Parallel Programming Concepts

Feng Chen
HPC User Services
LSU HPC & LONI
sys-help@loni.org

Louisiana State University
Baton Rouge
May 30, 2018

What we'll be doing

- **Learning to write C/Fortran programs that are explicitly parallel.**
  - Parallel Programming Concepts
  - Introduction to MPI, Part 1
    - Point to Point communications
  - Introduction to MPI, Part 2
    - User Defined Types
    - Collective Operations
  - Understanding Parallel Applications

Outline

- **Why parallel programming?**
  - Some background
- **Parallel hardware and software**
  - Parallel computing architecture
  - Parallel programming models
- **Performance evaluation**
  - Speedup, efficiency
  - Amdahl's law
  - Gustafson's law
  - Scalability

- **Lab session**
  - Compile and run OpenMP/MPI codes
  - Speedup and efficiency
  - Load balancing

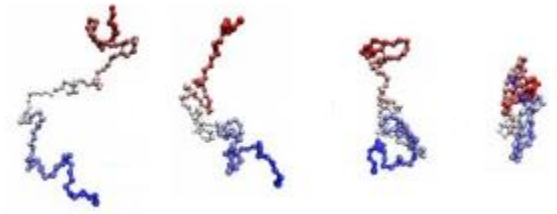
Parallel Programming Concepts

Why Parallel Programming?

Applications

Climate Modeling

Data Analysis

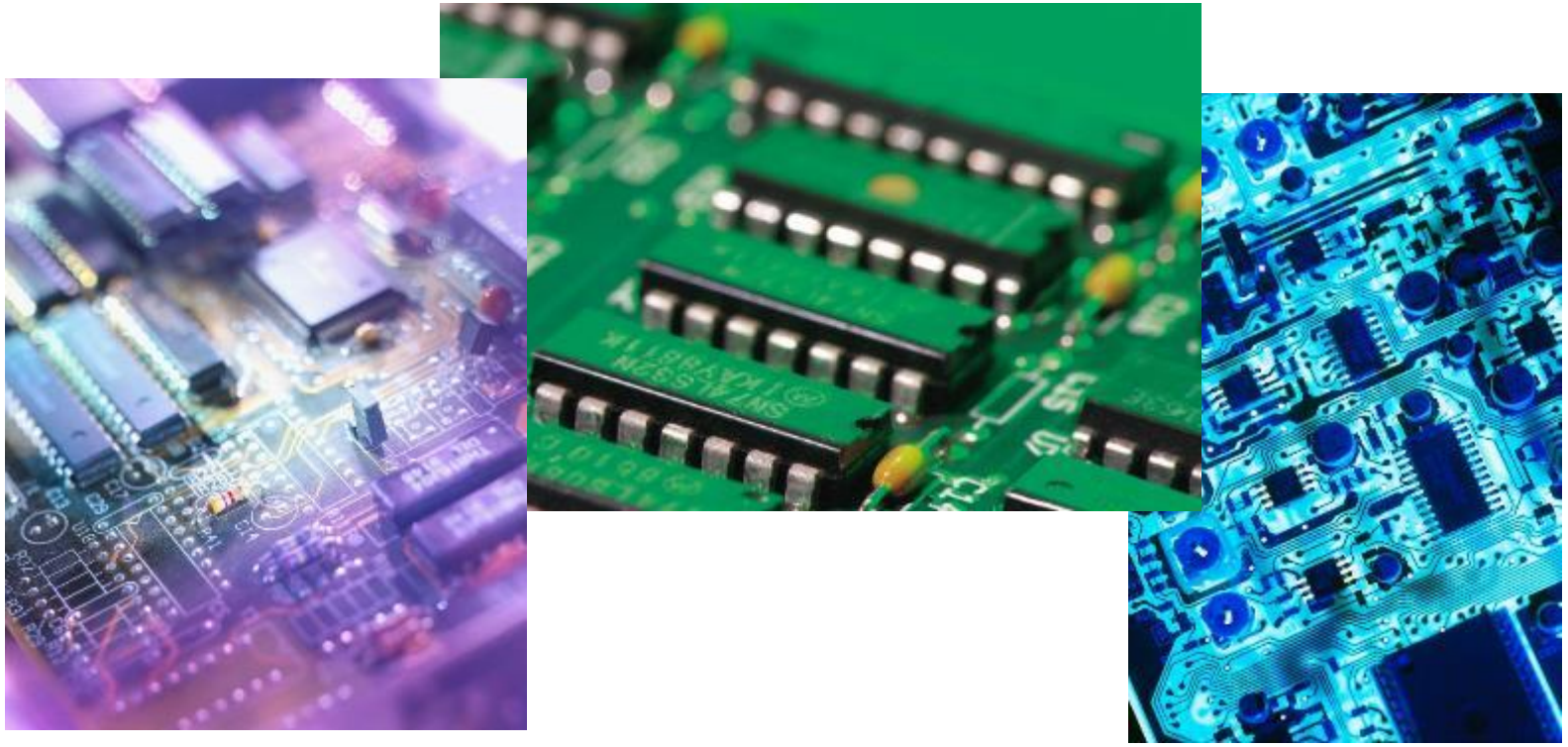


Protein folding

Energy Research

Drug Discovery

Some background



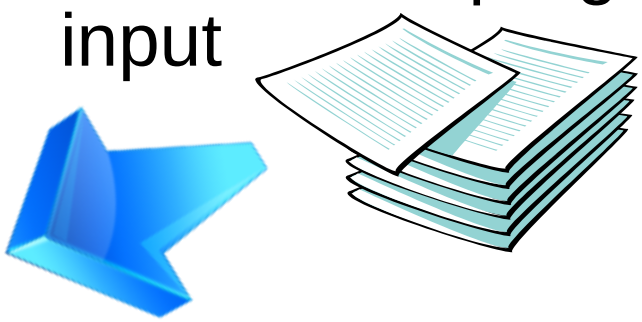
Serial hardware and software

Computer runs one program at a time.

output

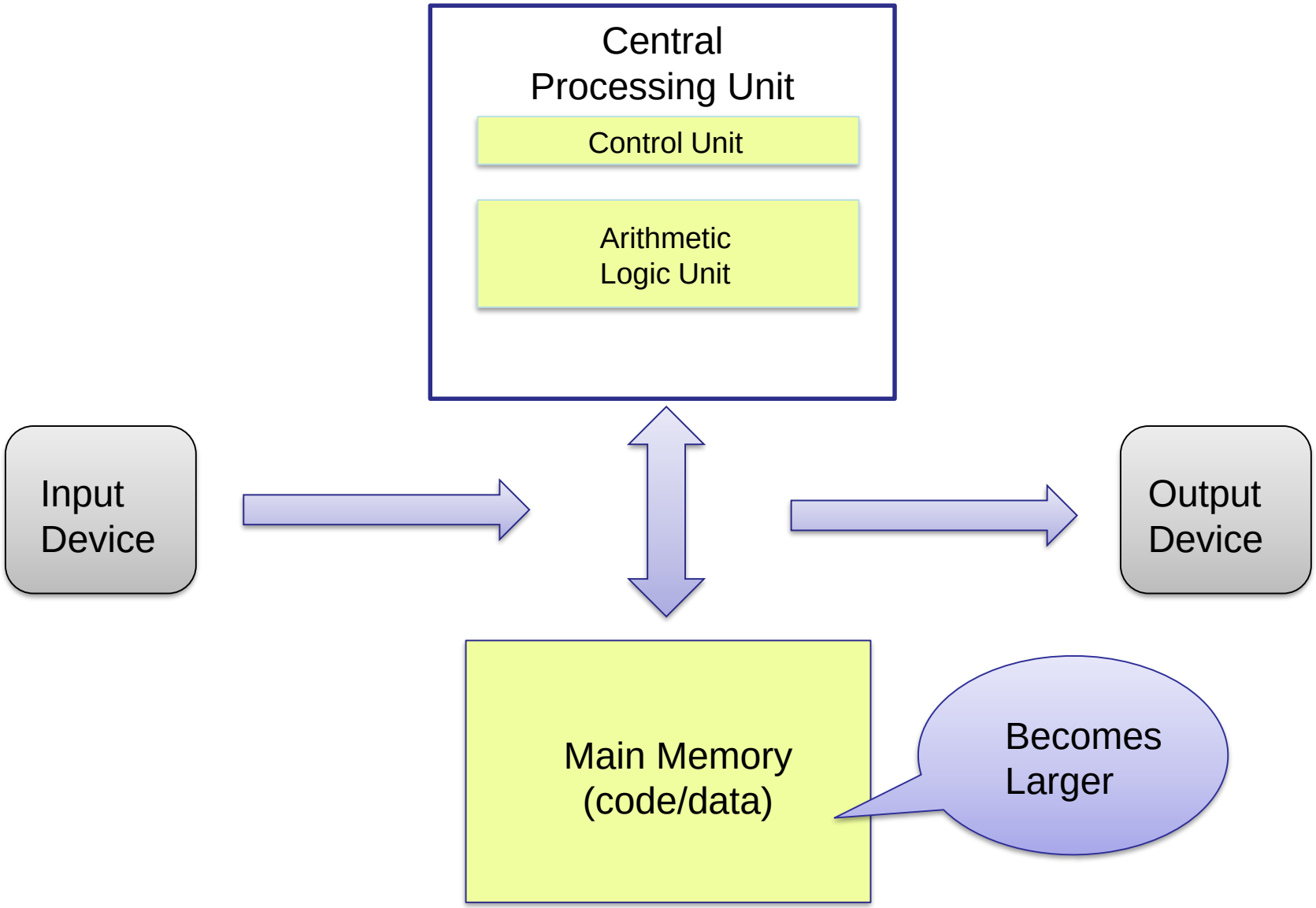
programs

input

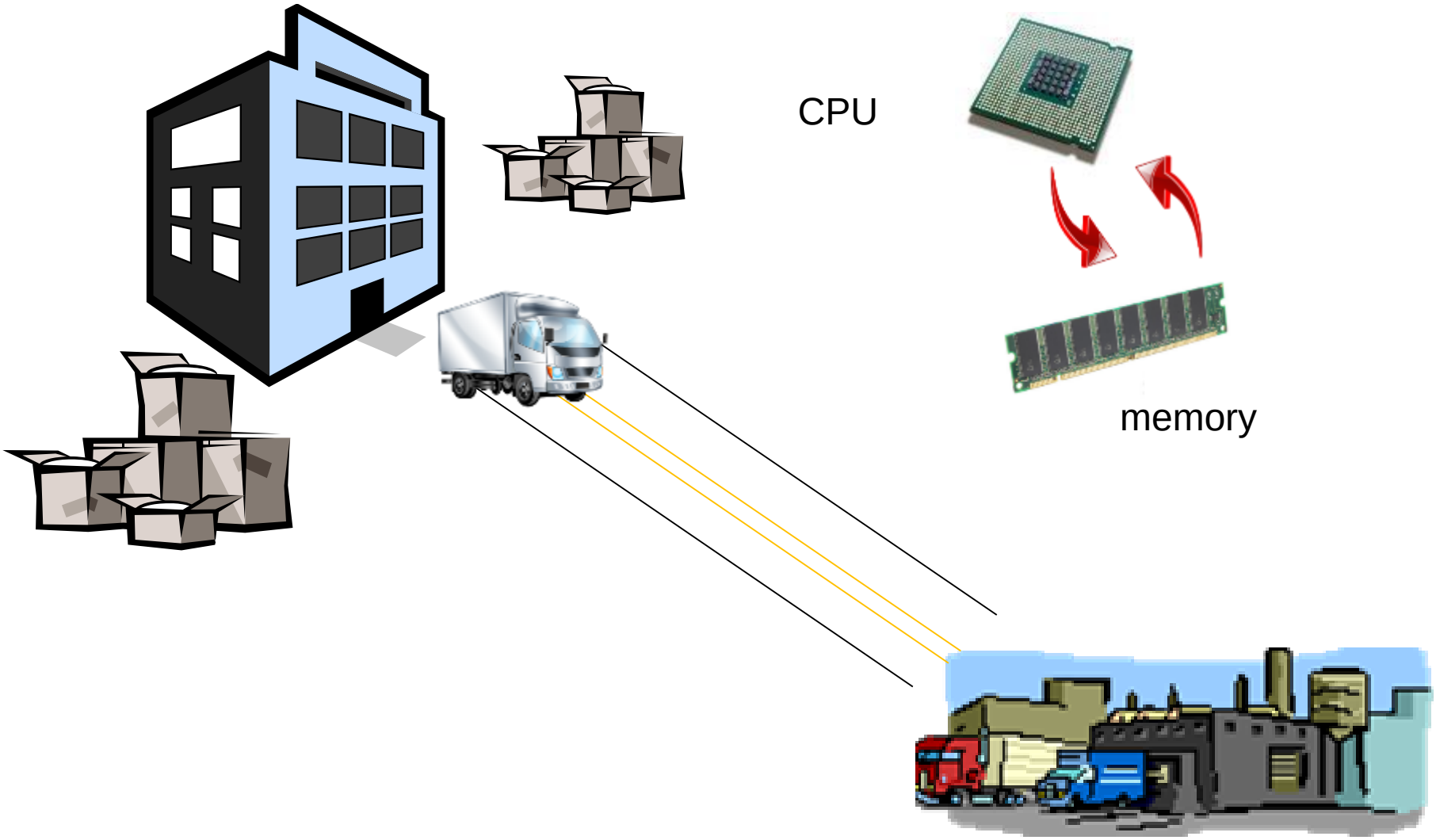


Can we have something that just run 100x faster?

