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

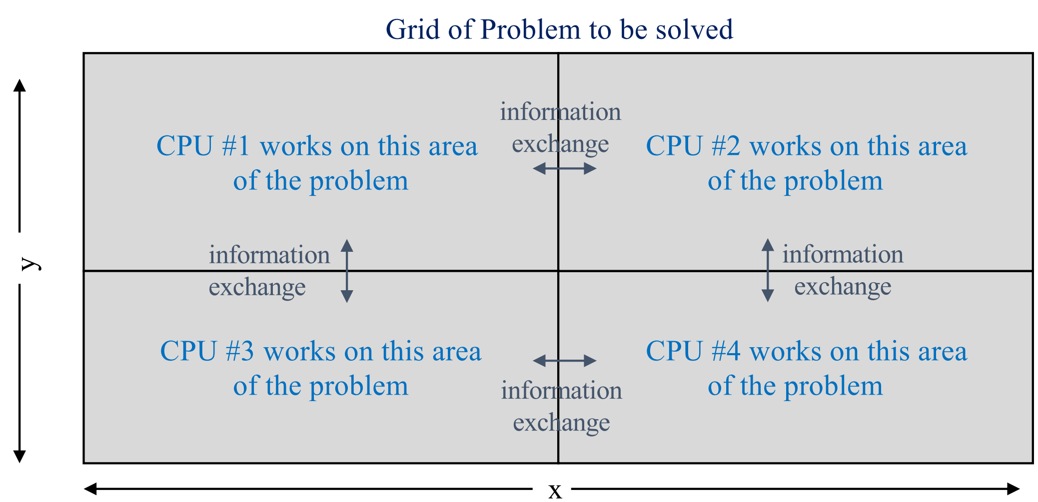
1. Introduction
2. Theoretical background
3. Types of parallel computing systems
4. Programming models
5. Examples
6. Hands-on session

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

What is Parallel Computing?

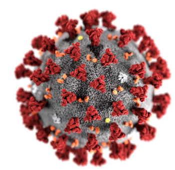
- Parallel computing: use of multiple processors or computers working together on a common task.
  - Each processor works on its section of the problem
  - Processors can exchange information



Supercomputing Applications

- Black Hole Simulation
  - NASA article
  - Black Hole simulation
- Formular 1 Racing Car Aerodynamics:
  - F1 aerodynamics
  - Winner of Baku Grand Prix, Flex wing debate

Formula 1

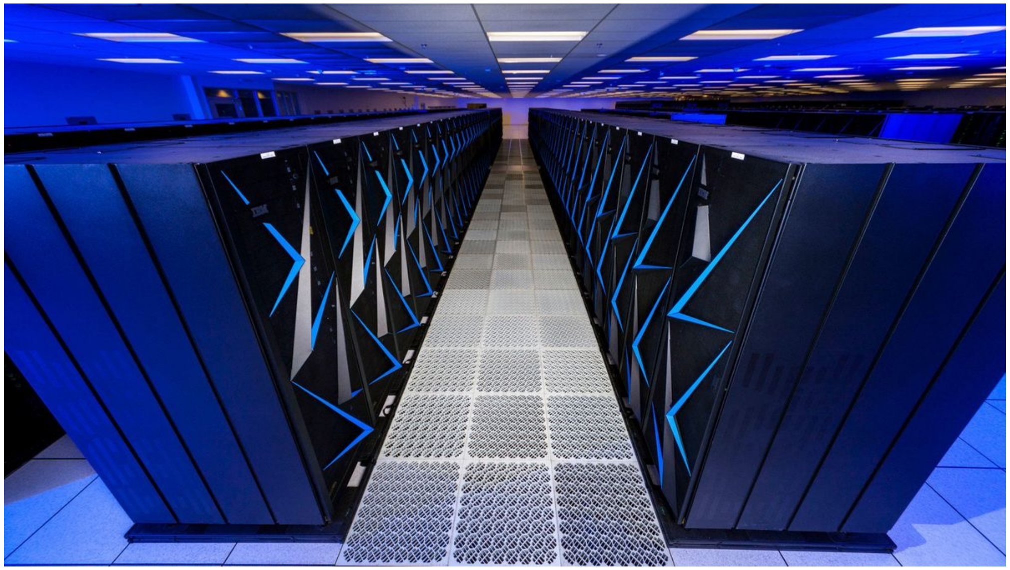


- Supercomputing vs. COVID-19
  - Drug discovery for COVID-19 using supercomputer
- AI & Supercomputer
  - DeepMind AlphaGo defeated professional Go player 4:1
  - AlphaGo Movie
- Stock Market High-Frequency Trading

