

INTRODUCTION TO PARALLEL COMPUTING

*Evan Schneider**

**with significant inspiration from Lawrence Livermore's parallel computing tutorial*

WHERE TO LEARN MORE

Online resources:

This tutorial: https://computing.llnl.gov/tutorials/parallel_comp/

Summer schools:

Argonne Training Program on Extreme Scale Computing (ATPESC):
<https://extremecomputingtraining.anl.gov>

International HPC Summer School: <http://www.ihpcss.org>

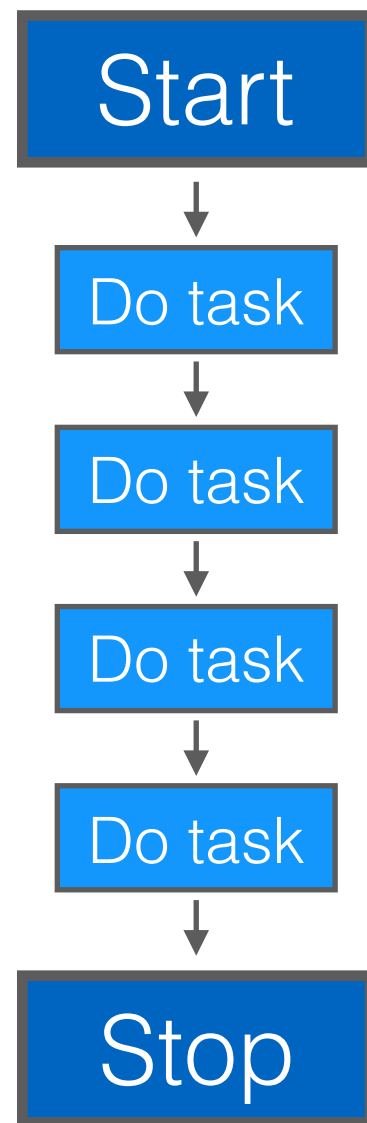
Hack weeks:

Oak Ridge GPU Hackathons:
<https://www.olcf.ornl.gov/for-users/training/gpu-hackathons/>

And more!

WHAT IS PARALLEL COMPUTING?

Serial Approach



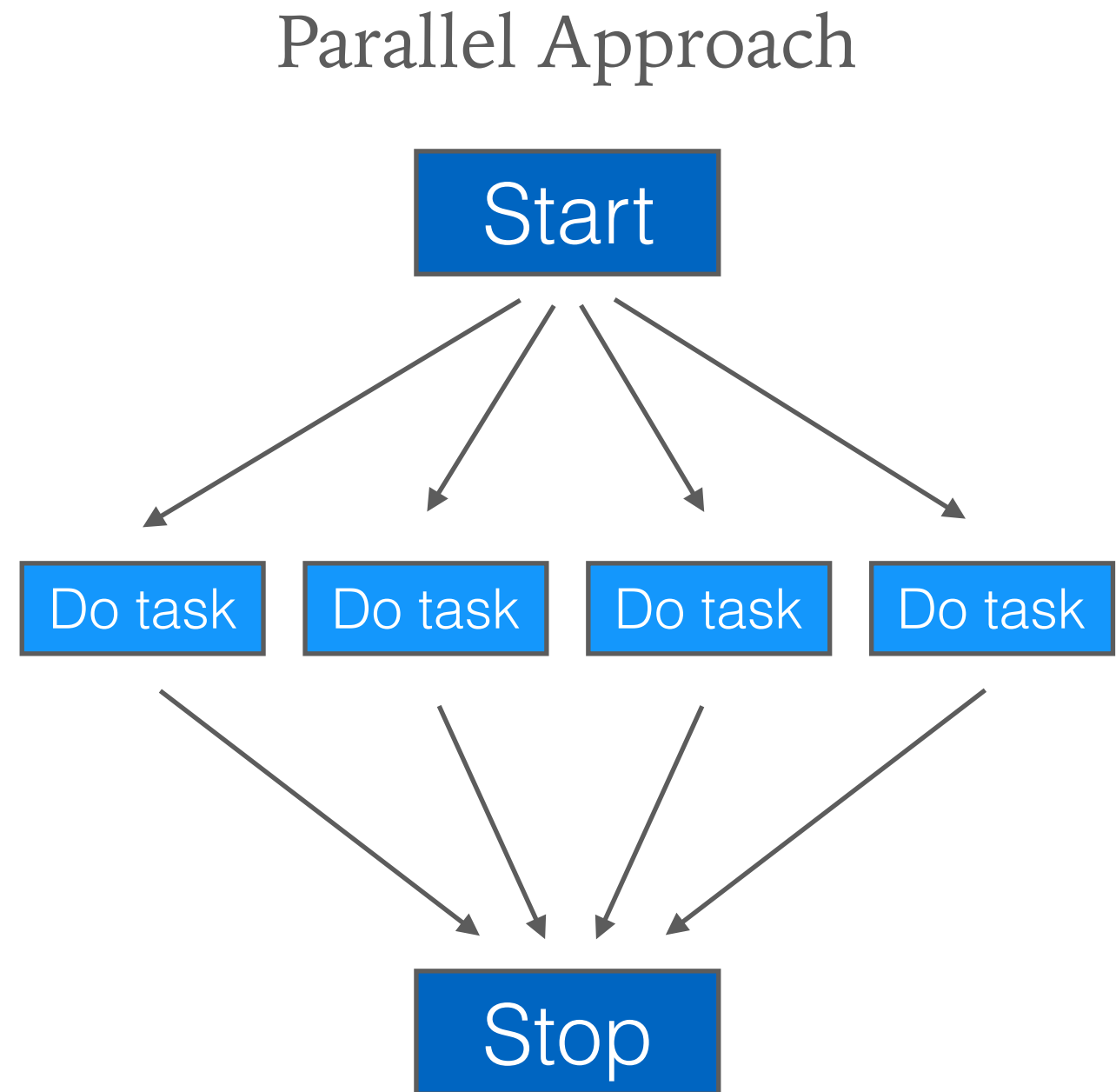
Traditionally, software has been written for serial computation:

- A problem is broken down into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time

WHAT IS PARALLEL COMPUTING?

Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

- A problem is broken down into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed



WHEN TO USE PARALLEL COMPUTING

The computational problem should be able to:

- Be broken apart into discrete pieces of work that can be solved simultaneously
- Execute multiple program instructions at any moment in time
- Be solved in less time with multiple compute resources than with a single compute resource

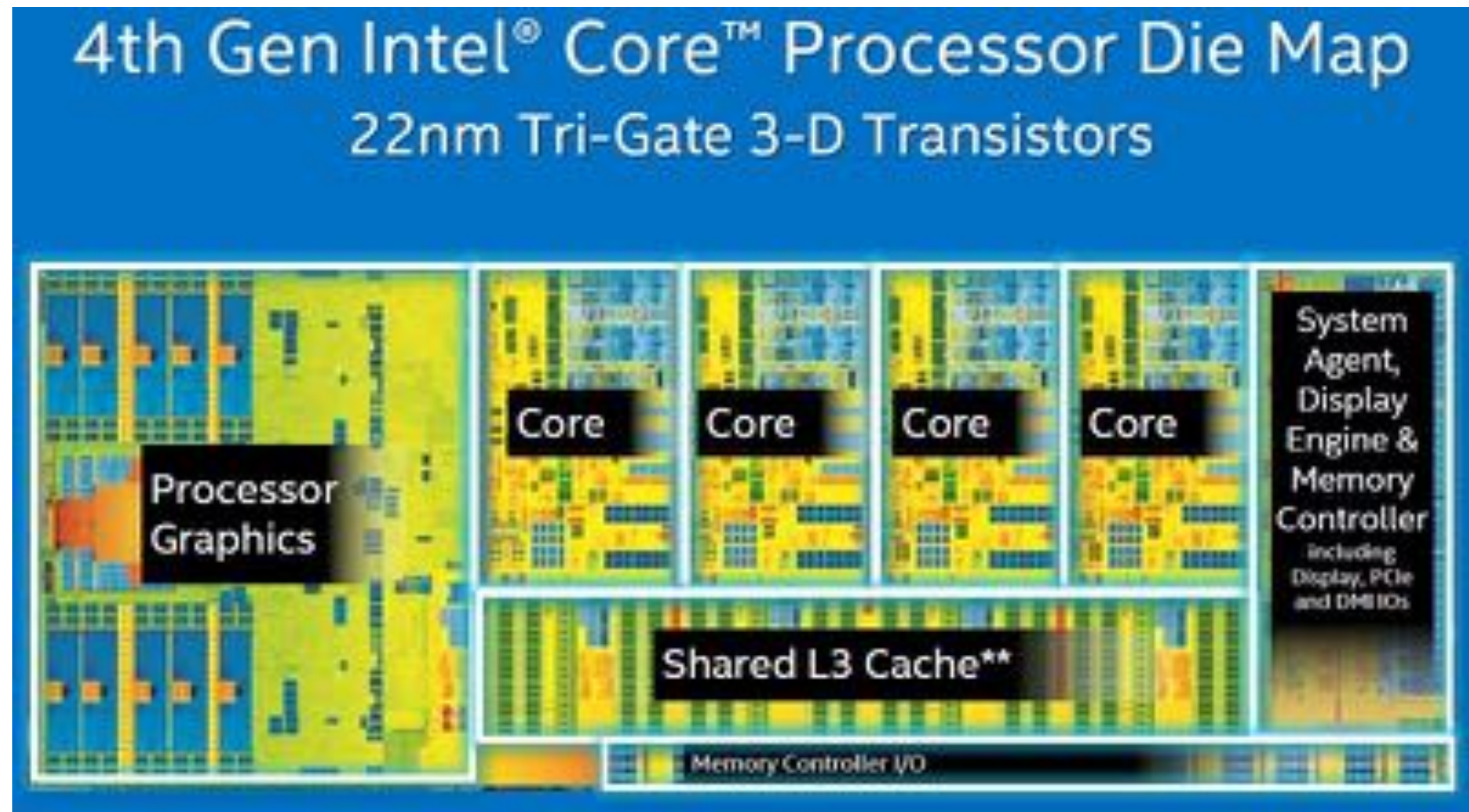
The compute resources are typically:

- A single computer with multiple processors/cores
- An arbitrary number of such computers connected by a network

WHY USE PARALLEL COMPUTING?