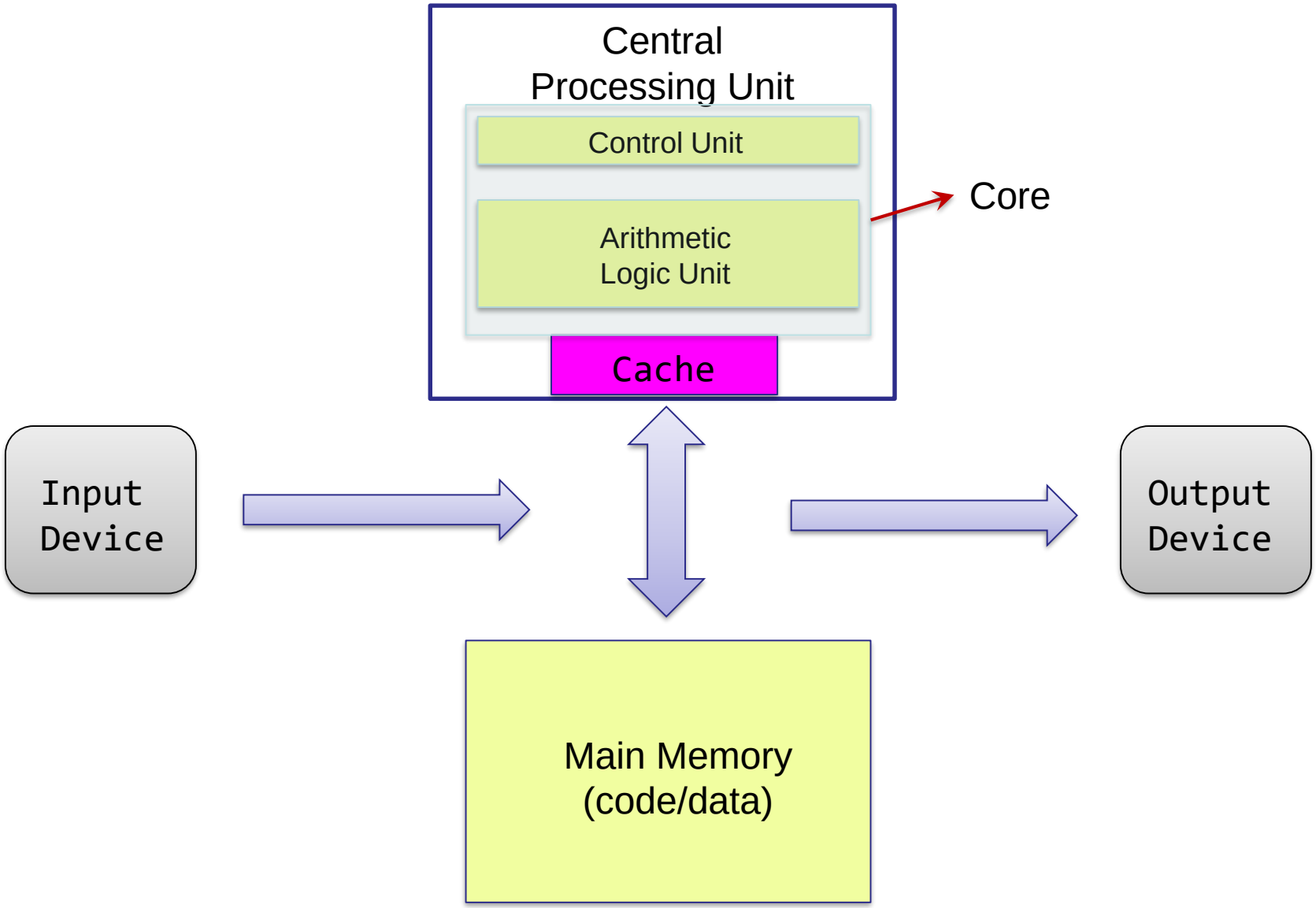
The von Neumann Architecture



von Neumann bottleneck



The von Neumann Architecture



Cache - Principle of locality

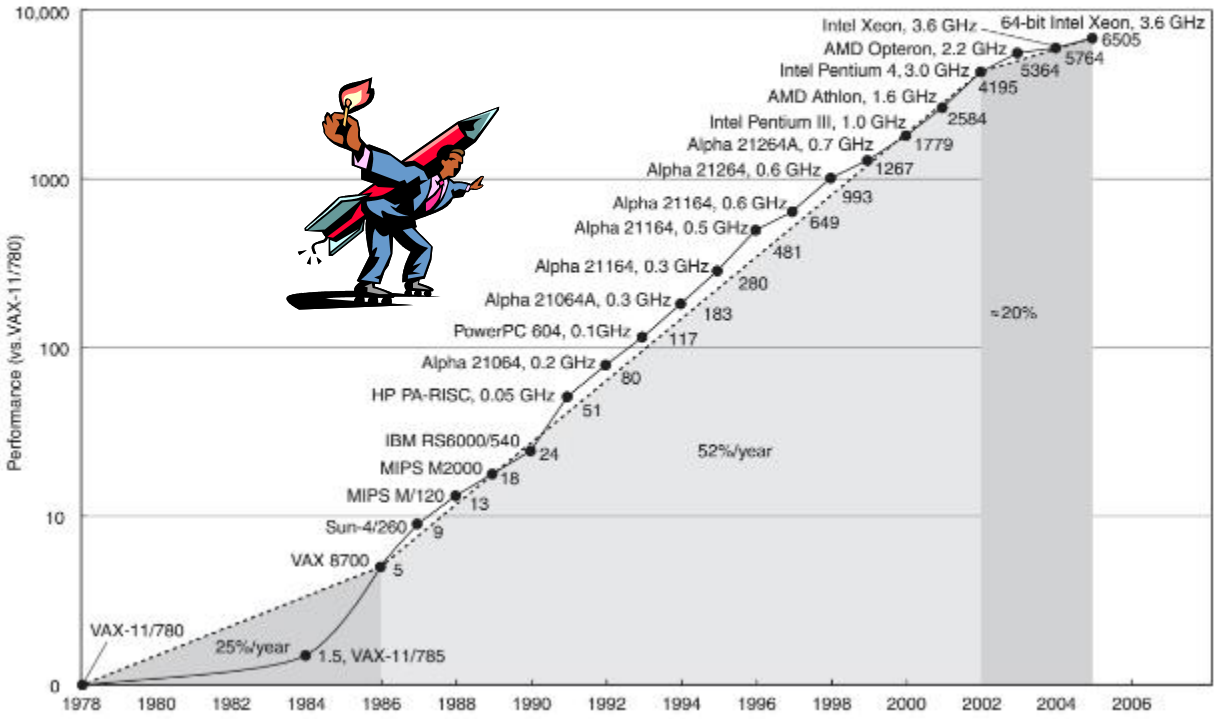
- **A CPU cache is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory.**
- **A cache is a smaller, faster memory, closer to a processor core, which stores copies of the data from frequently used main memory locations.**

- **Accessing one location is followed by an access of a nearby location.**
  - **Spatial locality** – accessing a nearby location.
  - **Temporal locality** – accessing in the near future.

Changing Times

- **From 1986 - 2002, microprocessors were speeding like a rocket, increasing in performance an average of 50% per year.**
- **Since then, it's dropped to about 20% increase per year.**

History of Processor Performance



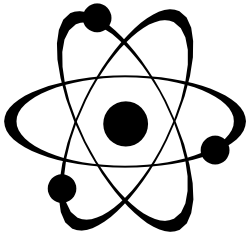
Limitation:

2 GHz Consumer
4 GHz Server

Source:
http://www.cs.columbia.edu/~sedwards/classes/2012/3827-spring/

A Little Physics Problem

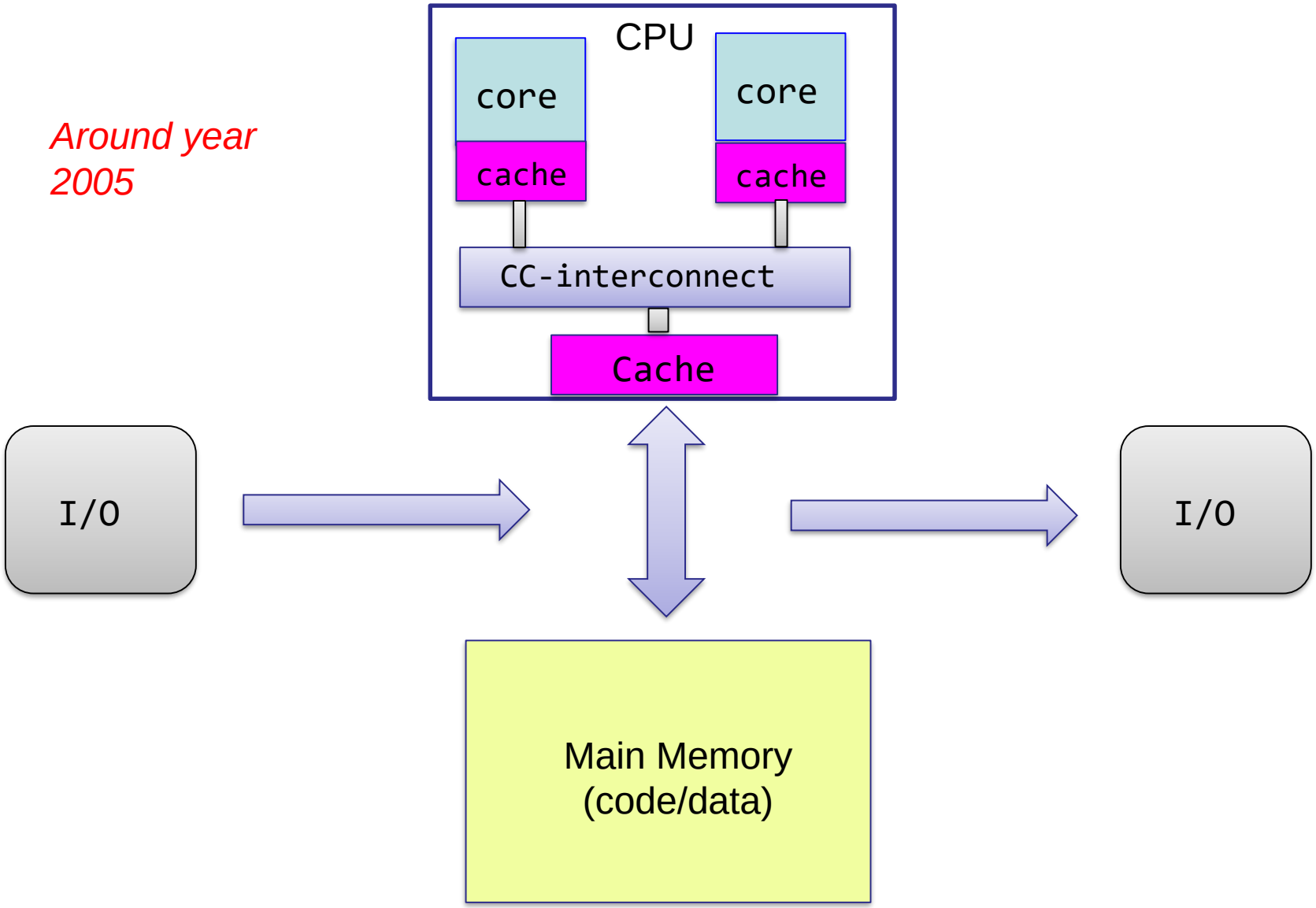
- **Smaller transistors = faster processors.**
- **Faster processors = increased power consumption.**
- **Increased power consumption = increased heat.**
- **Increased heat = unreliable processors.**