AlphaGo

Top 500

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



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

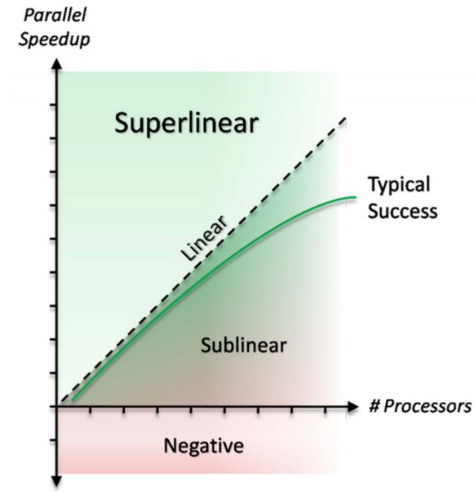
Theoretical Upper Limit to Performance

- 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: $S = \frac{1}{f_s + \frac{f_p}{N}}$

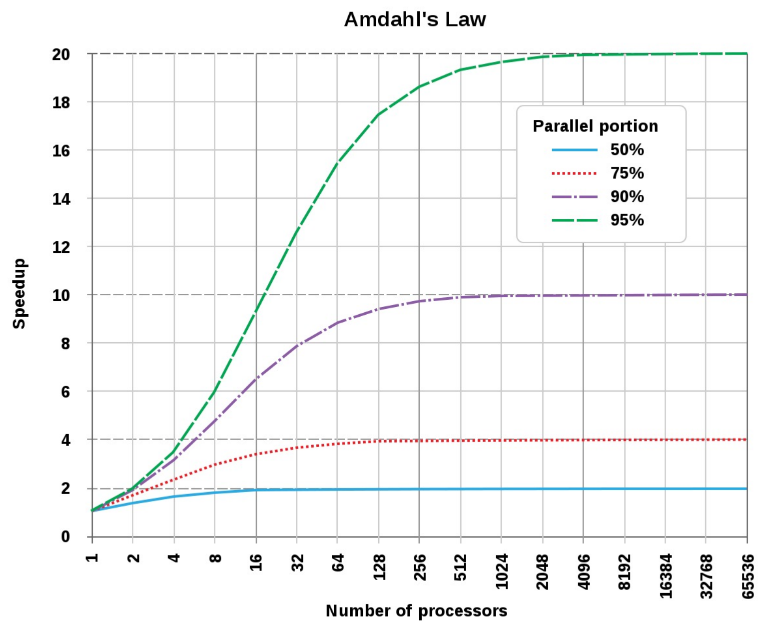
    where
    - f_s = serial fraction of code
    - f_p = parallel fraction of code
    - N = number of processors