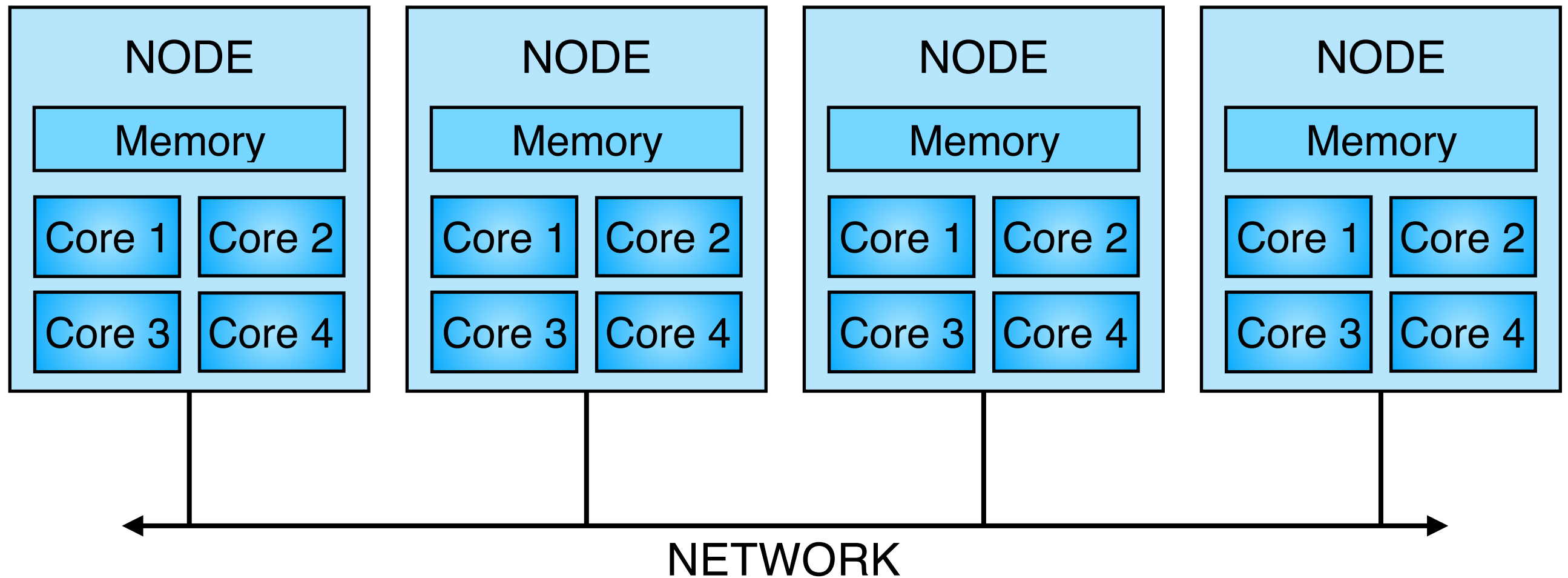
.....

Practically speaking, all computers today are parallel.



WHY USE PARALLEL COMPUTING?

*Networks connect multiple stand-alone computers (**nodes**) to make larger parallel computer clusters.*



WHY USE PARALLEL COMPUTING?

.....

The largest computing resources today (supercomputers) are massively parallel systems.



 compute node

 infiniband switch

 management hardware



login / remote partition server node



gateway node

WHY USE PARALLEL COMPUTING?

- The real world is massively parallel.
 - Many problems can be tackled in a parallel way - climate modeling, web searches, astrophysical hydrodynamics!
- Save time / money
 - Should be able to shorten time to complete problems
 - Can build bigger computers with commodity hardware
- Solve larger / more complex problems
- Provide concurrency
- Take advantage of non-local resources (ex. Zooniverse)
- Make better use of modern parallel hardware

WHY USE PARALLEL COMPUTING?

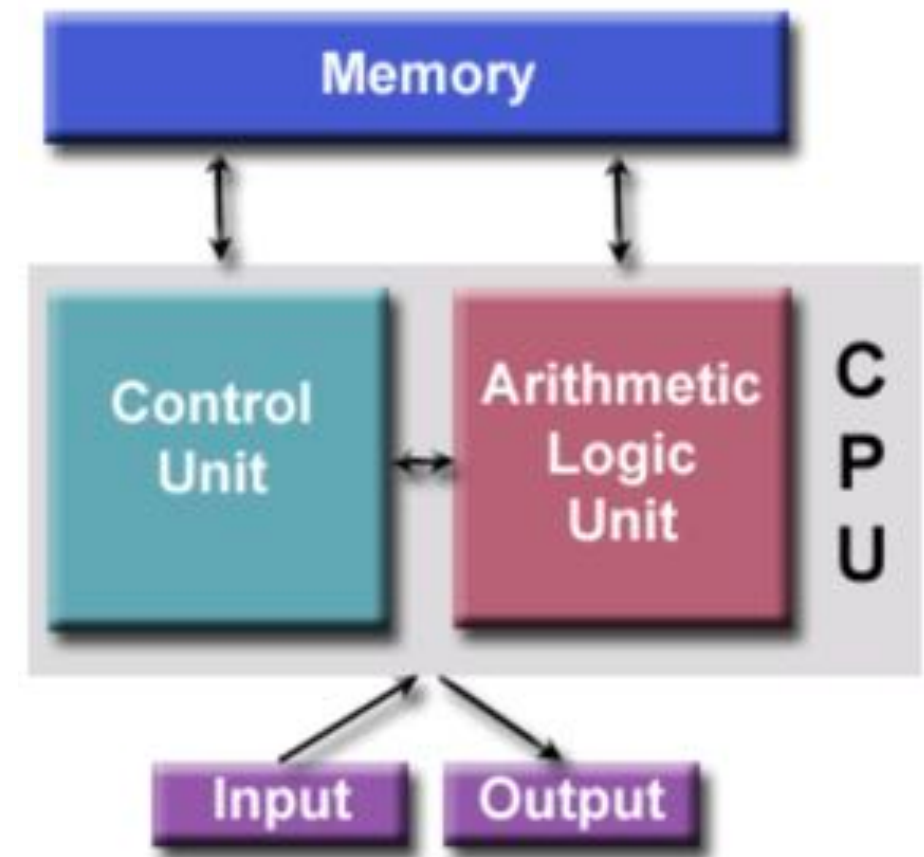
.....

The future is parallel.

VON NEUMANN ARCHITECTURE

Computers comprised of four main components:

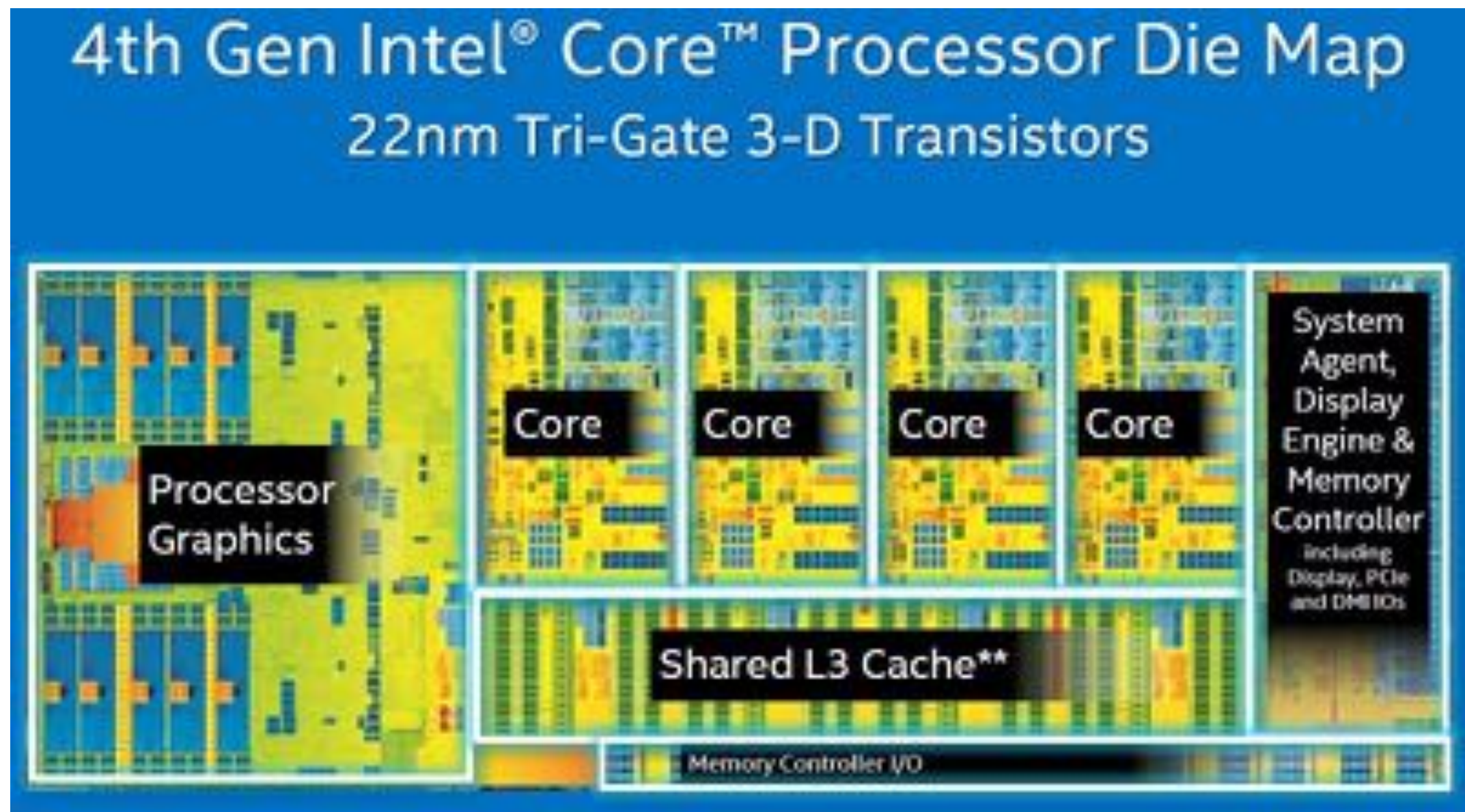
- Memory
- Control Unit
- Arithmetic Logic Unit
- Input / Output