- **Solution:**
  - Move away from single-core systems to multicore processors.
  - “core” = central processing unit (CPU)
  - Introducing parallelism
    - *What if your problem is also not CPU dominant?*

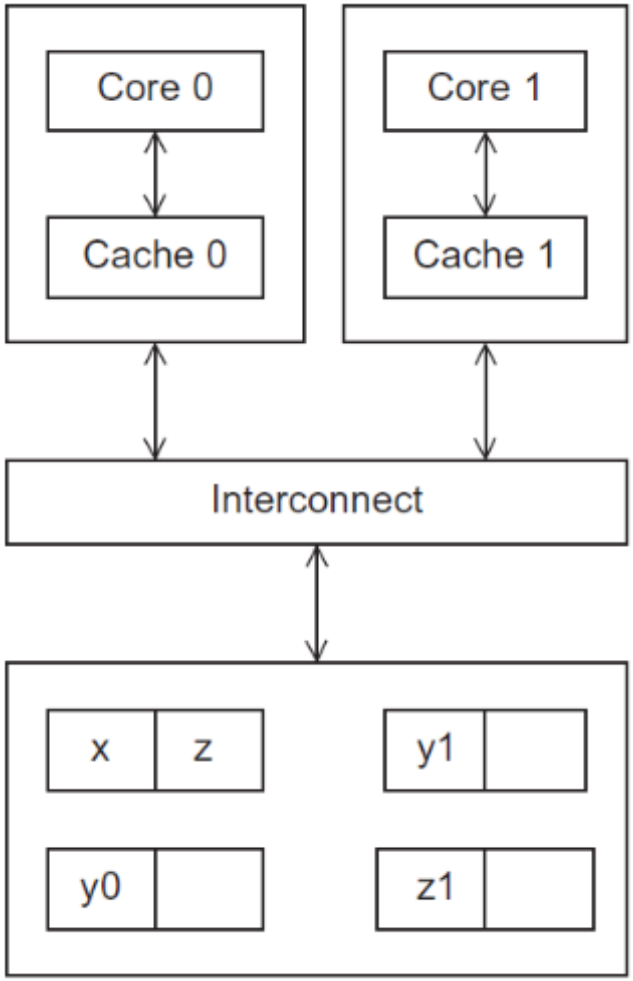
The von Neumann Architecture

Around year 2005



Cache coherence

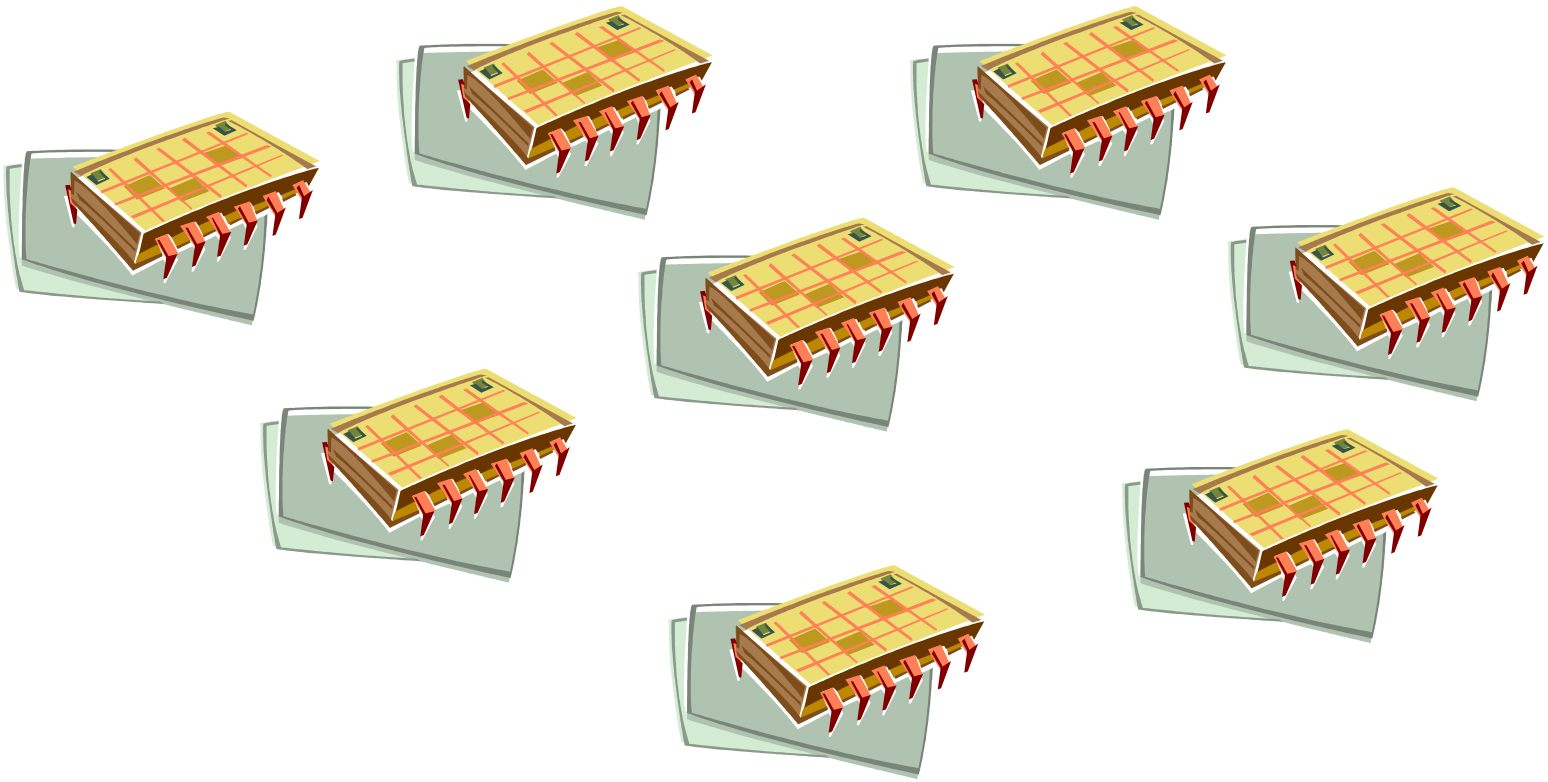
- **Programmers have no control over caches and when they get updated.**



A shared memory system with two cores and two caches

An intelligent solution

- **Instead of designing and building faster microprocessors, put <u>multiple</u> processors on a single integrated circuit.**



Now it's up to the programmers

- **Up to now, performance increases have been attributable to increasing density of transistors.**
  - But there are inherent problems...
- **Adding more processors doesn't help much if programmers aren't aware of them…**
  - … or don't know how to use them.
- **Serial programs don't benefit from this approach (in most cases).**

Why we need to write parallel programs

- **Running multiple instances of a serial program often isn't very useful.**
  - Exception: Many embarrassingly parallel problems
    - Solution: E.g. GNU Parallel
- **Think of running multiple instances of your favorite game.**
  - Run VR game twice?
  - What you really want is for it to run faster.

Approaches From Serial To Parallel

- **Write translation programs that automatically convert serial programs into parallel programs.**
  - This is very difficult to do.
  - Success has been limited.

- ***Rewrite serial programs so that they're parallel.***
  - Some coding constructs can be recognized by an automatic program generator, and converted to a parallel construct.
  - However, it's likely that the result will be a very inefficient program.
  - Sometimes the best parallel solution is to step back and devise an entirely new algorithm.

How Do We Write Parallel Programs?

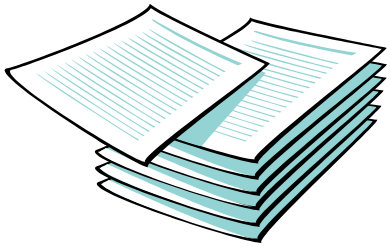
- **Task parallelism**
  - Partition various tasks carried out solving the problem among the cores.

- **Data parallelism**
  - Partition the data used in solving the problem among the cores.
  - Each core carries out similar operations on it's part of the data.

Professor P

15 questions

300 exams

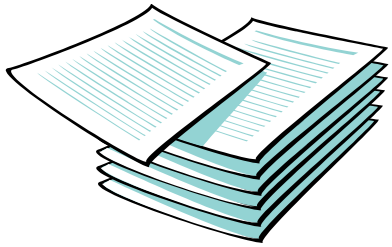


Professor P's grading assistants

Division Of Work

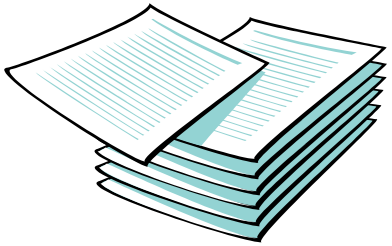
Data Parallelism

TA#1



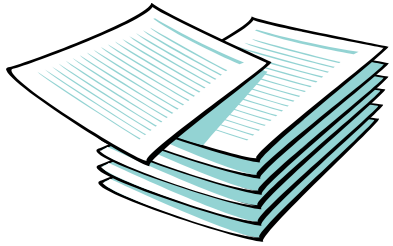
100 exams

TA#2



100 exams

TA#3



100 exams

Task Parallelism

TA#1

Questions 1 - 5

TA#2

Questions 6 - 10

TA#3

Questions 11 - 15

Coordination

- **Cores usually need to coordinate their work.**
  - Communication – one or more cores send their current partial sums to another core.
  - Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest.
  - Load balancing – share the work evenly among the cores so that one is not heavily loaded.