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

Illustration of Amdahl's Law

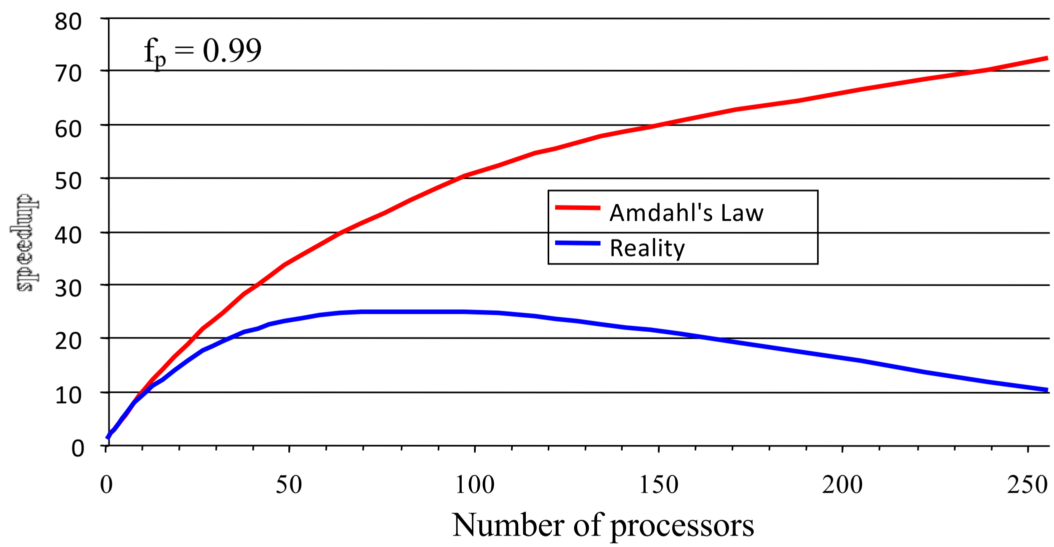
- It takes only a small fraction of serial content in a code to degrade the parallel performance.



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.



Practical Limit: Amdahl's Law vs. Reality

- In reality, the situation is even worse than predicted by Amdahl's Law due to:
  - Load balancing (waiting)
  - Scheduling (shared processors or memory)
  - Communications
  - I/O

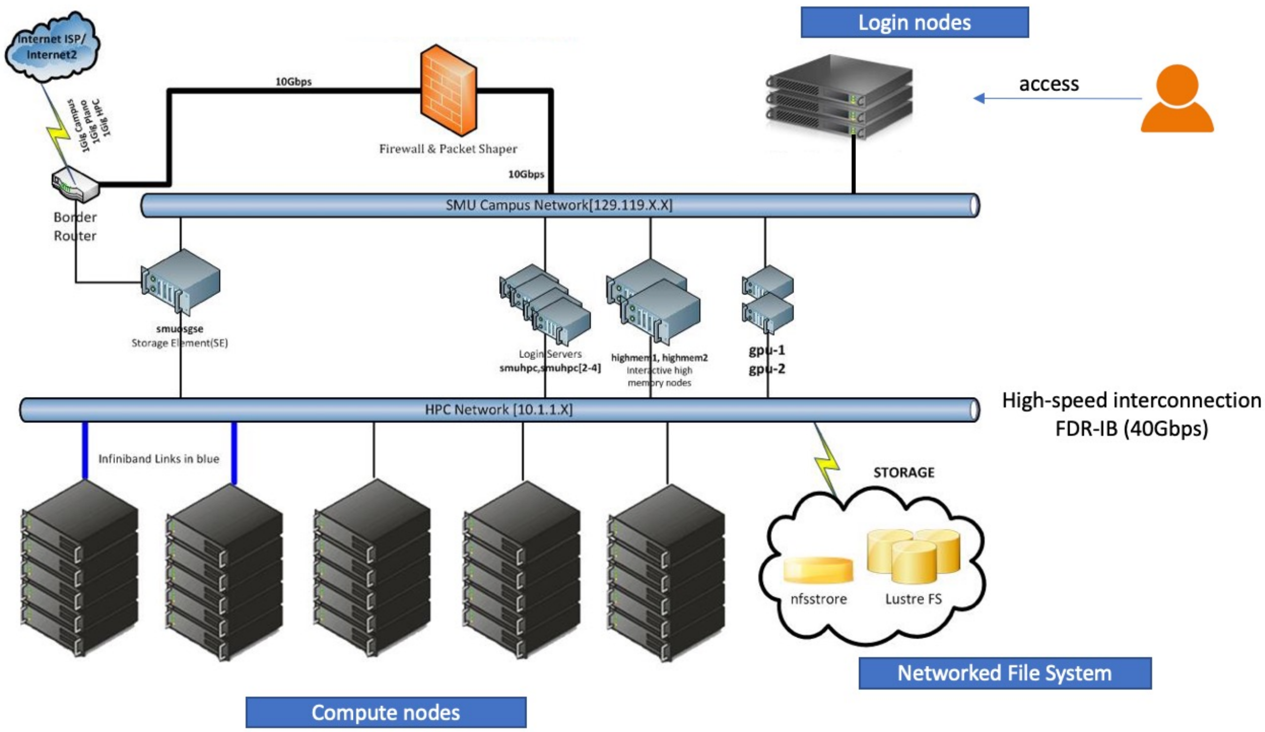
Other Considerations

- 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 Computing Models

- Parallel System Architecture
- Parallel Programming Models
- MPI Programming Basics

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.