SOME TERMINOLOGY

.....

CPU (Processor/Socket): Central Processing Unit; the primary component of a computer that processes instructions; can contain multiple cores, each of which can run a single program context (or multiple via hardware threading...)

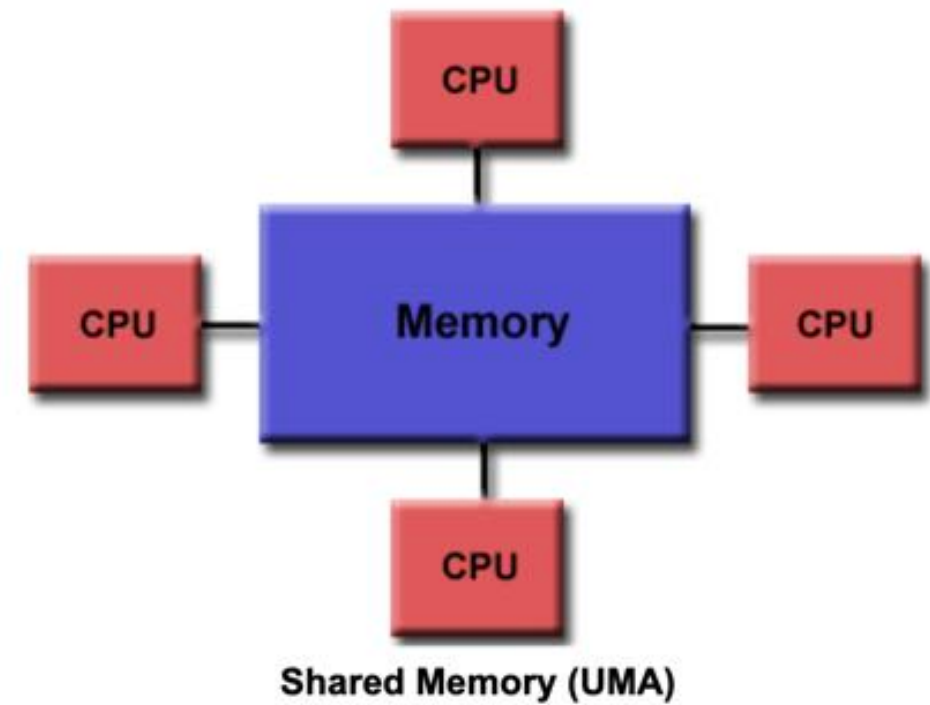


SOME TERMINOLOGY

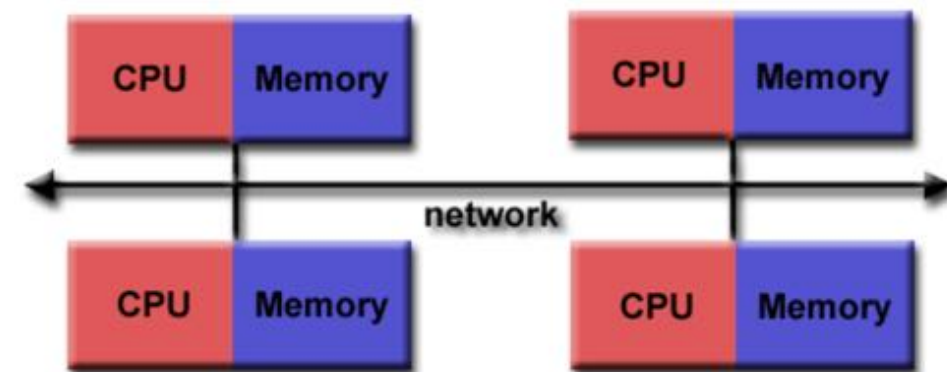
Task: A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.

SOME TERMINOLOGY

***Shared Memory:** From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists*



***Distributed Memory:** In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.*



SOME TERMINOLOGY

Communications: *Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.*

Synchronization: *The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.*

SOME TERMINOLOGY

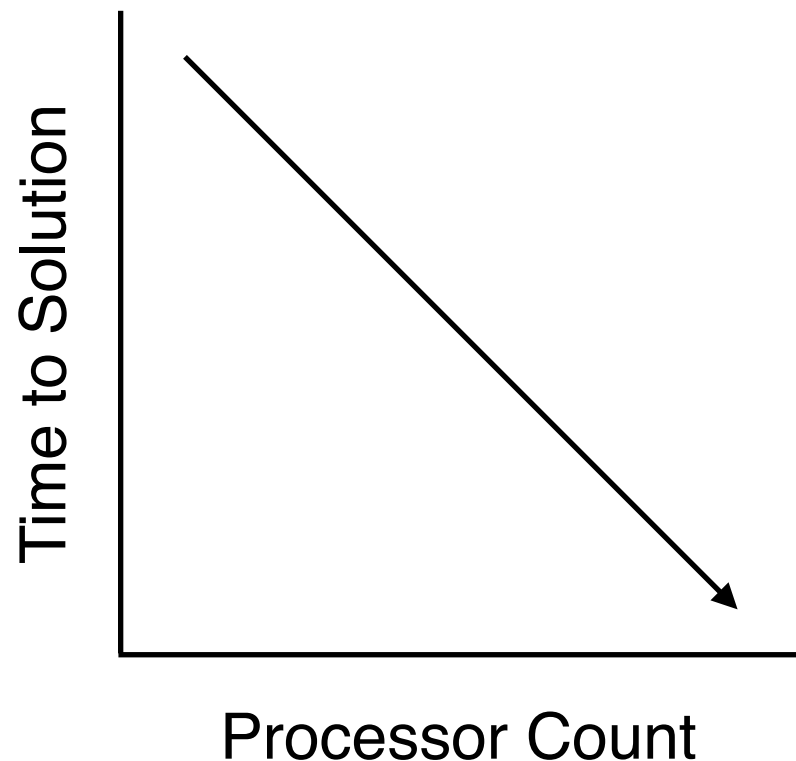
Observed Speedup: Observed speedup of a code which has been parallelized, defined as:

$$\frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$$

SOME TERMINOLOGY

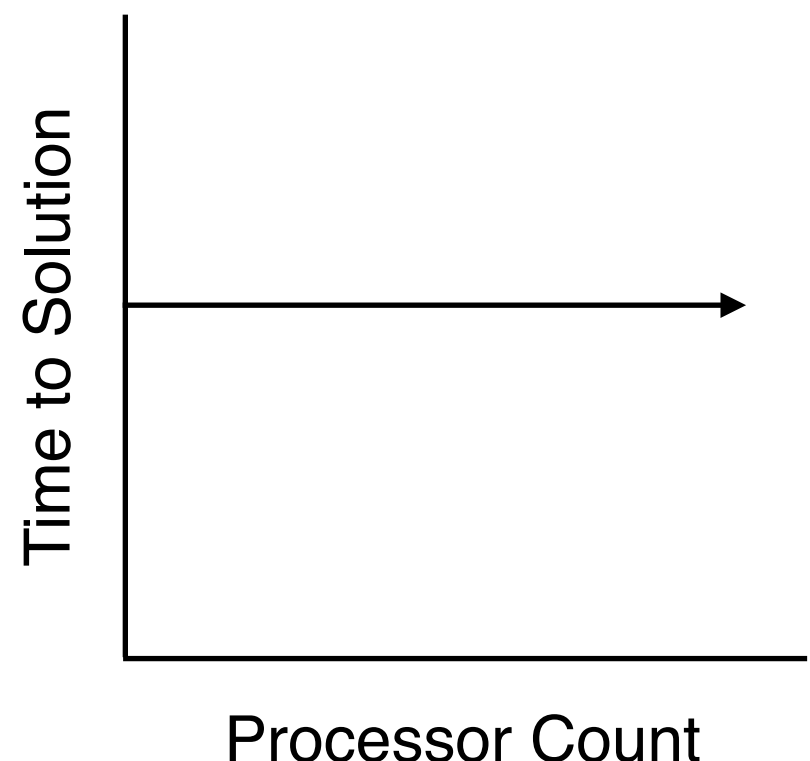
Scalability: Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources.

