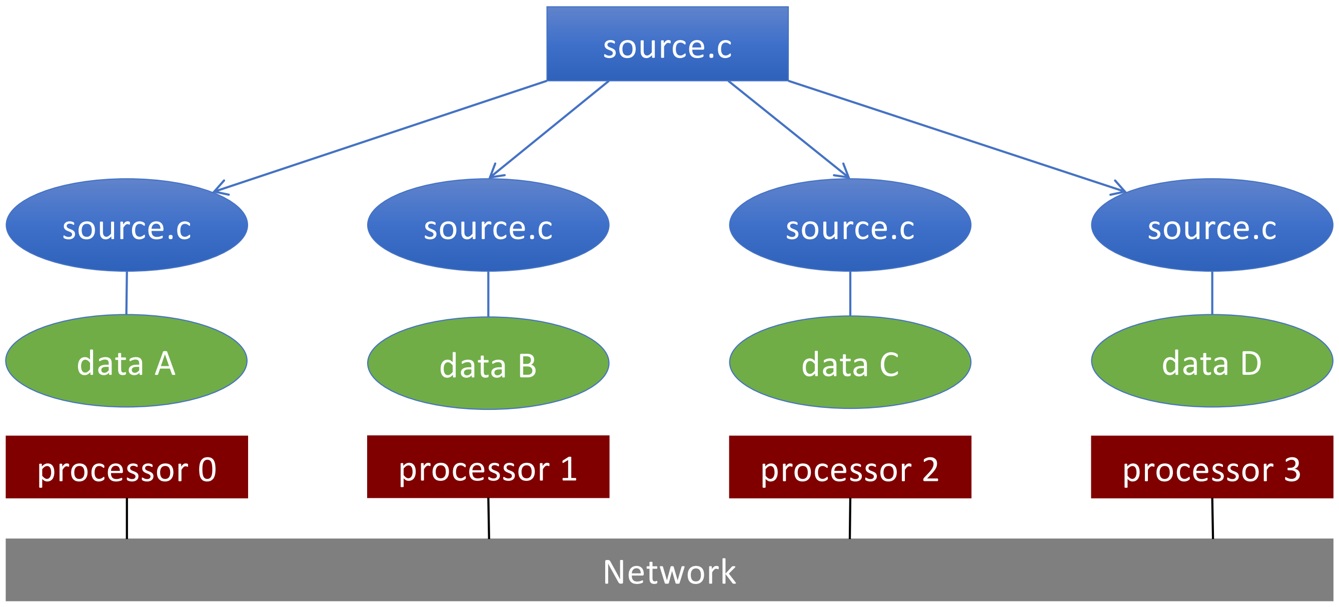
CPU B

```
program:
…
initialize
…
do task b
…
end program
```

Single Program Multiple Data

- SPMD: dominant programming model for shared and distributed memory machines.
  - One source code is written
  - Code can have conditional execution based on which processor is executing the copy
  - All copies of code start simultaneously and communicate and sync with each other periodically