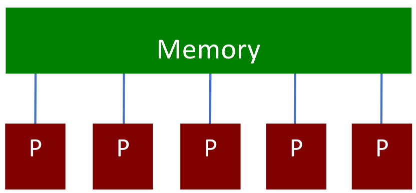
System OS, File System, Provisioning, Config Mgmt

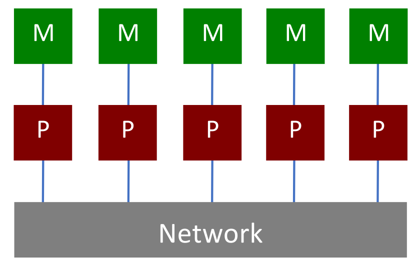
Cluster, Storage, Network

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.

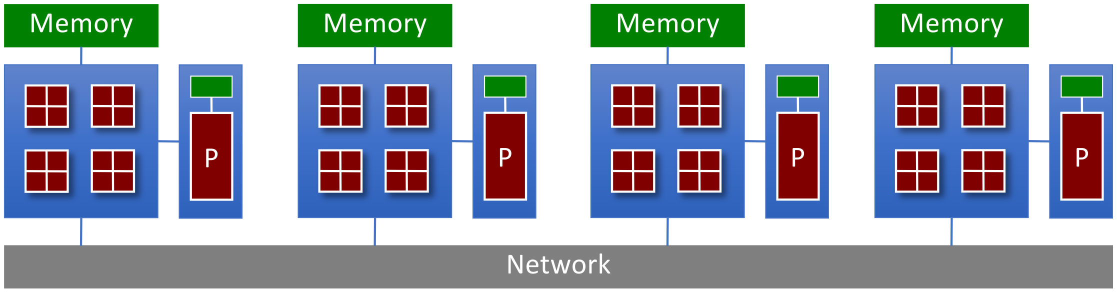
- Methods of memory access :
  - Bus, Crossbar
- Programming model: OpenMP

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
  - Main memory access
  - Quick Path / Hyper Transport socket connections
  - Node to node connection via network
- Load balancing critical for performance
- Requires specific libraries and compilers (CUDA, OpenCL, ACC, etc.)