Strong Scaling



Problem size remains the same, resources increase - time to solution goes down.

Weak Scaling



Problem size increases linearly with resources - time to solution remains the same.

PARALLEL PROGRAMMING MODELS

Flynn's Taxonomy:

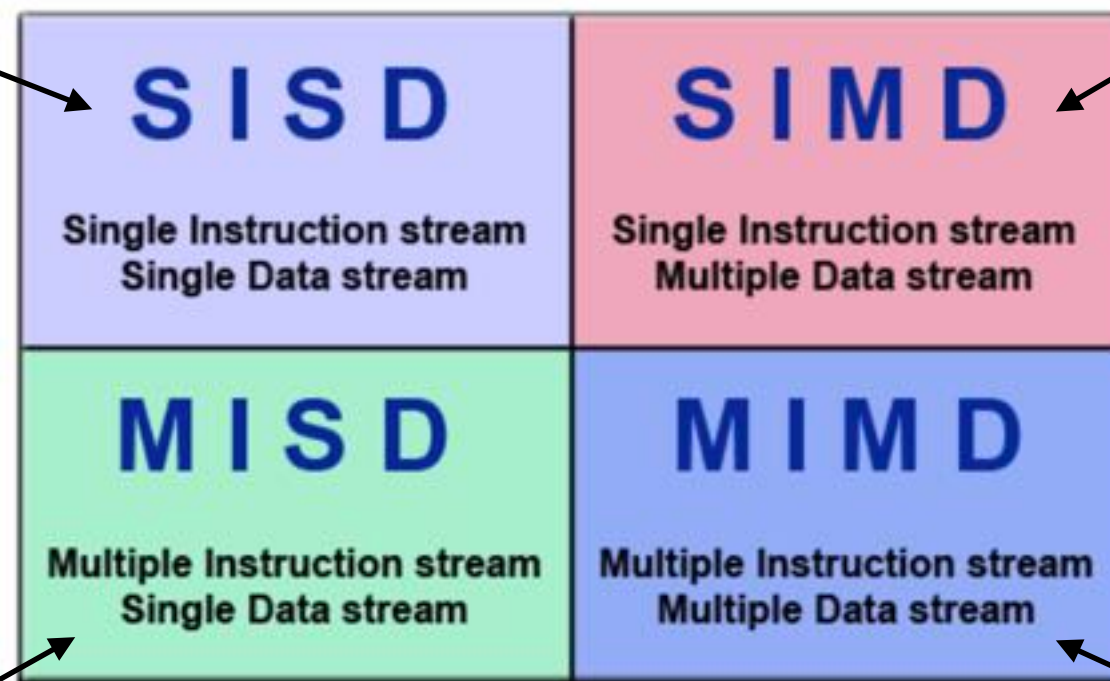
A serial computer

The oldest kind of architecture

A type of parallel computer

Best suited to highly regular problems

Think graphics processing



A type of parallel computer

Very few exist

A type of parallel computer

The most common version

Think multi-core laptops

PARALLEL PROGRAMMING MODELS

Parallel programming models exist as an abstraction above hardware and memory architectures.

- Shared Memory Model (without threads)
- Threads
 - Threads execute subroutines of the main program
- Distributed Memory / Message Passing Model
 - Message Passing Interface (MPI) is the de facto standard
- Hybrid models
 - E.g. Using MPI to communicate between CPUs, while threads perform computationally intensive tasks with local data on the CPU or GPU

DESIGNING PARALLEL PROGRAMS

Automatic vs. Manual Parallelization

- Automatic: Parallelizing compiler
 - Fully automatic: compiler analyzes source code and identifies opportunities for parallelism
 - Programmer directed: using compiler directives or flags to identify regions of parallelism in code (e.g. Numba vectorize)
 - Must be careful with both!
- Manual: programmer codes explicitly parallel (e.g. writing a CUDA program)

DESIGNING PARALLEL PROGRAMS

First step: Understand the problem. Determine whether the problem can be parallelized.

- Example of an inherently parallel problem: Calculate the potential energy for several thousand independent conformations of a molecule. When done, find the minimum energy conformation.
- Example of an inherently serial problem: Calculate the Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13...) using the formula $F(n) = F(n-1) + F(n-2)$.

DESIGNING PARALLEL PROGRAMS

- Identify a program's hotspots
 - Where is the work being done? Use profilers to help with this. Focus on parallelizing the hotspots.
- Identify a program's bottlenecks
 - Are there areas that are disproportionately slow? Ex. I/O often slows a program down.
- Identify inhibitors to parallelism (e.g. data dependence as demonstrated in the Fibonacci example)
- Investigate other algorithms if possible!
- Take advantage of third party software and libraries (e.g. many highly optimized math libraries already exist)

AMDAHL'S LAW

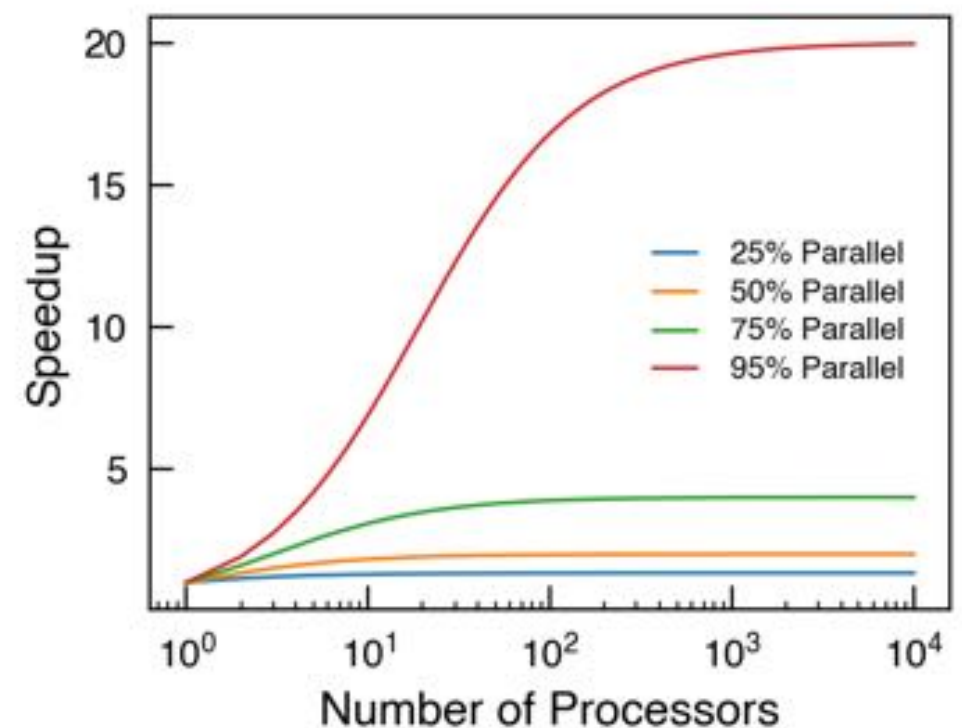
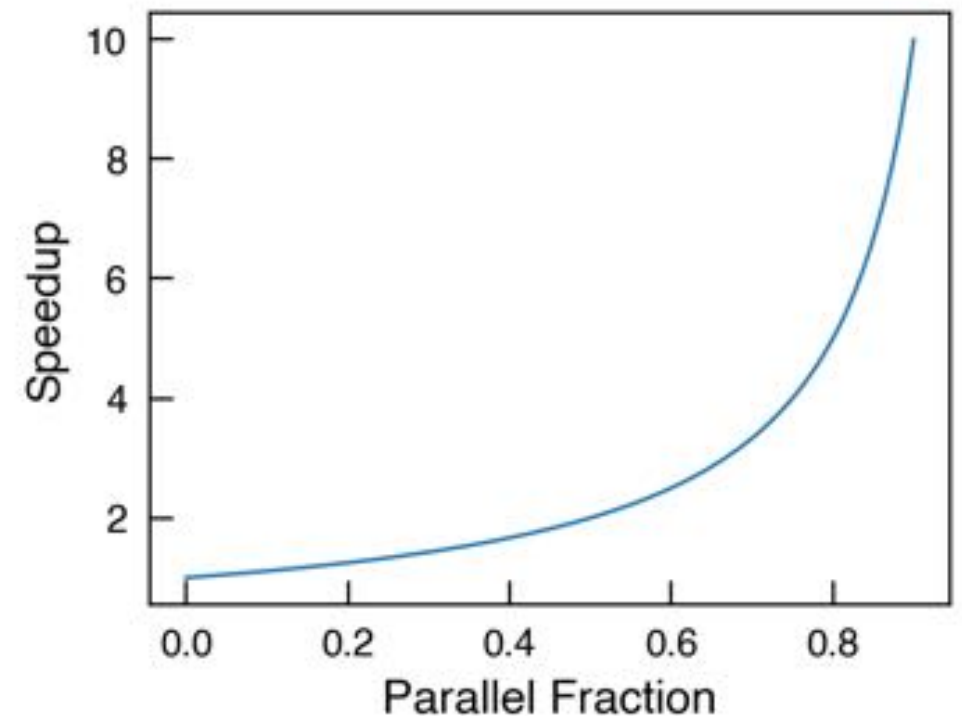
Amdahl's law states that the potential program speedup is defined by the fraction of code that can be parallelized (P):

$$\text{speedup} = \frac{1}{1 - P}$$

Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled as:

$$\text{speedup} = \frac{1}{\frac{P}{N} + S}$$

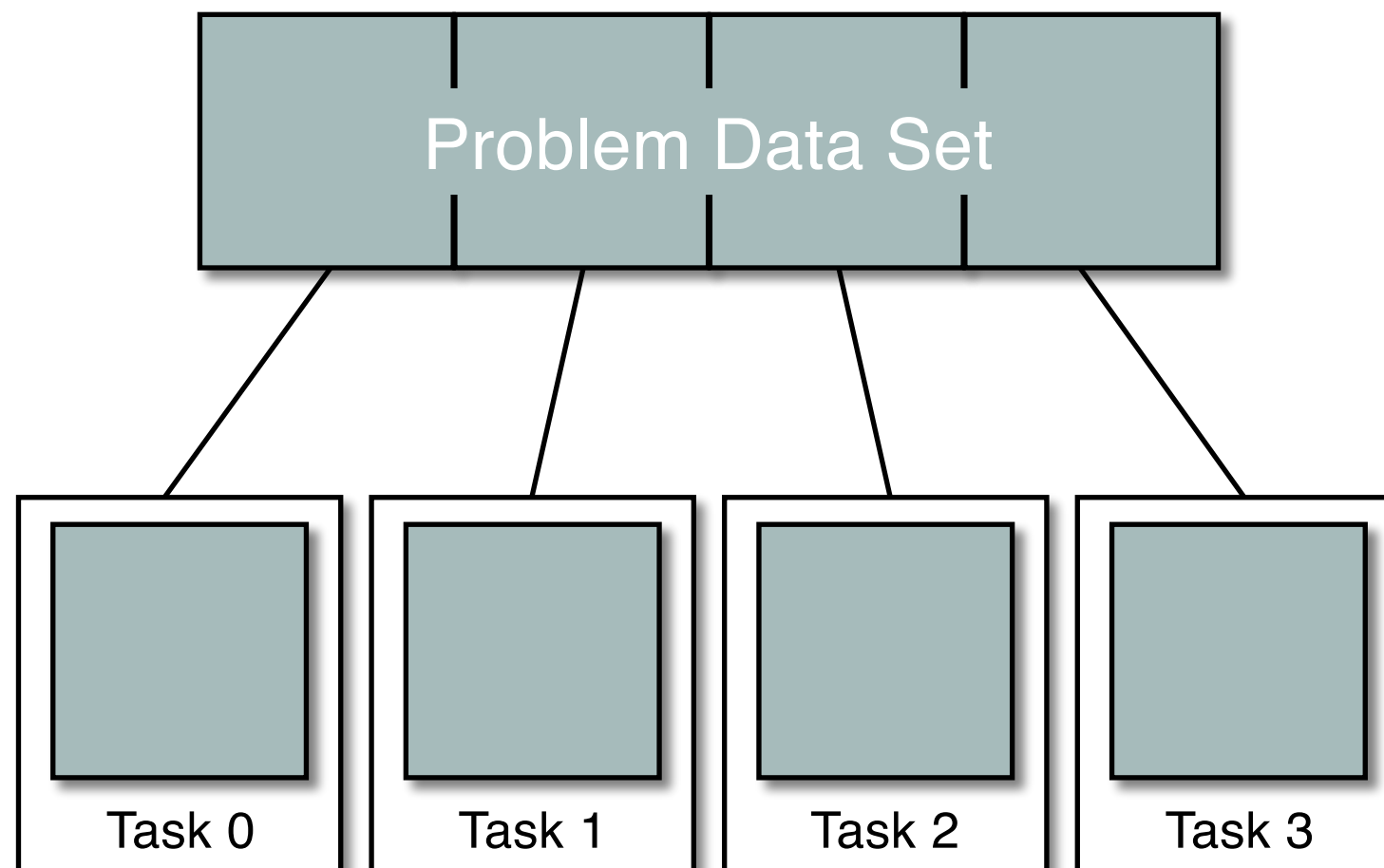
where N is the number of processors, P is the parallel fraction, and S is the serial fraction.



DESIGNING PARALLEL PROGRAMS – PARTITIONING

Partitioning: Break the problem down into discrete “chunks” of work that can be distributed to multiple tasks.

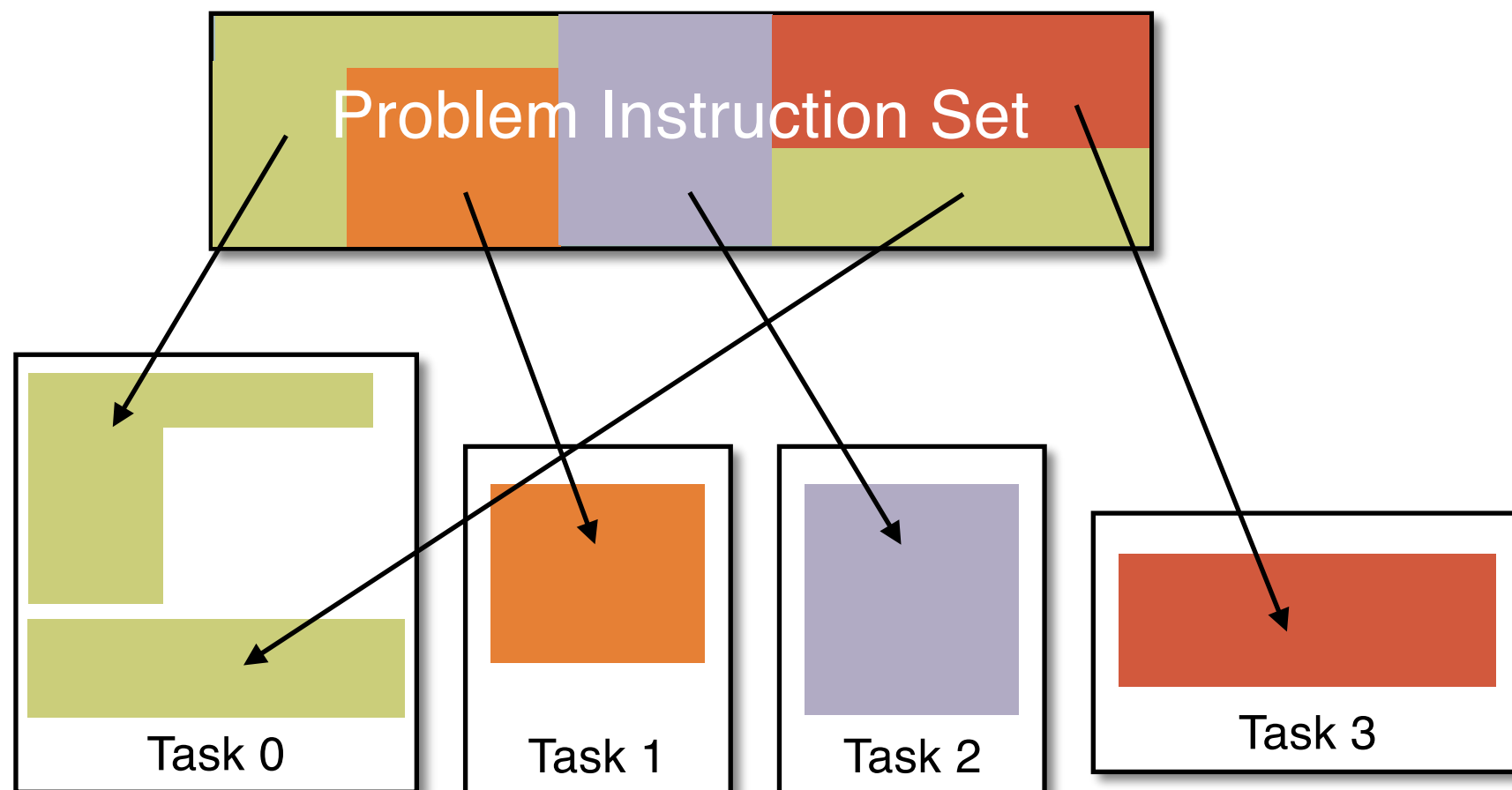
- Domain decomposition: Data associated with a problem is decomposed. Each parallel task works on a portion of the data.



DESIGNING PARALLEL PROGRAMS – PARTITIONING

Partitioning: Break the problem down into discrete “chunks” of work that can be distributed to multiple tasks.

- Functional decomposition: Problem is decomposed according to the work that must be done. Each task performs a portion of the overall work.