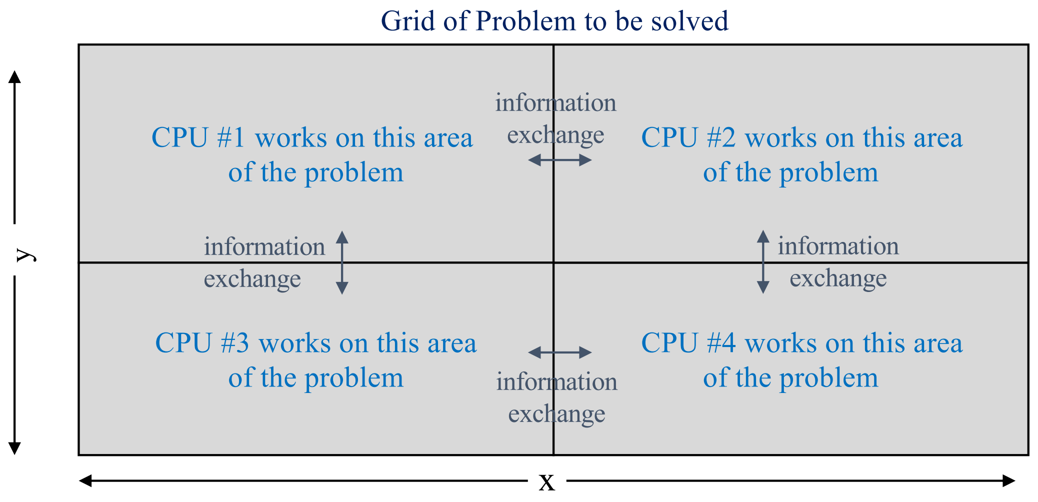
SPMD Model



- Ideal programming model in multi-node system environment

Data Decomposition

- For distributed memory systems, the ‘whole’ grid or sum of particles is decomposed to the individual processors
  - Each CPU works on its section of the problem
  - CPUs/Nodes can exchange information



Grid of Problem to be solved
CPU #1 works on this area of the problem
information exchange
CPU #2 works on this area of the problem
information exchange
information exchange
CPU #3 works on this area of the problem
information exchange
CPU #4 works on this area of the problem
y
x

Domain Decomposition Example

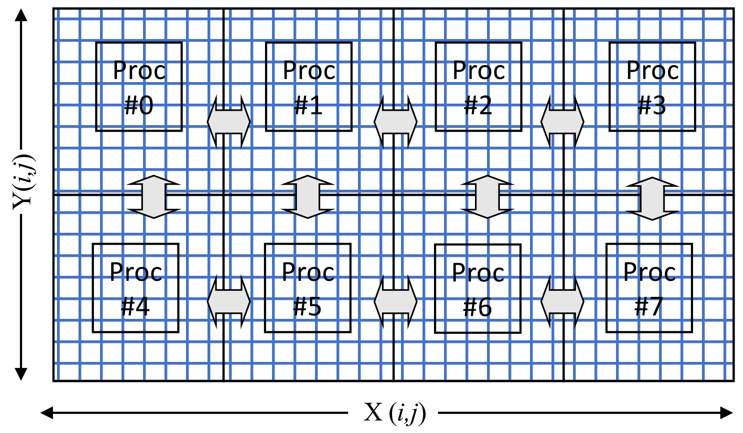
- EX) 2-D wave propagation problem

Starting partial differential equation:

$$\frac{\partial \Psi}{\partial t} = D \cdot \frac{\partial^2 \Psi}{\partial x^2} + B \cdot \frac{\partial^2 \Psi}{\partial y^2}$$

Finite Difference Approximation:

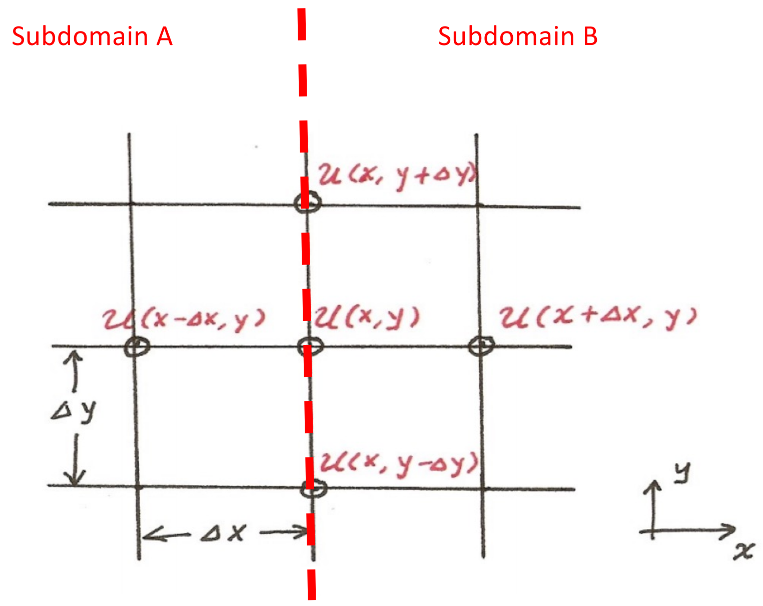
$$\frac{f_{i,j}^{n+1} - f_{i,j}^{n}}{\Delta t} = D \cdot \frac{f_{i+1,j}^{n} - 2 f_{i,j}^{n} + f_{i-1,j}^{n}}{\Delta x^2} + B \cdot \frac{f_{i,j+1}^{n} - 2 f_{i,j}^{n} + f_{i,j-1}^{n}}{\Delta y^2}$$