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

CPU A

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

CPU B

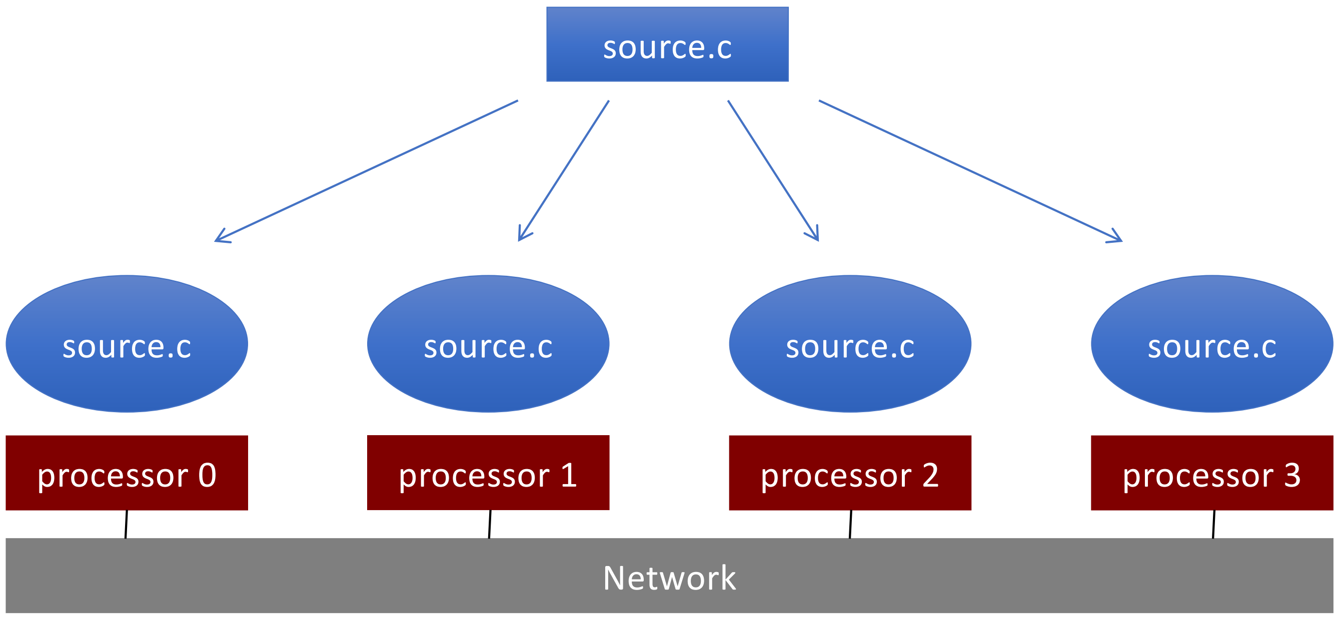
```
program.c:
…
initialize
…
do task b
…
end program
```

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

- MPMD: more general, and possible in hardware, but no system/programming software enables it

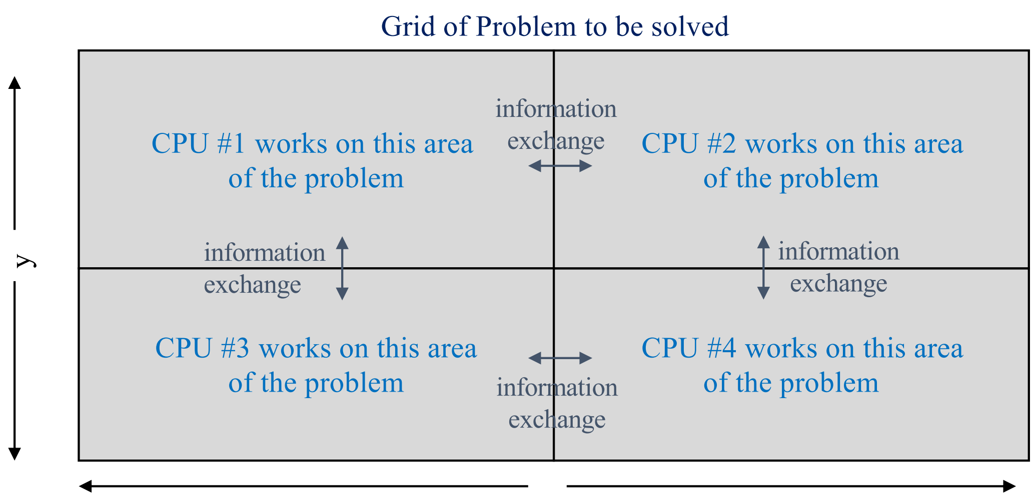
SPMD Model



- 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



Domain Decomposition Example

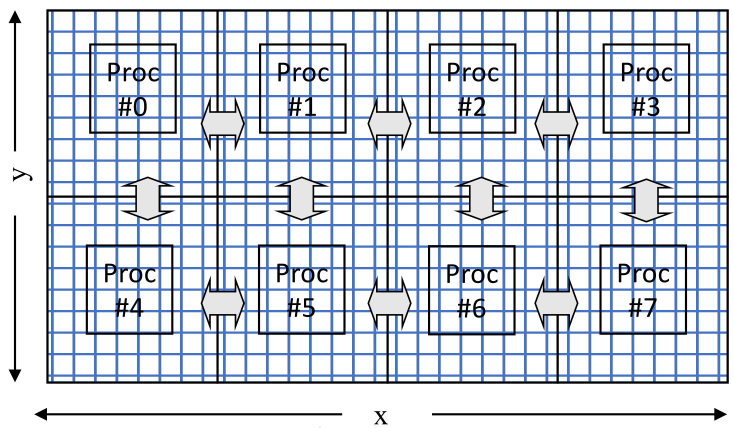
- Integrated 2-D wave propagation problem