Terminology

- **Concurrent computing – a program is one in which multiple tasks can be in progress at any instant.**
- **Parallel computing – a program is one in which multiple tasks cooperate closely to solve a problem**
- **Distributed computing – a program may need to cooperate with other programs to solve a problem.**

Concluding Remarks

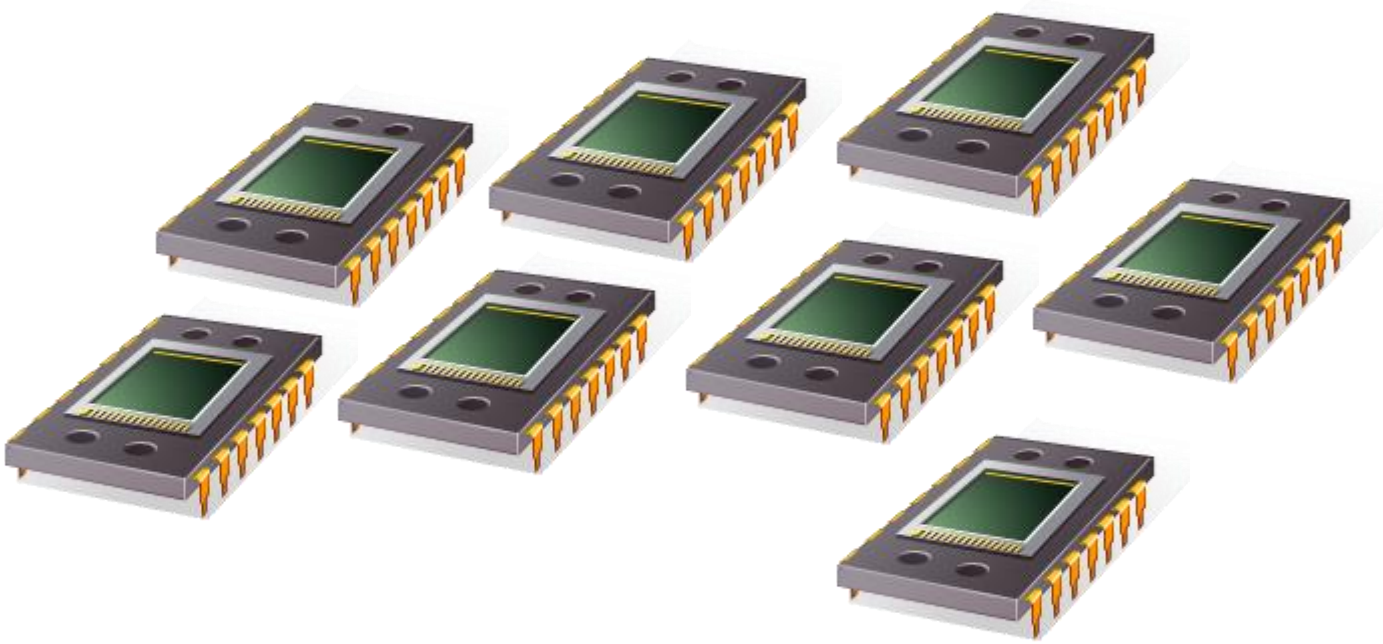
- **The laws of physics have brought us to the doorstep of multicore technology.**
- **Serial programs typically don't benefit from multiple cores.**
- **Automatic parallel program generation from serial program code isn't the most efficient approach to get high performance from multicore computers.**
- **Parallel programs are usually very complex and therefore, require sound program techniques and development.**
  - Task Parallelism vs Data Parallelism
  - Learning to write parallel programs involves learning how to coordinate the cores.
    - Communication/Load balancing/Synchronization

Parallel Programming Concepts

Parallel Hardware

Parallel Hardware

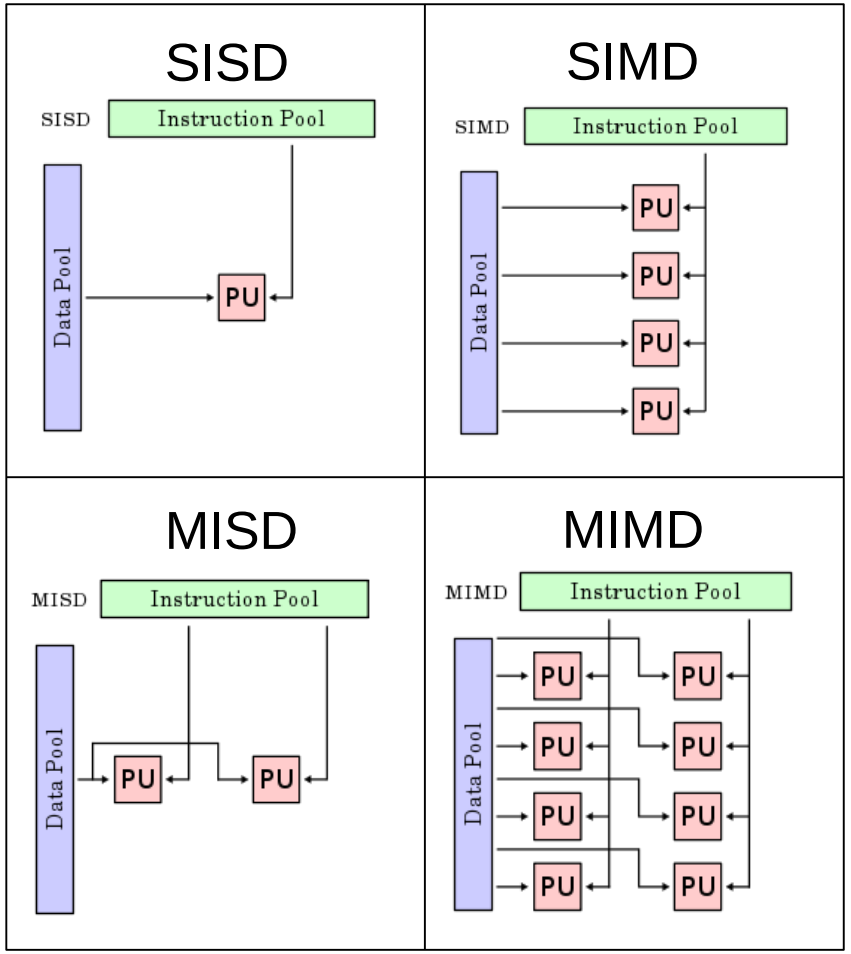
- **A programmer can write code to exploit.**



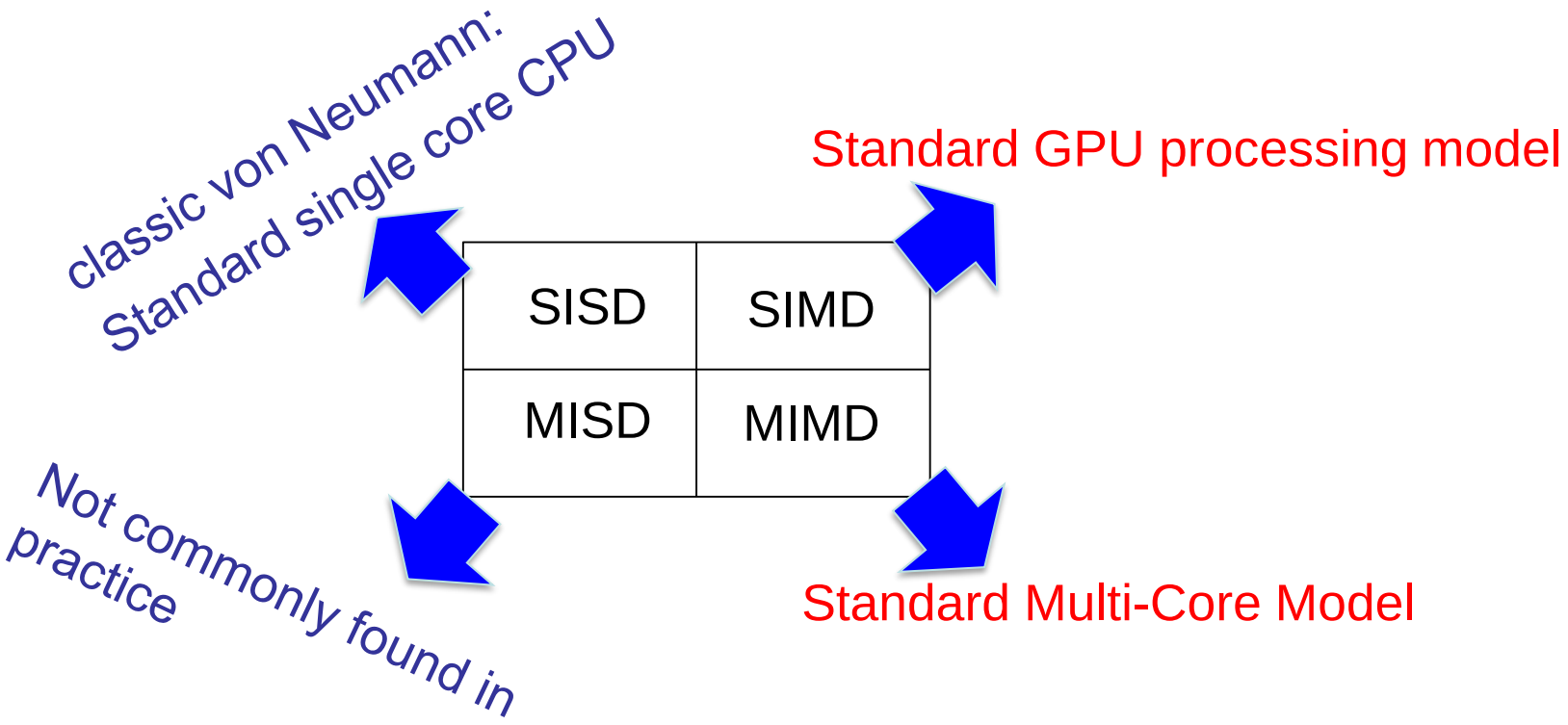
Flynn's Taxonomy on Computer Architectures

design of modern processors

	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD



Flynn's Taxonomy of Parallelism

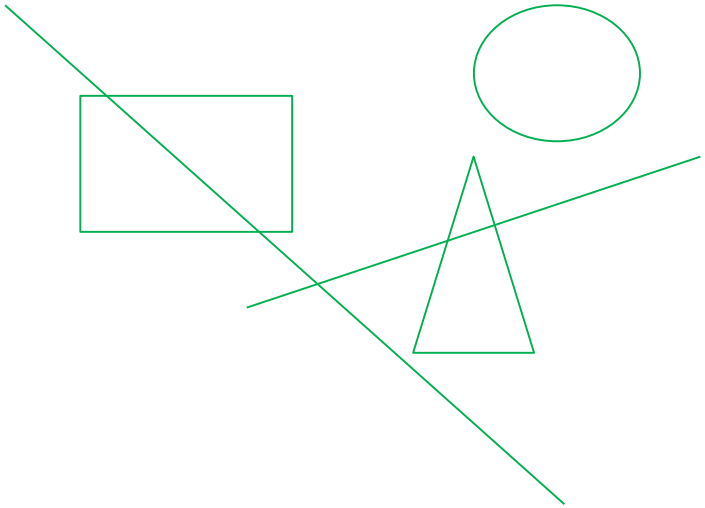


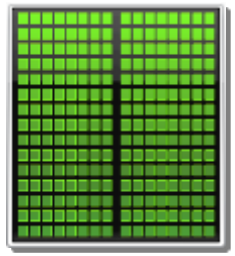
SIMD

- **Parallelism achieved by dividing data among the processors.**
- **Applies the same instruction to multiple data items.**
- **Called data parallelism.**

SIMD example
Graphics Processing Units (GPU)

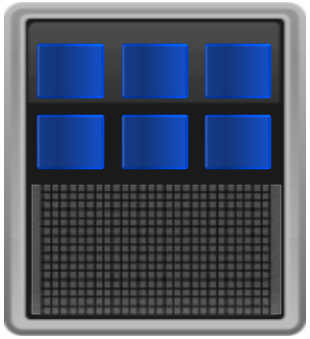
- **A graphics processing pipeline converts the internal representation into an array of pixels that can be sent to a computer screen.**
- **Real time graphics application programming interfaces or API's use points, lines, and triangles to internally represent the surface of an object.**
- **GPU's can often optimize performance by using SIMD parallelism.**
  - The current generation of GPU's use SIMD parallelism, although they are not pure SIMD systems.