Numerical discretization of PDE

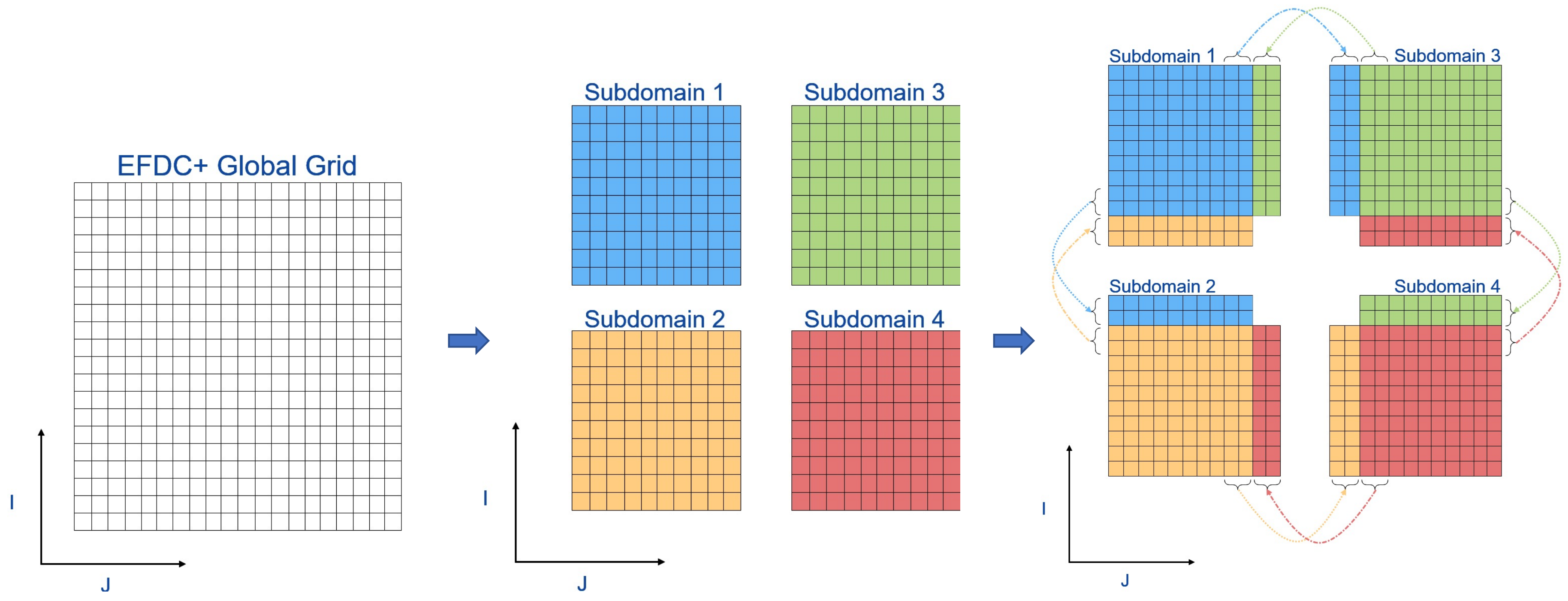
$$\left(\frac{\partial^2 u}{\partial x^2}\right) \approx \frac{u(x-\Delta x, y) - 2u(x,y) + u(x+\Delta x, y)}{(\Delta x)^2},$$

$$\left(\frac{\partial^2 u}{\partial y^2}\right) \approx \frac{u(x, y-\Delta y) - 2u(x,y) + u(x, y+\Delta y)}{(\Delta y)^2}$$



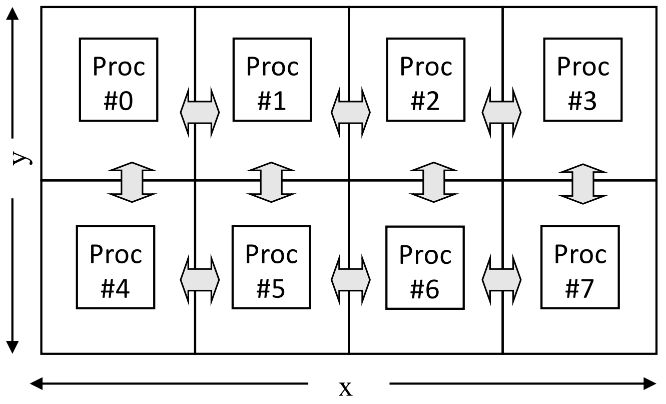
Implementation of Domain Decomposition

https://www.eemodelingsystem.com/efdc-insider-blog/domain-decomposition-details, by Nghiem Tien Lam