Starting partial differential equation:

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

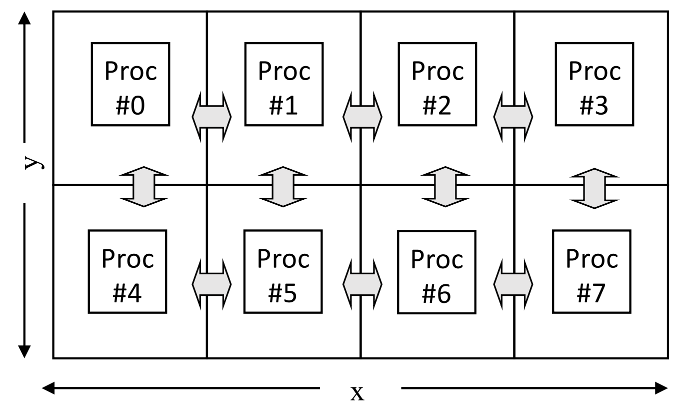
Finite Difference Approximation:

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

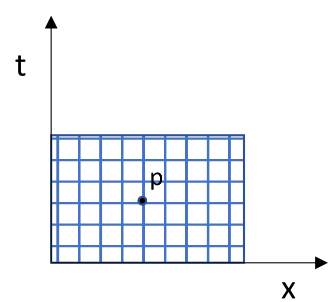


Information Propagation between Processes

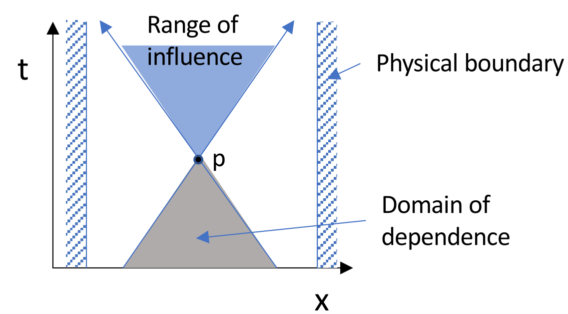
- How do we decide on what & how much of information should be passed along between decomposed computational domain?
  - based on PDE's *characteristics*



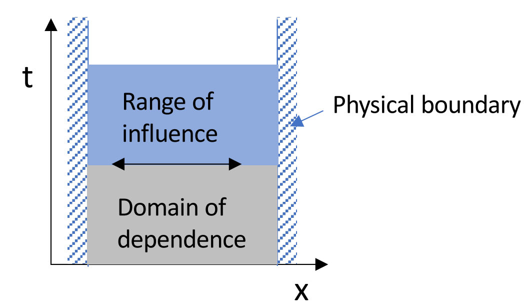
Elliptic PDE



Parabolic PDE



Hyperbolic PDE



* Information propagation speed: Undefined Infinite Finite

MPI: Message Passing Interface

- MPI 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

MPI Programming: Basic Structure

Every MPI program needs these:

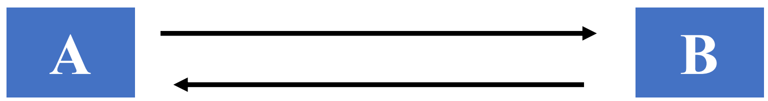
```
#include <mpi.h> /* the mpi include file */
int main(int argc, char *argv[])
{
  /* Initialize MPI */
  ierr = MPI_Init(&argc, &argv);
/* How many total PEs are there */
  ierr = MPI_Comm_size(MPI_COMM_WORLD, &nPEs);
/* What node am I (what is my rank? */
  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &iam);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

MPI Example

```
#include
#include "mpi.h”