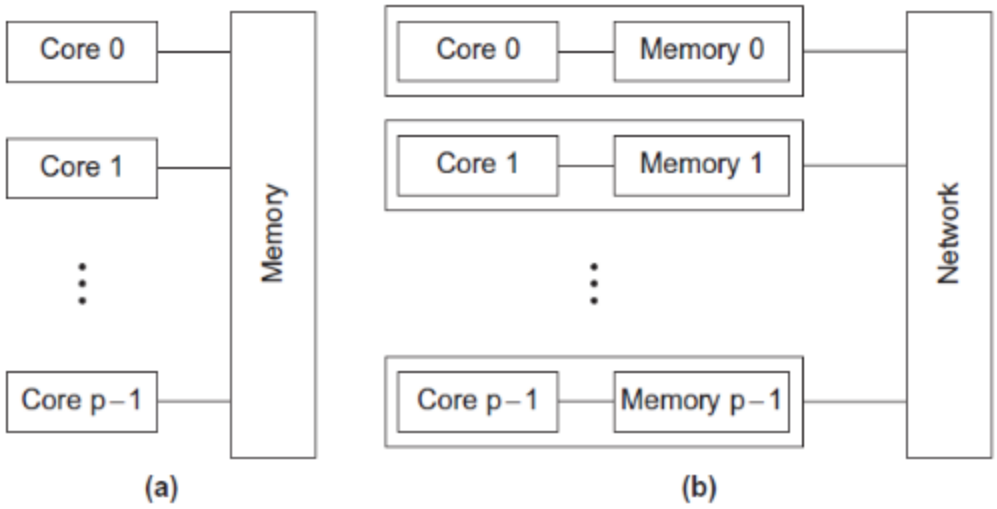
MIMD

- **In computing, MIMD (multiple instruction, multiple data) is a technique employed to achieve parallelism.**
- **Machines using MIMD have a number of processors that function asynchronously and independently. At any time, different processors may be executing different instructions on different pieces of data.**
  - Supports multiple simultaneous instruction streams operating on multiple data streams. Each of the processors in an SMP has its own instruction decoding unit, so they can all carry out different instructions simultaneously (Multiple Instruction).
  - All processors in an SMP address the same memory banks, but there is no architectural requirement that they have to coordinate which elements they are working on within that memory, so processors can simultaneously manipulate different data elements (Multiple Data).



Type Of Parallel ***Hardware*** Systems

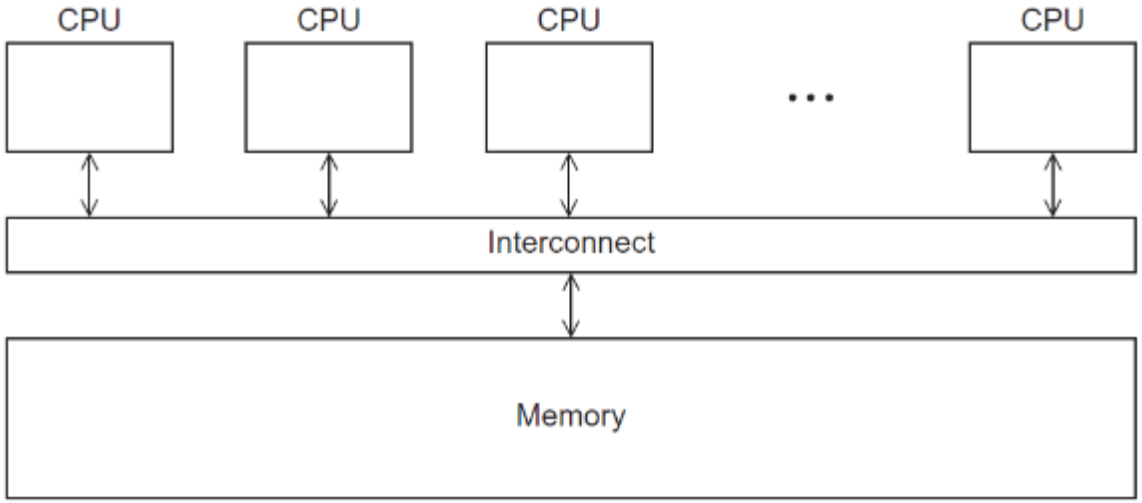
- **Shared-memory**
  - The cores can share access to the computer's memory.
  - Coordinate the cores by having them examine and update shared memory locations.
- **Distributed-memory**
  - Each computation unit has its own, private memory.
  - The computation units must communicate explicitly by sending messages across a network.



Shared-memory Distributed-memory

Shared Memory System

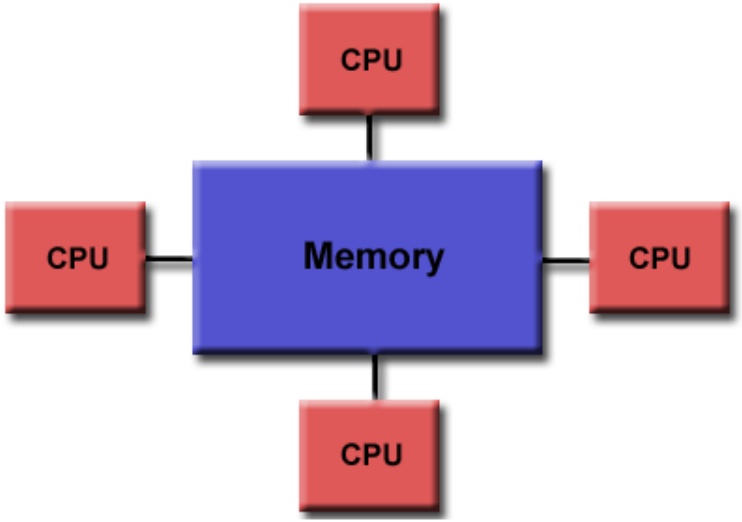
- **A collection of autonomous processors is connected to a memory system via an interconnection network.**
- **Each processor can access each memory location.**
- **The processors usually communicate implicitly by accessing shared data structures.**
- **Most widely available shared memory systems use one or more multicore processors.**
  - (multiple CPU's or cores on a single chip)



UMA Multicore System

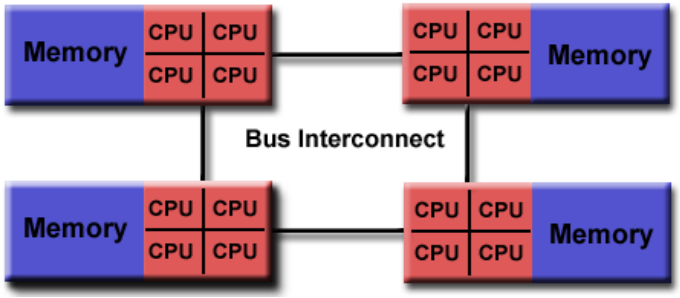
- **UMA (Uniform Memory Access) is commonly represented today by Symmetric Multiprocessor (SMP) machines**
  - Identical processors
  - Equal access and access times to memory
  - Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

Time to access all the memory locations will be the same for all the cores.



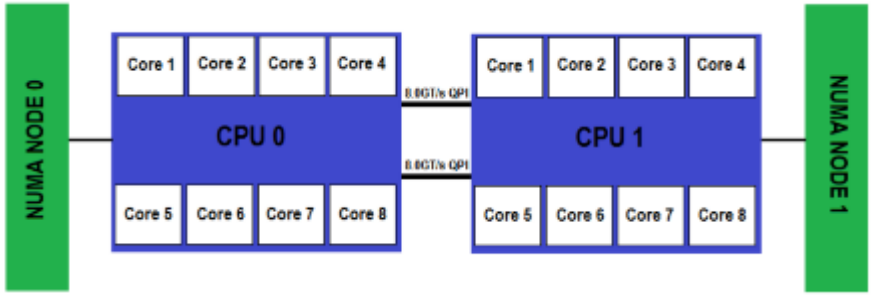
NUMA multicore system

- **NUMA (Non-Uniform Memory Access) often made by physically linking two or more SMPs**



  - One SMP can directly access memory of another SMP
  - Not all processors have equal access time to all memories
  - Memory access across link is slower
  - If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

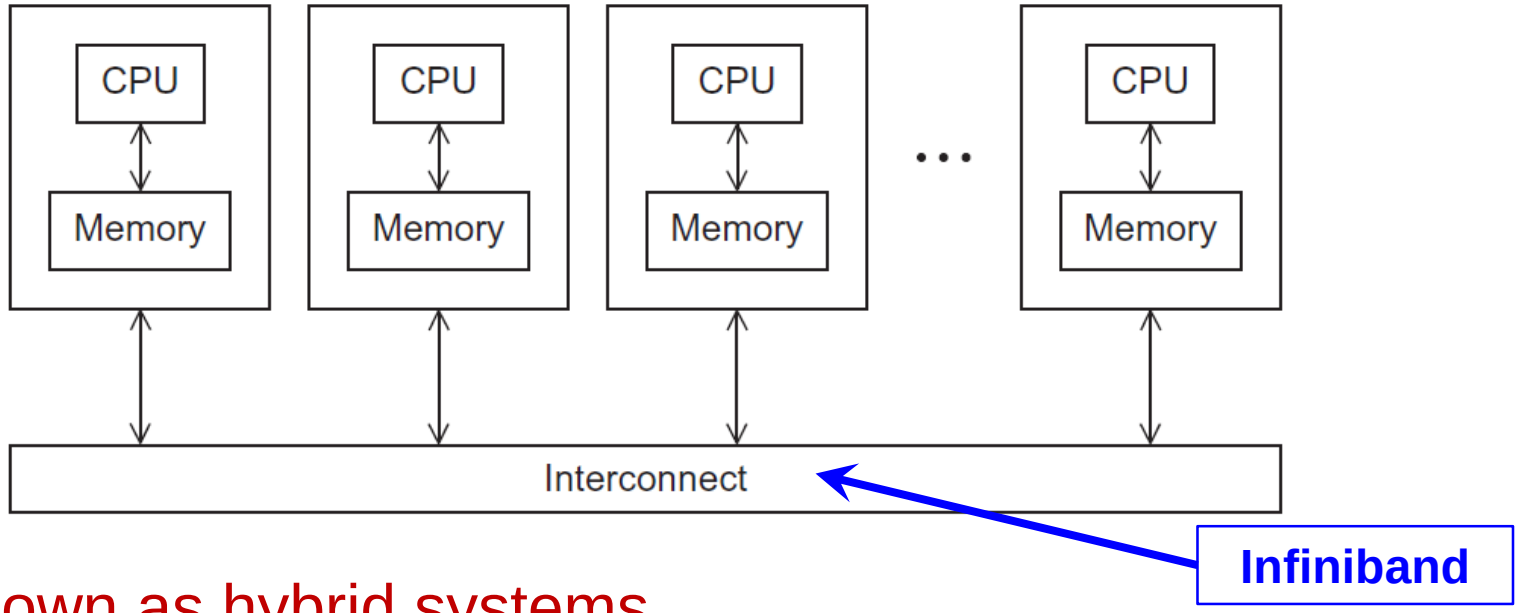


a shelob node model

Distributed Memory System

- **Clusters (most popular)**
  - A collection of commodity systems.
  - Connected by a commodity interconnection network.

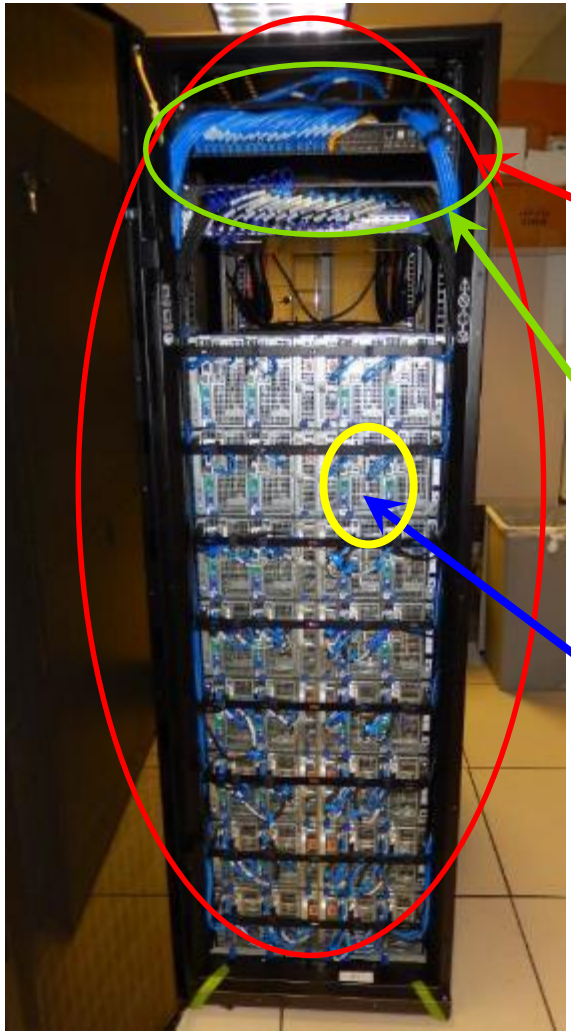
- **Nodes of a cluster are individual computations units joined by a communication network.**