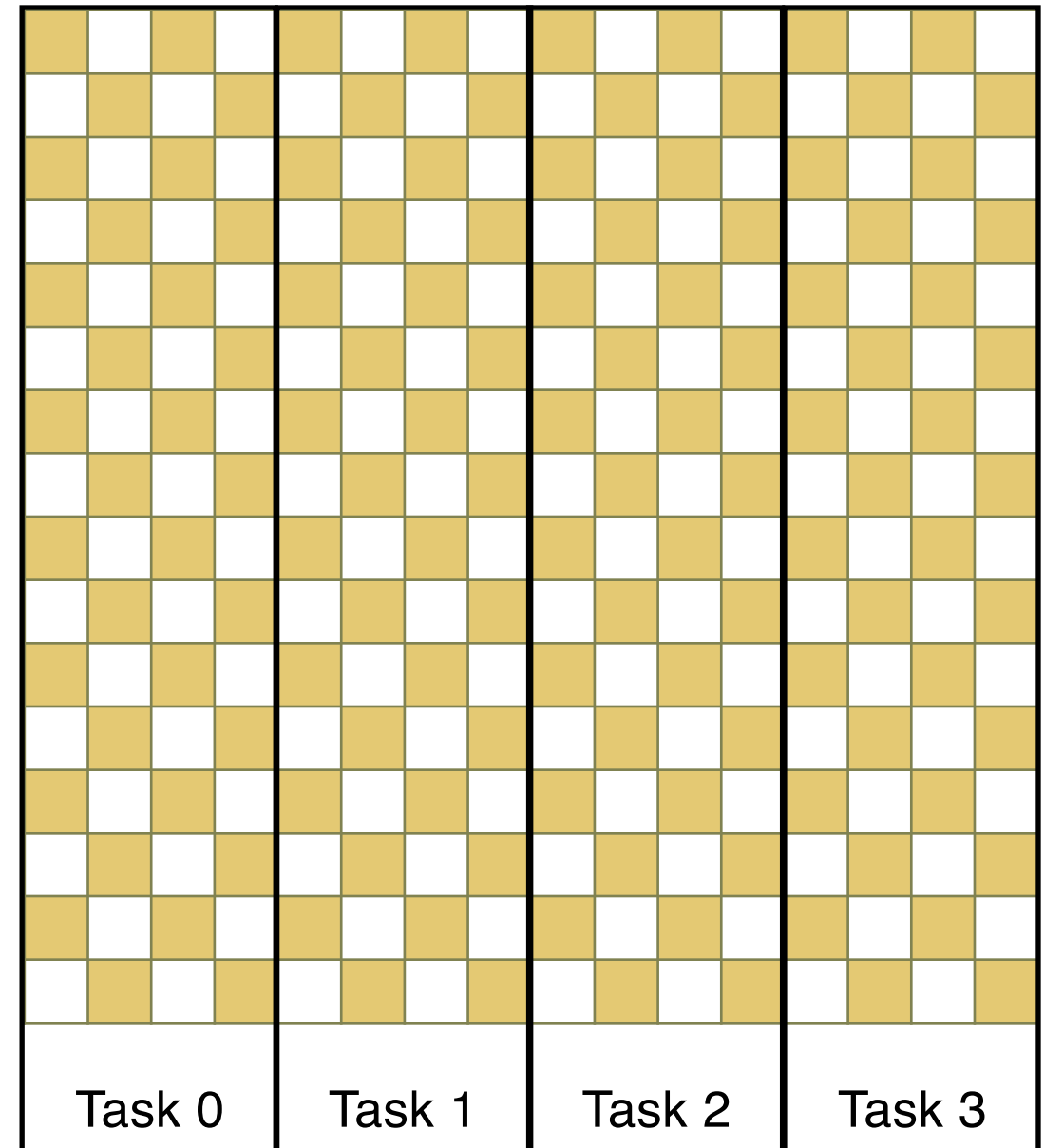


DESIGNING PARALLEL PROGRAMS – COMMUNICATIONS

Who needs communications?

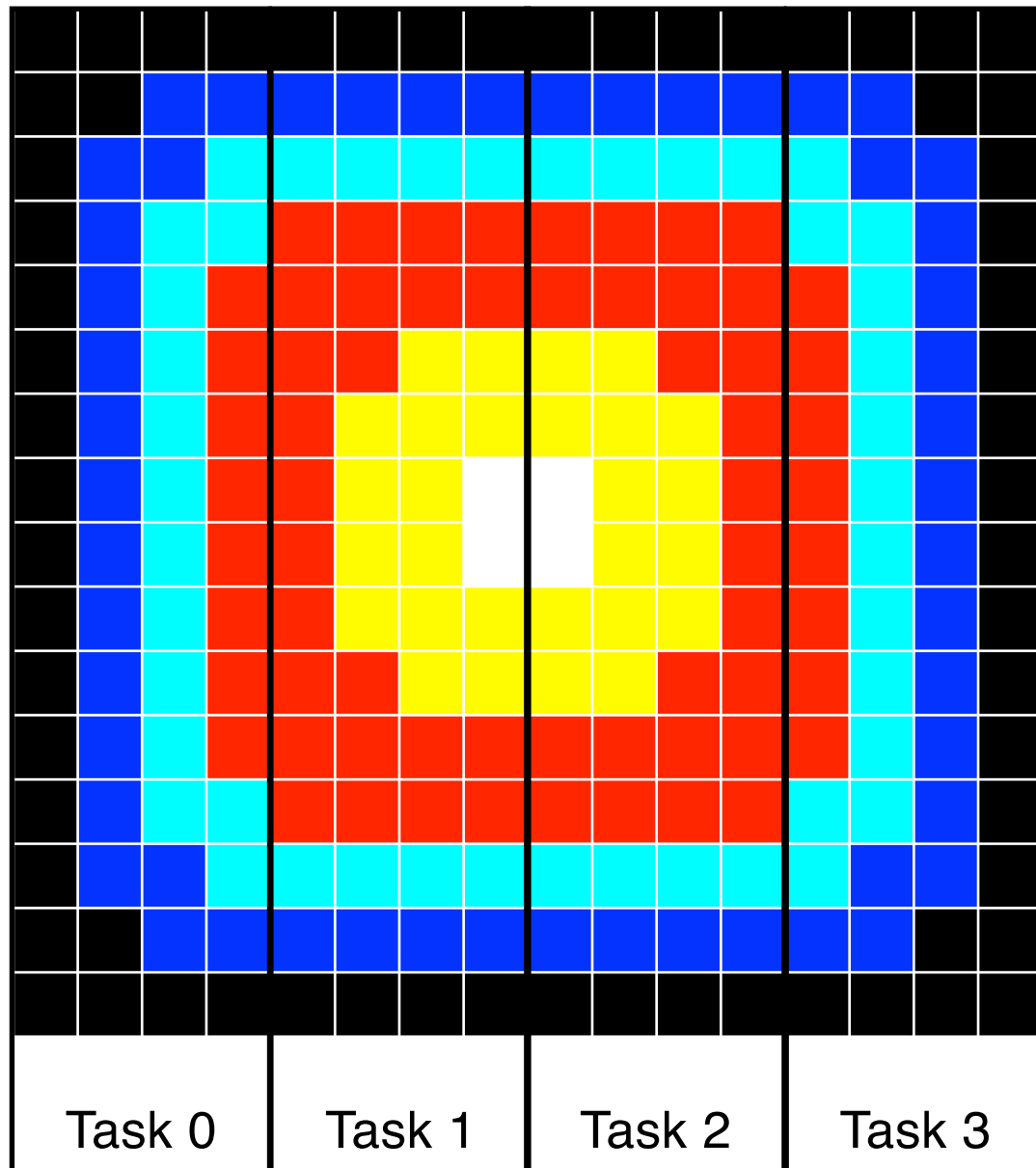
You don't need communications if:

- The problem can be decomposed and executed in parallel with no need for tasks to share data
- Often called “embarrassingly parallel” problems



DESIGNING PARALLEL PROGRAMS – COMMUNICATIONS

Who needs communications?



You do need communications if:

- Tasks cannot be executed completely independently - data needs to be shared
- Example: heat diffusion on a grid

DESIGNING PARALLEL PROGRAMS – COMMUNICATIONS

Factors to consider:

- Communication overhead
- Latency vs. Bandwidth
 - **Latency** is the time it takes to send a minimal message from point A to point B
 - **Bandwidth** is the amount of data that can be communicated per unit time
- Visibility of communications
- Synchronous vs. Asynchronous communications
- Scope of communications (point-to-point vs. collective)
- Efficiency of communications (network considerations, etc.)

DESIGNING PARALLEL PROGRAMS – SYNCHRONIZATION

Can be a significant factor in program performance - or lack thereof!

Types of synchronization:

- Barrier: Each task performs its work until it reaches the barrier. It then stops and waits for all the other tasks to reach the barrier before moving on.
- Lock / semaphore: Typically used to protect access to global data or section of code. Ex: One task acquires the lock and no other task can use that data until it is released.
- Synchronous communications operations: Usually involves e.g. sending data from one task to another.

DESIGNING PARALLEL PROGRAMS – DATA DEPENDENCIES

A *dependence* exists when one task cannot be performed until another task has completed. A *data dependence* exists when one task needs data being operated on by another task.

- Ex: loop carried data dependence

```
for i in range(0,N):  
    A[i] = A[i-1] * 2
```