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

Message Passing Communication

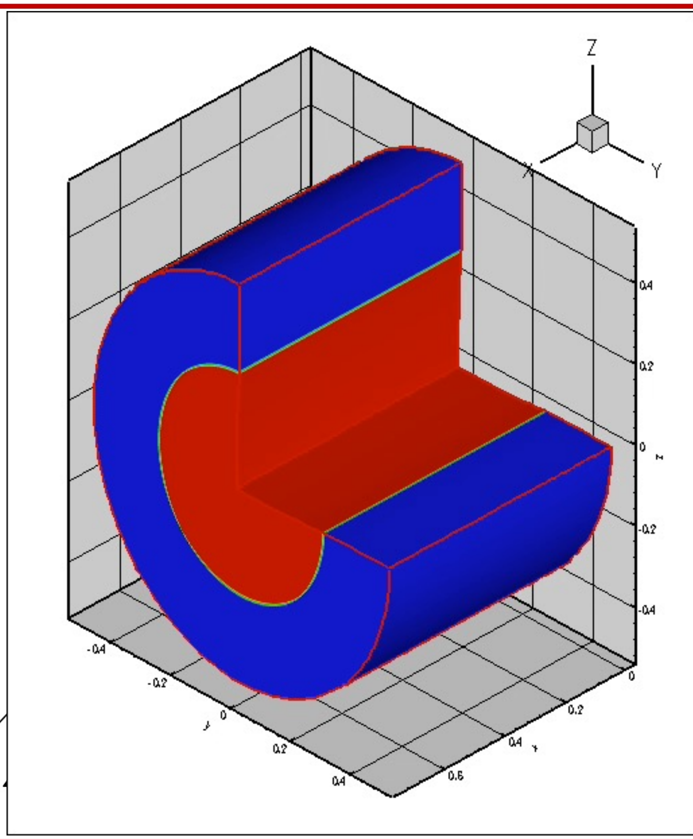


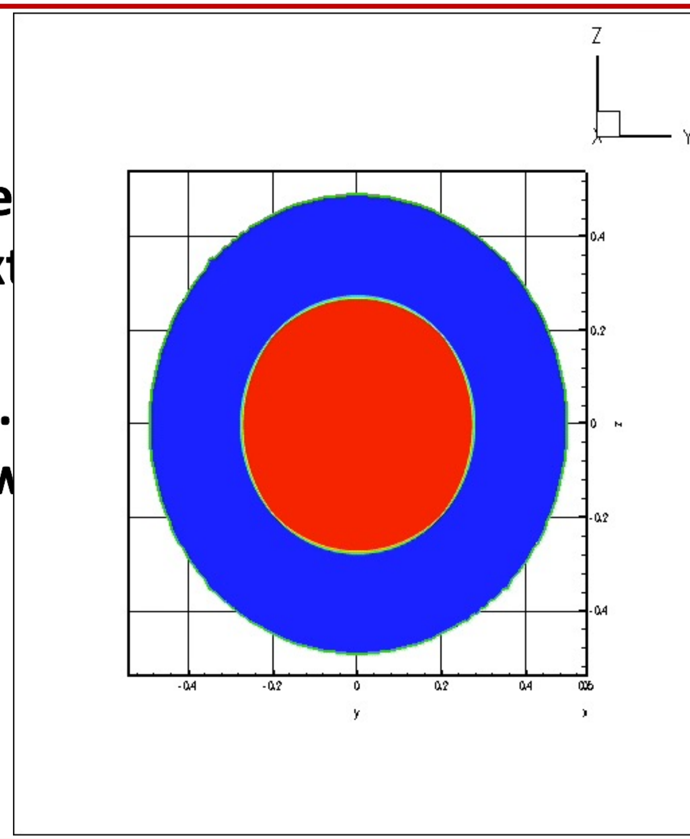
- Processes in message passing program communicate by passing messages
- Basic message passing primitives
  - MPI_Send (parameters list)
  - MPI_Receive (parameter list)
- These calls are blocking: the source processor issuing the send/receive cannot move to the next statement until the target processor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

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();
}
```

Quick Overview of Parallel Computing





Hands-on session w/ examples

- Hello World
- Pi-calculation

- MPICH vs OpenMPI – different MPI stack
- Compilers – Intel vs gcc
- MKL vs OPENBLAS/LAPACK/SCALAPACK