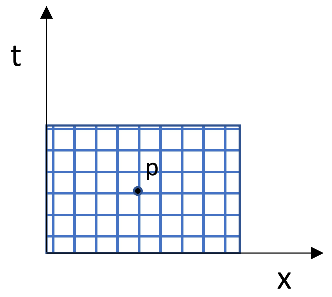


Information Propagation between Processes

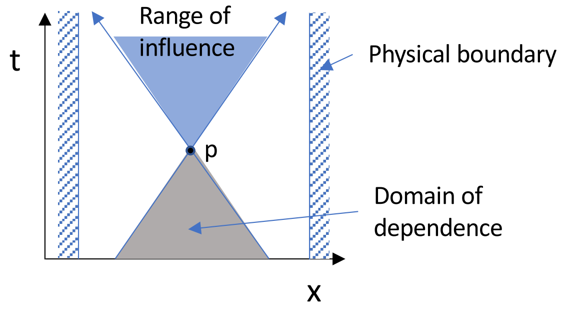
- How do we decide on what & how much of information should be passed along between decomposed computational domain?
  - based on PDE's *characteristics*



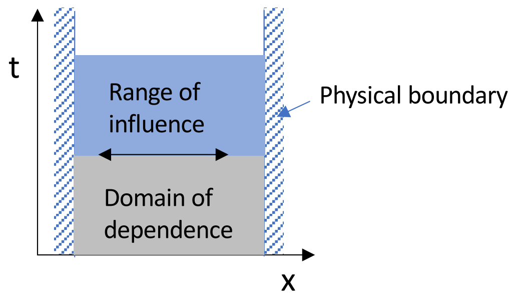
Elliptic PDE



Parabolic PDE



Hyperbolic PDE



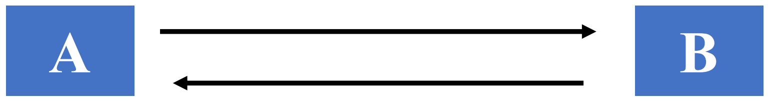
* Information propagation speed: Undefined Infinite Finite

MPI: Message Passing Interface

MPI: Message Passing Interface

- MPI 1 was released in 1994, MPI 2.0 in 1997, and MPI 3.0 in 2012
- Distributed memory programming
- Ideal for multi-node parallelization
- Can use with OpenMP for better scalability
- Distributed memory systems have separate address spaces for each processor
    - Local memory accessed faster than remote memory
    - Data must be manually decomposed
    - MPI is the standard for distributed memory programming

Message Passing Communication



- Processes in message passing program communicate by passing messages
- Basic message passing primitives
  - MPI_Send (parameters list)
  - MPI_Receive (parameter list)
- These calls are blocking: the source processor issuing the send/receive cannot move to the next statement until the target processor issues the matching receive/send.

Communicators

- MPI uses `MPI_Comm` objects to define subsets of processors which may communicate with one another.
- Two common functions for interacting with an `MPI_Comm` object are:
- `MPI_Comm_size(MPI_Comm_World, int *np)`
  - Gets the number of processes in a run, *NP*
- `MPI_Comm_rank(MPI_Comm_World, int *rank)`
  - Gets the rank of the current process
  - returns a value between 0 and *NP*-1 inclusive

- Both are typically called just after `MPI_Init`.

Sample MPI code (C)

```
#include <mpi.h>
 [other includes]

int main(int argc, char **argv){
   int ierr, np, rank;
    [other declarations]

   ierr = MPI_Init(&argc, &argv);
   ierr = MPI_Comm_size(MPI_COMM_WORLD, &np);
   ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
                :
    [actual work goes here]
                :
   MPI_Finalize();
}
```