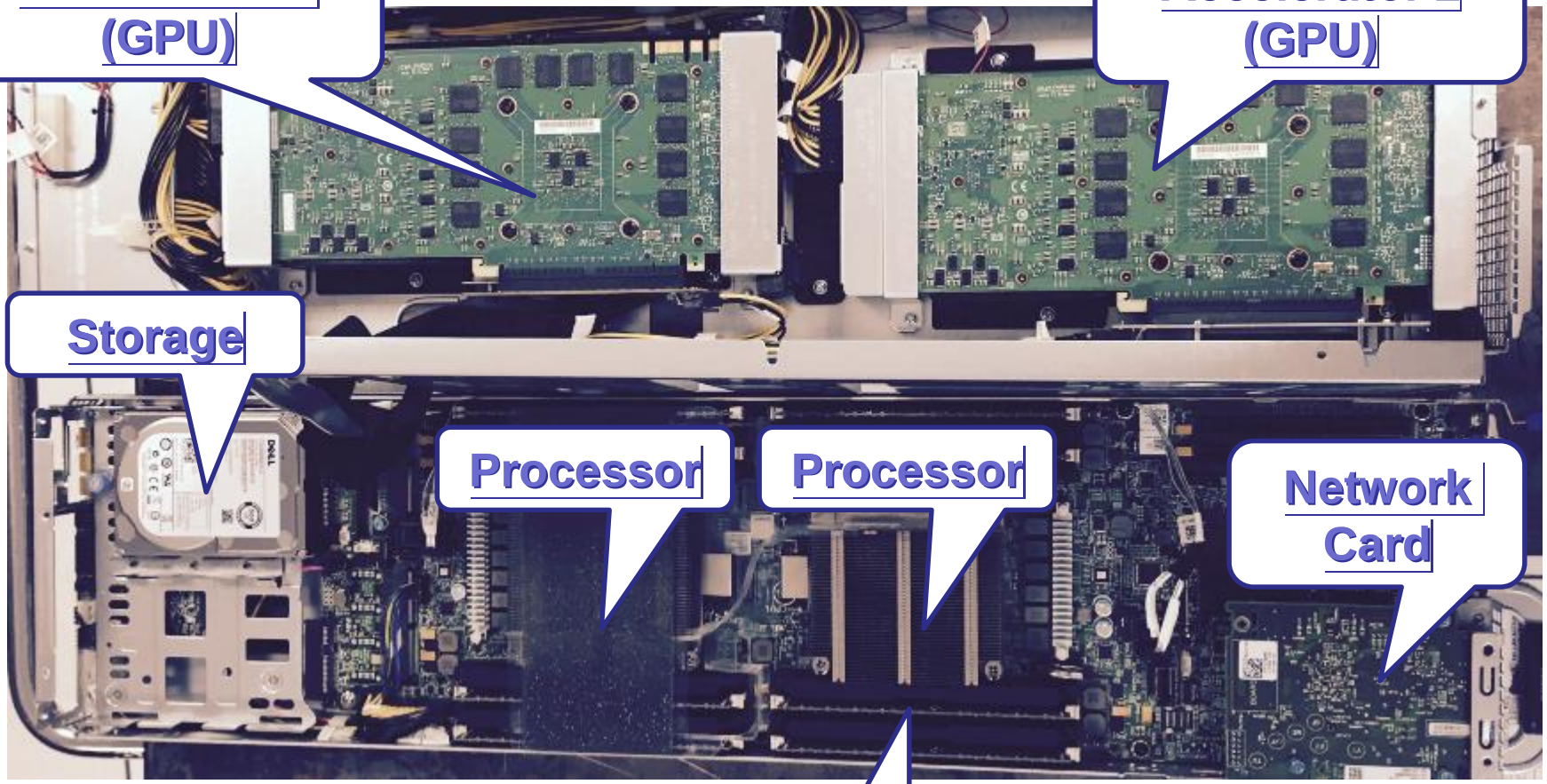
Also known as hybrid systems

Cluster And A Cluster Rack

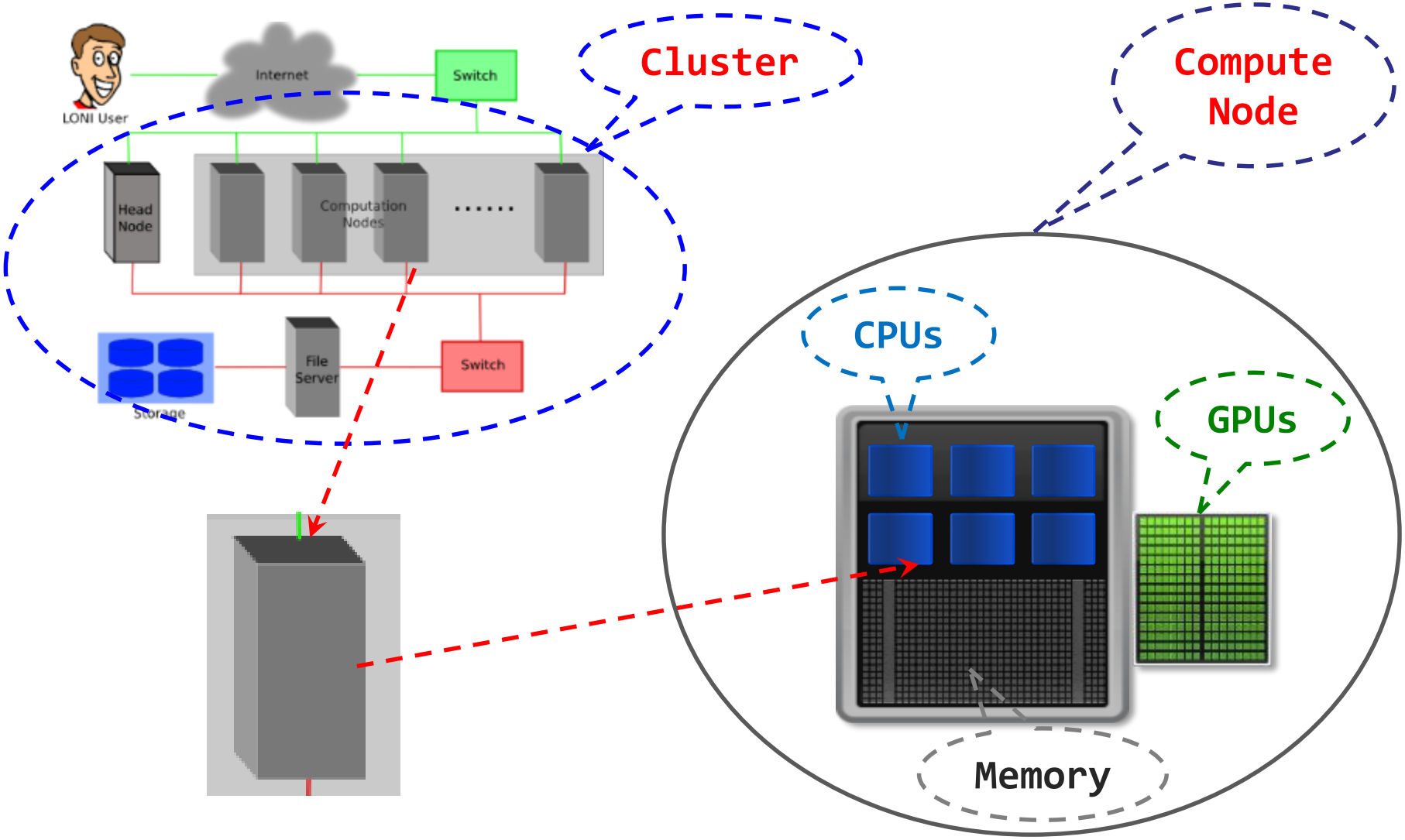
The QB2 cluster



Inside A Dell C8000 Compute Node



Conceptual Relationship



Conclusion Remarks

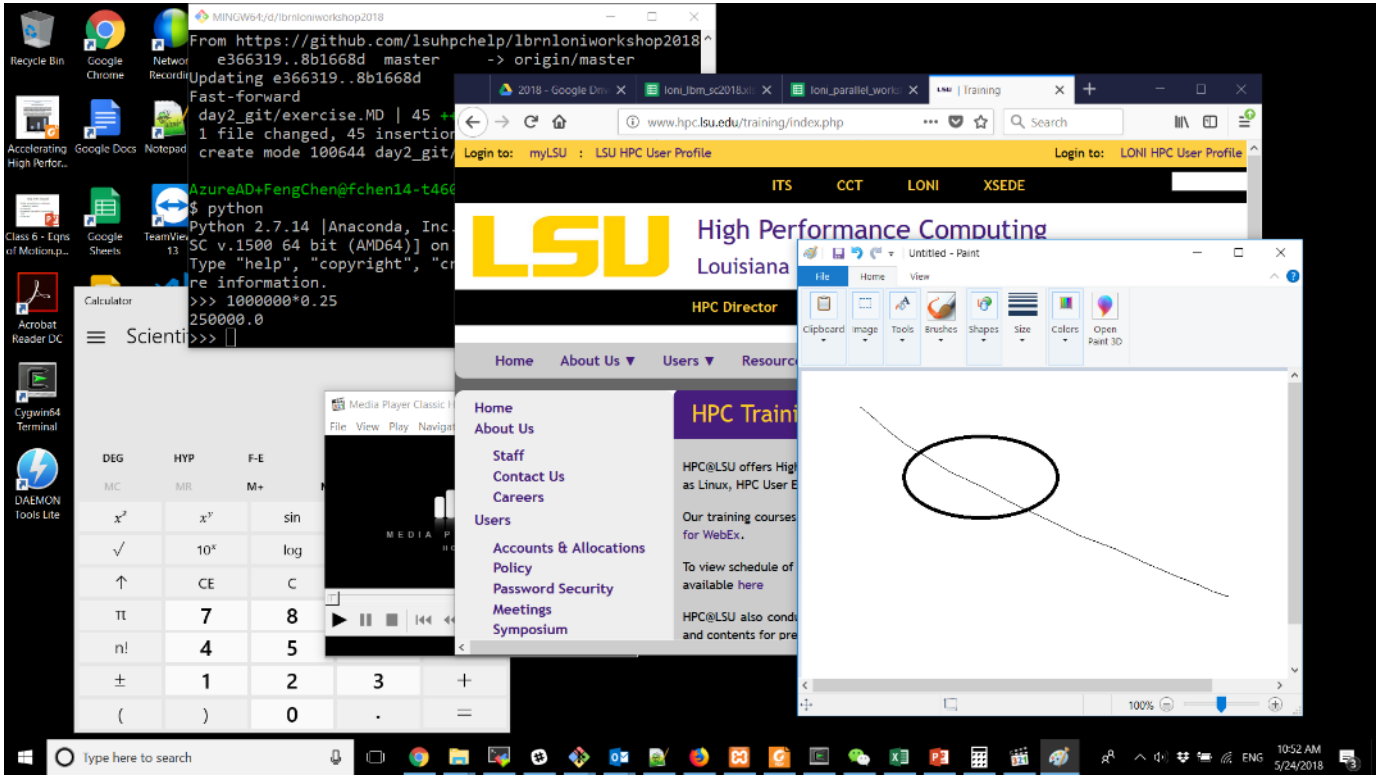
- **Flynn's taxonomy**
  - SIMD - GPU
  - MIMD - Multicore CPU
- **Shared Memory and Distributed Memory hardware systems**
  - Shared memory nodes:
    - NUMA
  - Distributed memory system:
    - Cluster