- Ex: loop independent data dependence

task 1

x = 2

•

•

y = x2**

task 2

x = 4

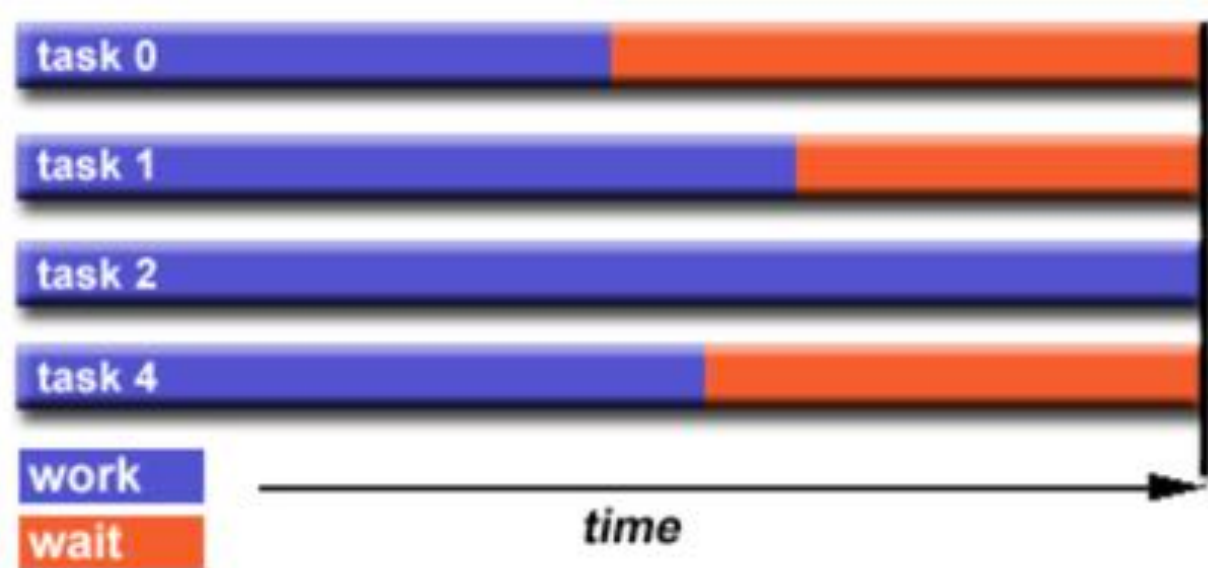
•

•

y = x3**

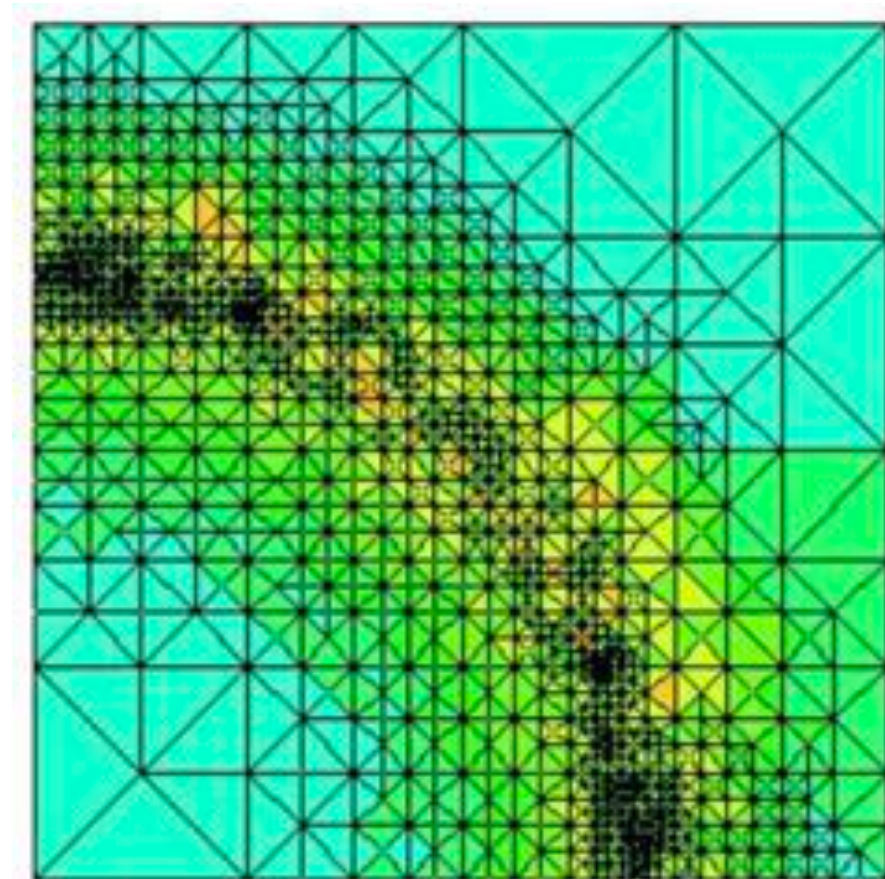
DESIGNING PARALLEL PROGRAMS – LOAD BALANCING

*Load balancing refers to the practice of making sure that work is distributed equally amongst all tasks. We want to keep **all** tasks busy **all** the time!*



This is where synchronization / barriers can get you.

Some problems can't be easily balanced by evenly distributing data. Ex: Adaptive mesh refinement simulations.



DESIGNING PARALLEL PROGRAMS – GRANULARITY

Granularity is a qualitative measure of the ratio of computation to communication.

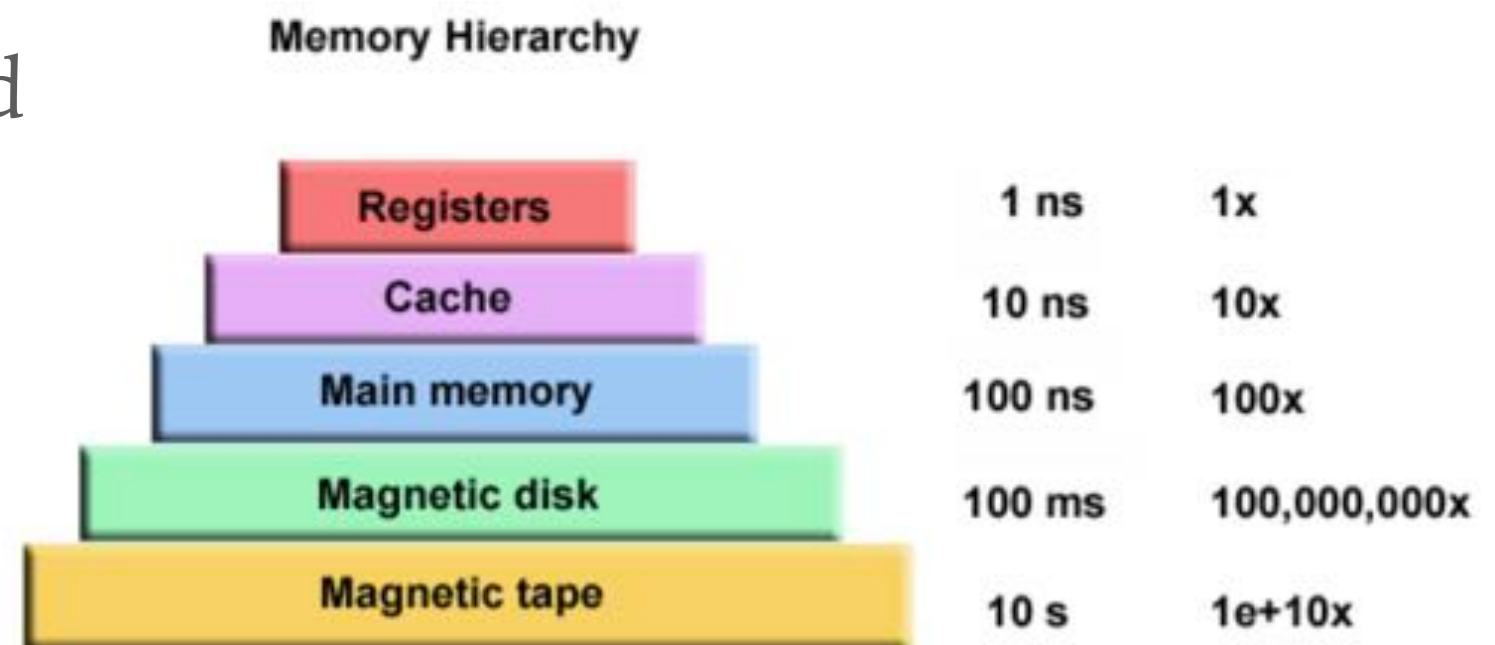
- Fine-grain parallelism: relatively small amounts of computation are done between communication / synchronization events
 - Facilitates load balancing
 - High communication overhead and less opportunity for improvement
- Coarse-grain parallelism: relatively large amounts of computation are done between communication / synchronization events
 - Harder to load balance
 - More opportunity for improvement

DESIGNING PARALLEL PROGRAMS – I/O

- I/O can be hard. It takes a long time, can be difficult to parallelize, and is often a program bottleneck.
- If done wrong, it can crash servers.

There are some general good practices:

- Reduce I/O as much as possible
- If you have access to a parallel file system, use it
- Fewer larger files is (usually) better than many smaller files