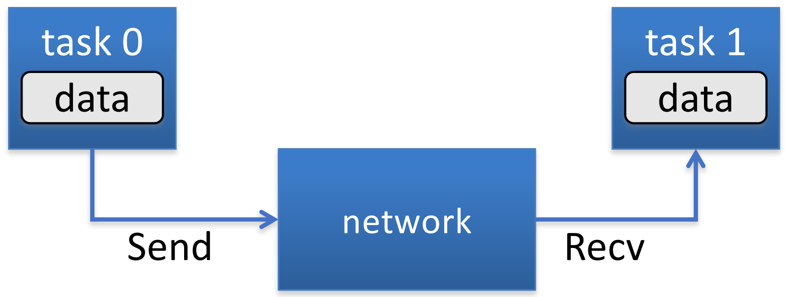
Sample MPI code (F90)

```
program sample-mpi
  use mpi
 [other includes]

  integer :: ierr, np, rank
     [other declarations]

  call mpi_init(ierr)
  call mpi_comm_size(MPI_COMM_WORLD, np, ierr)
  call mpi_comm_rank(MPI_COMM_WORLD, rank, ierr)
                :
         [actual work goes here]
                :
  call mpi_finalize(ierr)
end program
```

Point-to-Point Communication

- Sending data from one point (process/task) to another point (process/task)
- One task sends while another receives

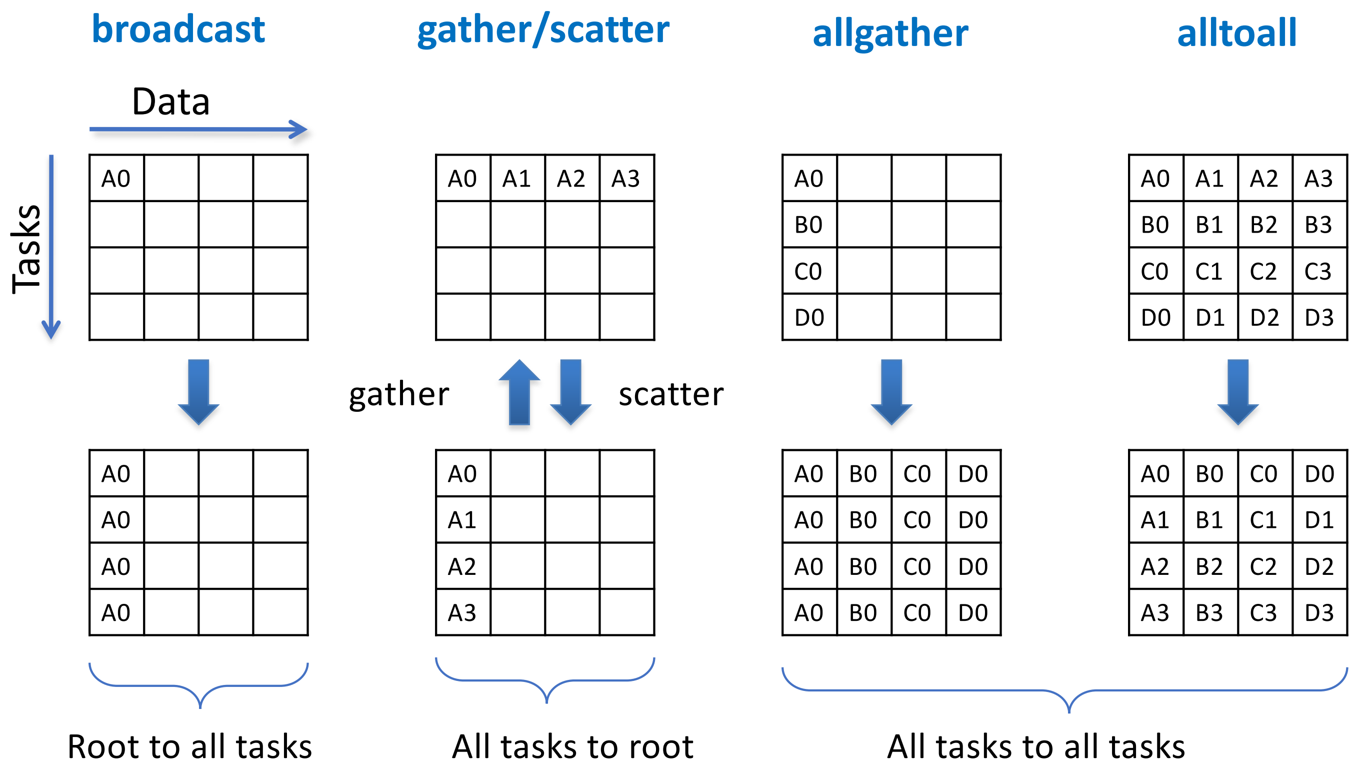
```
MPI_Send(buf, count, datatype, dest,   tag, comm);
MPI_Recv(buf, count, datatype, source, tag, comm, status);
```