Parallel Programming Concepts

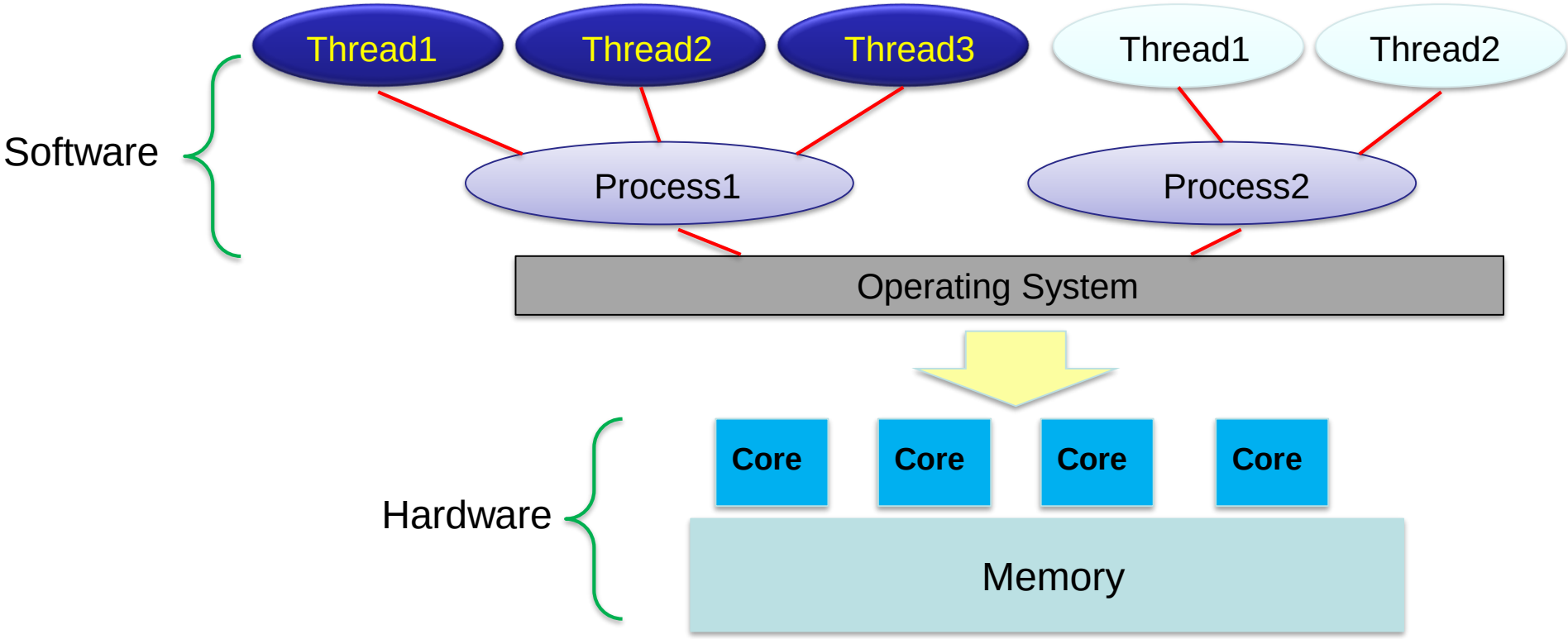
Parallel Software

Multitasking

- **Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources.**
- **Each CPU executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish.**

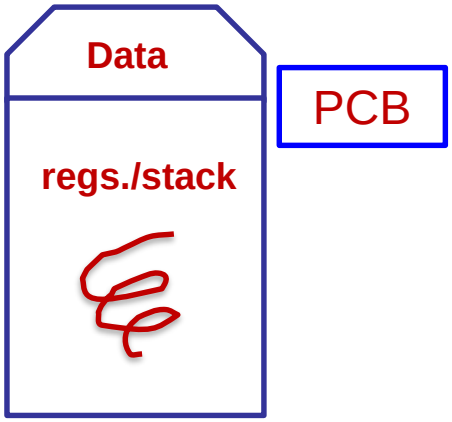


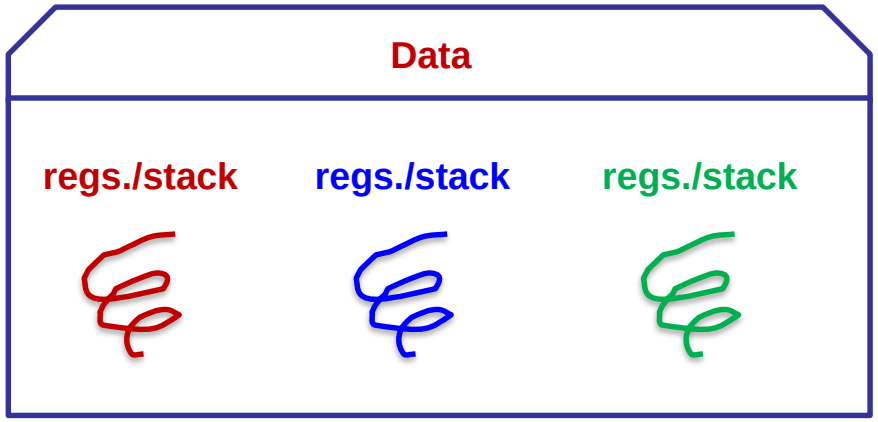
Process and threads

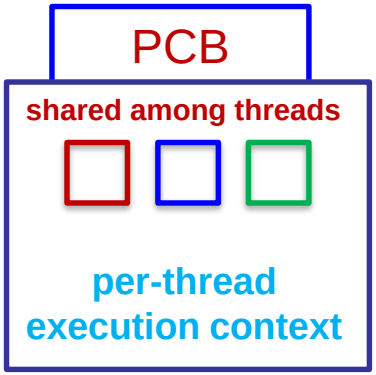


Process and Thread

- **An operating system "process" is basically an instance of program in execution**
  - Components of a process:
    - The executable machine language program.
    - Data
    - Process Control Block (PCB)
- **Threads are contained within processes:**
  - Thread is the smallest execution unit to which processor allocates time.
  - Components of a thread:
    - Program counter, stack, set of registers, A thread id







Difference Between Process/Threads

- **Process:**
  - A process is an instance of a computer program that is being executed. Each process has a complete set of its own variables.
- **Thread:**
  - A thread of execution results from a fork of a computer program into two or more concurrently running tasks. *In most cases, a thread is contained inside a process.*
  - A thread itself is not a program. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources.
- **The major difference:**
  - Multithreading threads are being executed in one process ***sharing common address space*** whereas in multi processing different processes have ***different address space***. Thus creating multiple processes is costly compare to threads.

The burden is on software

- **Hardware and compilers can keep up the pace needed.**
- **From now on…**
  - In shared memory programs:
    - Start a single process and fork threads.
    - Threads carry out tasks.
  - In distributed memory programs:
    - Start multiple processes.
    - Processes carry out tasks.

SPMD – single program multiple data

- **A SPMD programs consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.**
  - Both data and task parallelism

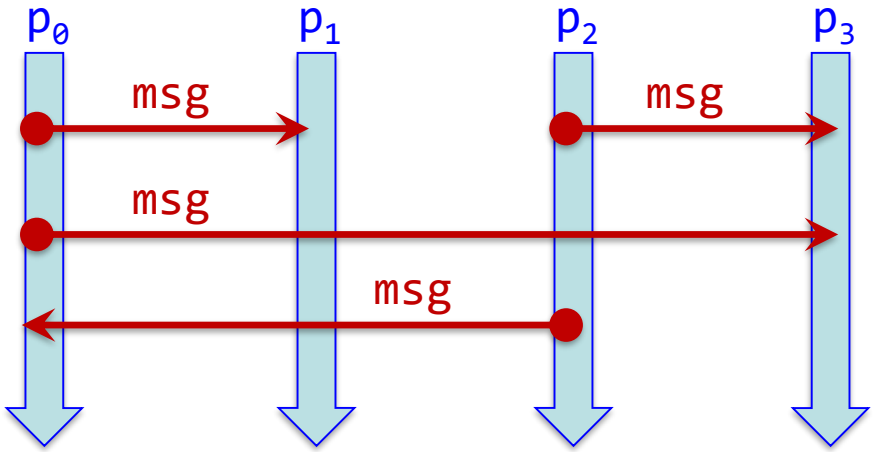
```
if (I'm thread or process i)
     do this;
else
     do that;
```

Parallel Programming Models

- **Shared memory programming model (Not covered in this year's workshop)**
  - OpenMP
  - Pthread
  - GPU
- **Distributed memory programming model**
  - ***MPI***
- **Hybrid Model (Not covered in this year's workshop)**
  - Distributed + Shared
    - MPI +
      - OpenMP
      - Pthread
      - GPU

Distributed Memory Programming

- **Distributed Memory Programming is process-based**
- **Processes running simultaneously communicate by exchanging messages**
  - Messages can be 2-sided - both sender and receiver are involved in the process
  - Or they can be 1-sided - only the sender or receiver is involved
- **Most message passing programs employ a mixture of data and task parallism**
- **Processes cannot access each other's memory**
  - Communication is usually explicit



Message Passing Interface (MPI)

- **MPI is a portable library used for writing parallel programs using the message passing model**
  - You can expect MPI to be available on any HPC platform you use
- **Based on a number of processes running independently in parallel**
  - HPC resource provides a command to launch multiple processes simultaneously (e.g. mpirun, aprun)
- **There are a number of different implementations but all should support the MPI 2 standard**
  - As with different compilers, there will be variations between implantations but all the features specified in the standard should work.
  - Examples: MVAPICH2, OpenMPI, MPICH2

Distributed Memory Prons and Cons

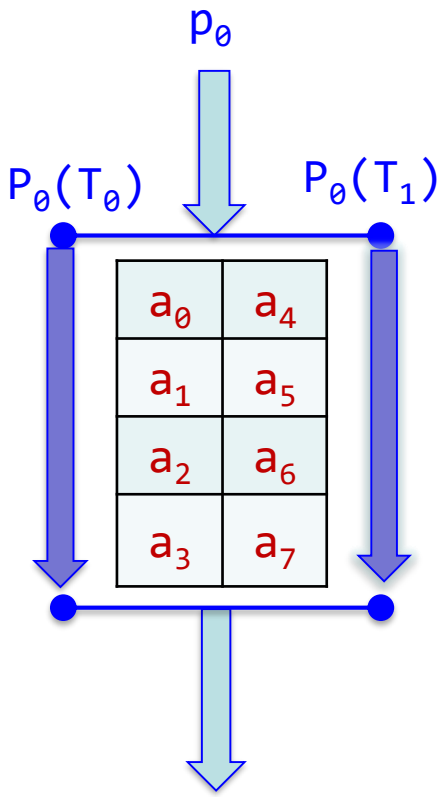
- **Advantages:**
  - Flexible - almost any parallel algorithm imaginable can be implemented
  - Scaling usually only limited by the choice of algorithm
  - Portable - MPI (Message Passing Interface) library is provided on all HPC platforms
- **Disadvantages:**
  - Parallel routines usually become part of the program due to explicit nature of communications
    - Can be a large task to retrofit into existing code
  - May not give optimum performance on shared memory machines
  - Can be difficult to scale to very large number of processes (>100,000) due to overheads

Shard Memory Programming

- **Shared memory programming is usually based on threads**
  - Although some hardware/software allows processes to be programmed as if they share memory
  - Sometimes known as Symmetric Multi-processing (SMP) although this term is now a little old-fashioned
- **Most often used for *Data Parallelism***
  - Each thread operates the same set of instructions on a separate portion of the data
- **More difficult to use for *Task Parallelism***
  - Each thread performs a different set of instructions

Shared Memory Concepts

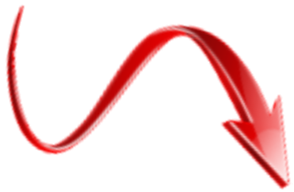
- **Threads "communicate" by having access to the same memory space**
- **Any thread can alter any bit of data**
- **No explicit communicate between the parallel tasks**



Nondeterminism

```
. . .
printf ( "Thread %d > my_val = %d\n", my_rank , my_x ) ;
. . .
```

```
Thread 1 > my_val = 19
Thread 0 > my_val = 7
```



```
Thread 0 > my_val = 7
Thread 1 > my_val = 19
```

Nondeterminism

```
my_val = Compute_val ( my_rank ) ;
x += my_val ;
```

Time	Core 0	Core 1
0	Finish assignment to `my_val`	In call to `Compute_val`
1	Load `x = 0` into register	Finish assignment to `my_val`
2	Load `my_val = 7` into register	Load `x = 0` into register
3	Add `my_val = 7` to `x`	Load `my_val = 19` into register
4	Store `x = 7`	Add `my_val` to `x`
5	Start other work	Store `x = 19`

Nondeterminism

- **Race condition**
- **Critical section**
- **Mutually exclusive**
- **Mutual exclusion lock (mutex, or simply lock)**

```
my_val = Compute_val ( my_rank ) ;
Lock(&add_my_val_lock ) ;
x += my_val ;
Unlock(&add_my_val_lock ) ;
```

Shared Memory Programming Models

- **Pthread**
  - POSIX Threads
- **OpenMP**
  - (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming
- **Threading Building Blocks**
  - A C++ template library developed by Intel for parallel programming on multi-core processors.
- **Cilk**
  - Cilk, Cilk++ and Cilk Plus are general-purpose programming languages designed for multithreaded parallel computing.
- **OpenCL**
  - OpenCL is an open standard that can be used to program CPUs, GPUs, and other devices from different vendors.
- **CUDA**
  - CUDA is a parallel computing platform and application programming interface (API) model created by NVidia.