DEBUGGING AND PERFORMANCE ANALYSIS

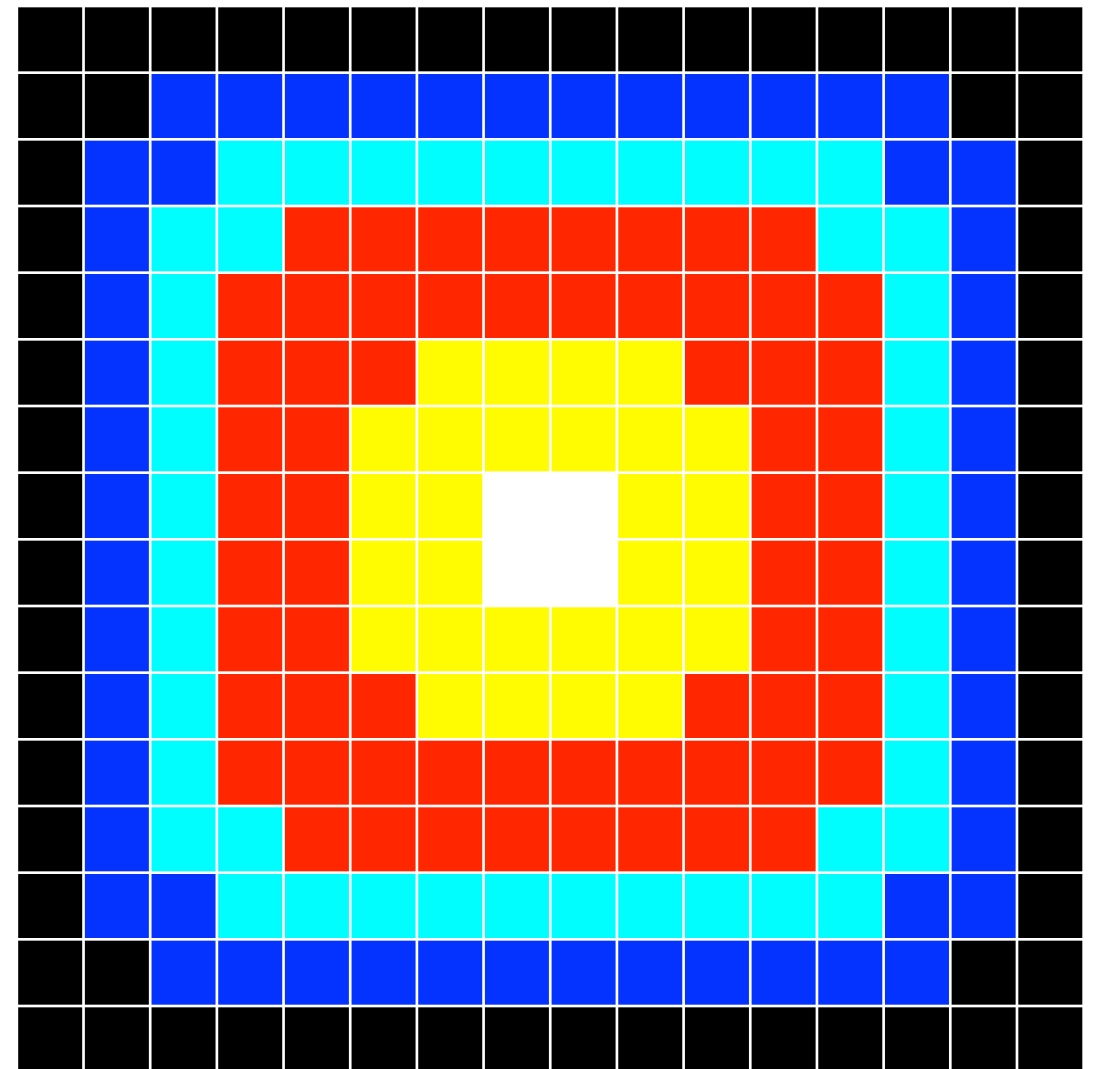
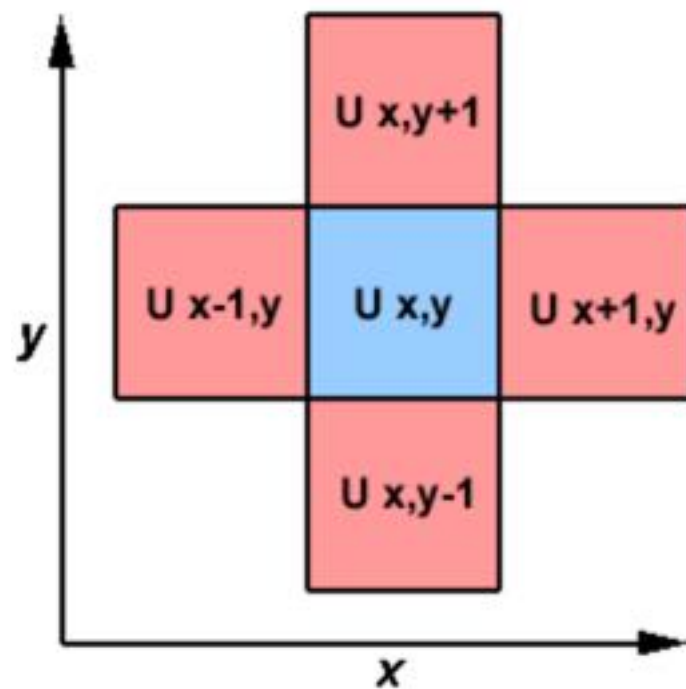
Debugging, analyzing, and tuning parallel programs can be much more challenging than serial programs.

- For debugging, can use (thoughtful) print statements
- Debuggers are recommended for complex programs
- Tools are changing all the time! Ask around / google regularly.

GROUP PROBLEM – MPI HEAT TRANSFER PROBLEM

The 2D heat equation describes temperature change over time, given an initial temperature distribution and boundary conditions.

The calculation of an element is dependent on neighboring values.



GROUP PROBLEM – MPI HEAT TRANSFER PROBLEM

.....

```
for i in range (1, nx-1):  
    for j in range (1, ny-1):  
        u2[i,j] = u1[i, j] + \\  
        cx * (u1[i+1, j] + u1[i-1, j] - 2*u1[i,j]) + \\  
        cy * (u1[i, j+1] + u1[i, j-1] - 2*u1[i,j])
```

- Is this problem able to be parallelized?
- How should the problem be partitioned?
- Are communications needed?
- Are there any data dependencies?
- Are there synchronization needs?
- Will load balancing be a concern?

GROUP PROBLEM – MPI HEAT TRANSFER PROBLEM

Possible Solution:

Find out if I am ROOT or CHILD

if I am ROOT

 initialize array

 send each WORKER starting info and subarray

 receive results from each WORKER

else if I am WORKER

 receive from ROOT starting info and subarray

 # Perform time steps

 from t = 0, nsteps

 update time

 send neighbors my boundary info

 receive from neighbors their boundary info

 update my portion of solution array

send ROOT results

PARALLEL PROGRAMMING TOOLS

So far, we have been talking about programming models. But how do we implement these models?

- Tools in python: mpi4py, numba
- Tools for C/C++: MPI, OpenMP
- Tools for GPUs: numba, OpenACC, CUDA

MPI4PY

<https://mpi4py.readthedocs.io/en/stable/>

- Mpi4py is a python interface to the MPI standard
- MPI stands for “message passing interface” - it provides a way to send messages from one process to another.
- These messages can be *data* needed by another process (Ex. boundary values in the heat diffusion problem), or *collectives* (Ex. calculating min or max across all processes)
 - Collectives include: reduce(sum, min, max, product), broadcast, gather, scatter

INTRODUCTION TO NUMBA

<https://numba.pydata.org/numba-doc/dev/index.html>

- Numba is a just-in-time compiler for python that works best on code that uses NumPy arrays, functions, and loops.
 - “Just-in-time” refers to the compilation strategy - the first time a Numba-decorated function is called, Numba compiles it and stores the result, so subsequent calls run at machine speed!
- Why is that important? Python is an “interpretive” language - every line is interpreted and compiled one at a time. This makes standard python *very* slow (relative to compiled languages like C).

INTRODUCTION TO NUMBA

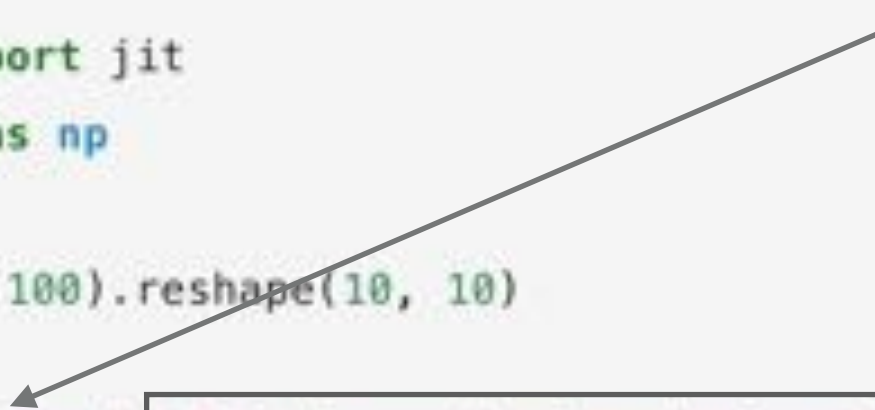
When does Numba work well?

```
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

@jit(nopython=True) # Set "nopython" mode for best performance, equivalent to @njit
def go_fast(a): # Function is compiled to machine code when called the first time
    trace = 0
    for i in range(a.shape[0]): # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace # Numba likes NumPy broadcasting

print(go_fast(x))
```



nopython mode instructs Numba to compile the decorated function without the use of the Python interpreter

INTRODUCTION TO NUMBA

When does Numba work poorly?

```
from numba import jit
import pandas as pd

x = {'a': [1, 2, 3], 'b': [20, 30, 40]}

@jit
def use_pandas(a): # Function will not benefit from Numba jit
    df = pd.DataFrame.from_dict(a) # Numba doesn't know about pd.DataFrame
    df += 1 # Numba doesn't understand what this is
    return df.cov() # or this!

print(use_pandas(x))
```


INTRODUCTION TO NUMBA

Measuring performance correctly is important!

```
from numba import jit
import numpy as np
import time

x = np.arange(100).reshape(10, 10)

@jit(nopython=True)
def go_fast(a): # Function is compiled and runs in machine code
    trace = 0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace

# DO NOT REPORT THIS... COMPILATION TIME IS INCLUDED IN THE EXECUTION TIME!
start = time.time()
go_fast(x)
end = time.time()
print("Elapsed (with compilation) = %s" % (end - start))

# NOW THE FUNCTION IS COMPILED, RE-TIME IT EXECUTING FROM CACHE
start = time.time()
go_fast(x)
end = time.time()
print("Elapsed (after compilation) = %s" % (end - start))
```

INTRODUCTION TO NUMBA

How does it work?

- Numba reads the Python bytecode for a decorated function
- It combines this with info about the *types* of the input arguments to the function
- It analyzes and optimizes your code, then uses the LLVM compiler library to generate a machine code version of your function, tailored to your CPU capabilities
- This compiled version is cached and used every time your function is called