Collective Communication

- Defined as communication between > 2 processors
  - One-to-many
  - Many-to-one
  - Many-to-many
- A collective operation requires that all processes within the communicator group call the *same* collective communication function with matching arguments.
- Type of collective operations
  - **Synchronization** (MPI_Barrier)
  - **Data Movement** (MPI_Bcast/Scatter/Gather/Allgather/AlltoAll)
  - **Computation** (MPI_Reduce/Allreduce/Scan)

Collective Communication Visualization



MPI Programming

USC | Advanced Research Computing
Enabling scientific breakthroughs at scale

MPI Programming: Basic Structure

Every MPI program needs these:

```
#include <mpi.h> /* the mpi include file */
int main(int argc, char *argv[])
{
  /* Initialize MPI */
  ierr = MPI_Init(&argc, &argv);
/* How many total processor are there */
  ierr = MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
/* What process am I (what is my rank? */
  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  ...
  ierr = MPI_Finalize();
```

MPI Example

```
#include
#include "mpi.h”

int main(int argc, char *argv[])
int argc;
char *argv[];
{
       int myid, nprocs;
       MPI_Init(&argc,&argv);
       MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
       MPI_Comm_rank(MPI_COMM_WORLD,&myid);
       /* print out my rank and this run’s NPROCS size*/
       printf("Hello from %d\n",myid," of ",nprocs);
       MPI_Finalize();
}
```

MPI Example: Send & Receive

```
#include "mpi.h"
/**************************************************************
This is a simple send/receive program in MPI
**************************************************************/
int main(int argc,char *argv[])
{
    int myid, numprocs, tag,source,destination,count,buffer ;
          MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=1234;
    source=0;
    destination=1;
    count=1;
    if(myid == source){
        buffer=5678;
        MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
        printf("processor %d  sent %d\n",myid, buffer);
    }
    if(myid == destination){
        MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
        printf("processor %d  got %d\n",myid, buffer);
    }
    MPI_Finalize();
}
```

Running MPI Jobs on Discovery

Compilers and MPI Stacks

- Available compilers on Discovery

Language	MPI Command (GCC, Intel)	Standard Command (GCC, Intel)
C	`mpicc, mpiicc`	`gcc, icc`
C++	`mpicxx, mpiicpc`	`g++, icpc`
Fortran 77/90	`mpif77/mpif90, mpiifort`	`gfortran, ifort`

- Available MPI stacks on Discovery: ***mpich, mvapich2, openmpi, intel-mpi***

- Run MPI jobs using Slurm's *srun* – use this within a Slurm job submission script

```
srun --mpi=pmix_v2 -n $SLURM_NTASKS ./mpi_program.x
```

← openmpi

```
srun --mpi=pmi2 -n $SLURM_NTASKS ./mpi_program.x
```

← mpich, mvapich2

MPI Job Slurm Scripts

```
#!/bin/bash

#SBATCH --nodes=6
#SBATCH --ntasks=144
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=3GB
#SBATCH --time=24:00:00
#SBATCH --constraint="xeon-4116"
#SBATCH --exclusive
#SBATCH --account=<account_id>

module purge
module load gcc/8.3.0
module load openmpi/4.0.2
module load pmix/3.1.3

ulimit -s unlimited

srun --mpi=pmix_v2 -n $SLURM_NTASKS ./mpi_program.x
```

144/6 = 24 (minimum number of cores available on each node

Total number of process = total number of cores required (cpu-per-task=1)
This is the value assigned to $SLURM_NTASKS, $SLRUM_NPROCS

A single core will be assigned to a single process

24 x 3GB = 72GB (minimum memory size required per node)

Asking xeon-4116 nodes only: is this the right choice? – check w/ *sinfo*

Not allowing other jobs to run on the selected nodes

You account ID

Default compiler & mpi-stack in ‘usc’ module group

$SLURM_NTASKS = 144

pmix_v2 is required for the use of openmpi

Hands-on session w/ examples