Shared Memory Prons and Cons

- **Advantages:**
  - Conceptually simple
  - Usually minor modifications to existing code
  - Often very portable to different architectures
- **Disadvantages**
  - Difficult to implement task-based parallelism, lack of flexibility
  - Often does not scale well
  - Requires a large amount of inherent data parallelism (e.g. large arrays) to be effective
  - Can be surprisingly difficult to get performance

Parallel Programming Concepts

Performance

Take Timing

- **In order to determine *the run time* of a code, we usually need to include calls to a timer function in our source code.**
- **We are more interested in wall clock time, not CPU time**
  - The program may be “active”-for example, waiting for a message-even when the CPU is idle.
- **To take parallel times:**
  - synchronize the processes/threads before starting the timer,
  - after stopping the timer, find the maximum elapsed time among all the processes/threads.
- **Because of system variability, we usually need to run a program several times with a given data set,**
  - Usually take the ***minimum time*** from the multiple runs.
- **To reduce variability and improve overall run-times**
  - Usually run no more than one thread per core.

Typical Timing Functions

- **Unix, Linux System Call**

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

- **OpenMP**

```
double omp_get_wtime(void);
```

- **MPI**

```
double MPI_Wtime(void)
```

- **Typical code segments:**

```
double start_time = get_current_time_function();
// barrier funtion if needed
// code segments to be timed
double time_elapsed = get_current_time_function() - start_time;
```

- **In Bash:**

```
TIC=$(date +%s.%N);
# your code to be timed ...
TOC=$(date +%s.%N);
DURATION=$(echo "$TOC - $TIC" | bc -l)
```

Timing Example

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>

int main(char *argc, char **argv) {
    const long N = 1000000000;
    double sum=0.0;
    long i;
    // openmp directives
    double start_time = omp_get_wtime();
    #pragma omp parallel for reduction(+:sum)
    for (i=0;i<N;i++)
        sum += i*2.0+i/2.0; // doing some floating point operations
    double time = omp_get_wtime() - start_time;
    printf("time= %4.3e\n", time);
    printf("sum= %4.3e\n", sum);
    return 0;
}
```

Speedup

- **Number of cores = p**
- **Serial run-time = T_{serial}**
- **Parallel run-time = $T_{parallel}$**
- **Speedup of a parallel program**

Not the time from running a parallel code using one thread/process

$$S = \frac{T_{serial}}{T_{parallel}}$$

- **Linear speedup:**

$$T_{parallel} = \frac{T_{serial}}{p}$$

Efficiency Of A Parallel Program

- **Definition of efficiency:**

$$E = \frac{S}{p} = \frac{\left(\frac{T_{serial}}{T_{parallel}} \right)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

Speedups And Efficiencies Of A Parallel Program - Results

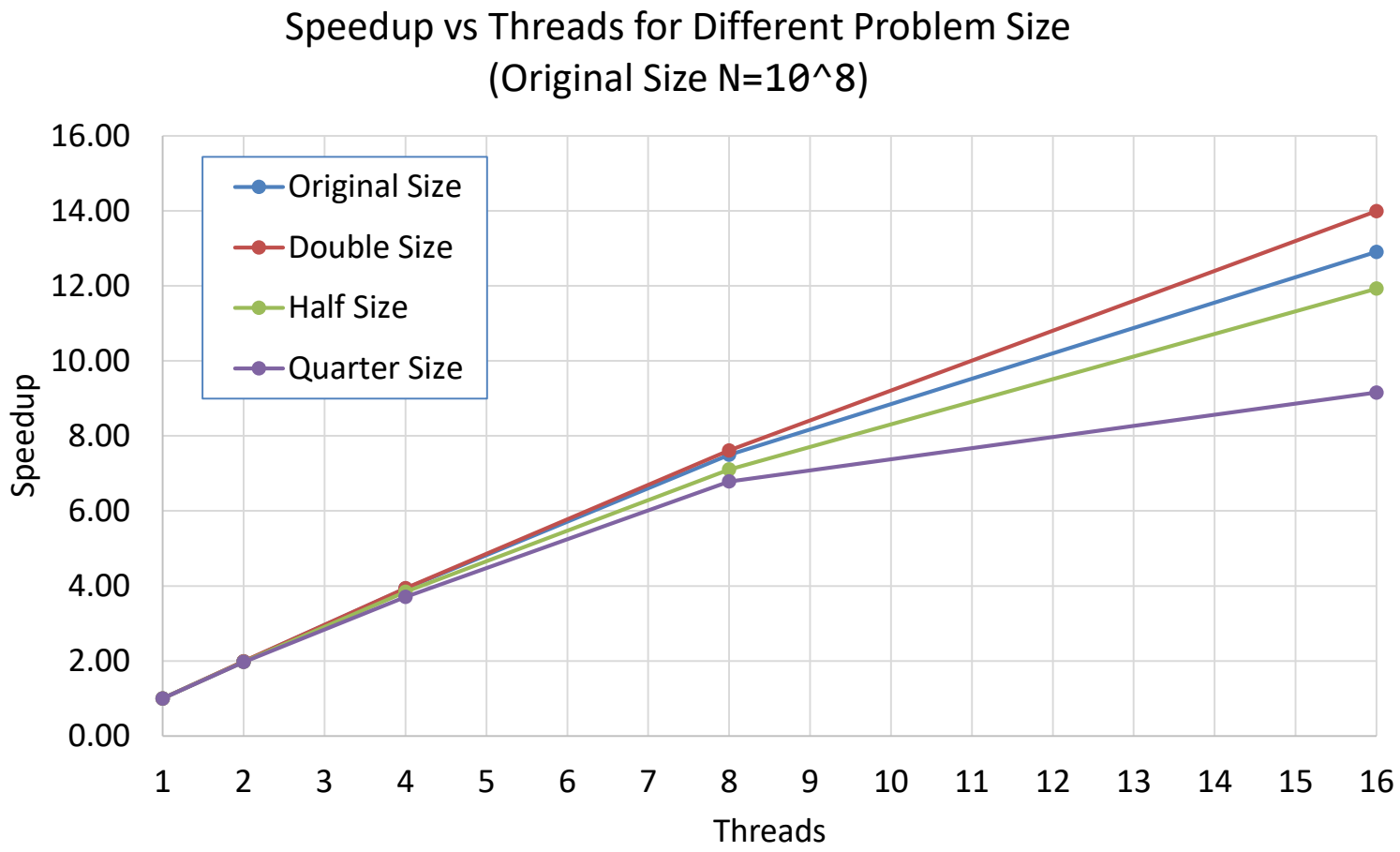
- **Example Parallel Program**

p	1	2	4	8	16
T	2.74	1.38	0.70	0.37	0.21
S	1.00	1.99	3.93	7.50	12.91
E=S/p	1.00	0.99	0.98	0.94	0.81

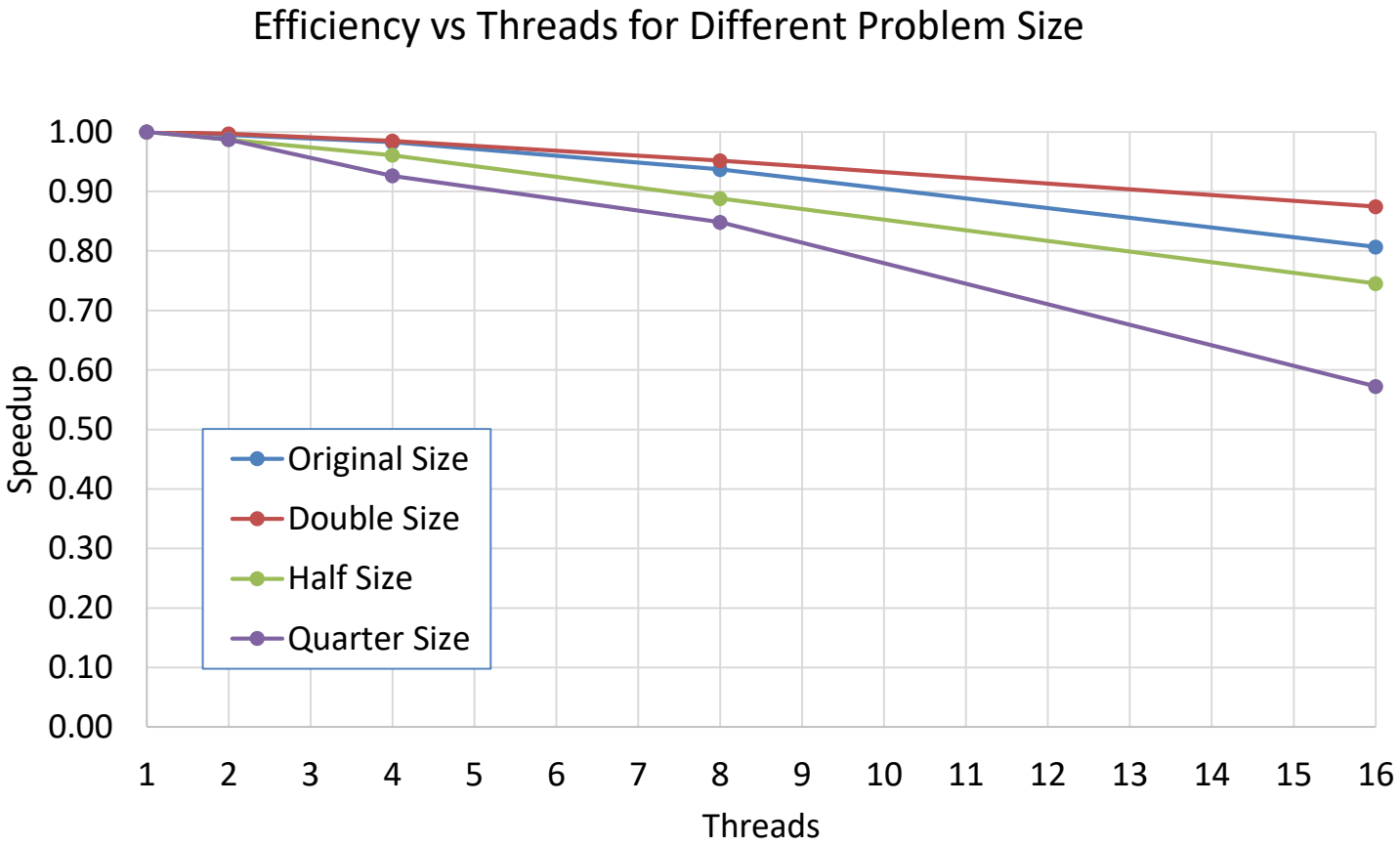
Speedups And Efficiencies On Different Problem Sizes

	p	1	2	4	8	16
Original N=1.0E+09	T	2.74	1.38	0.70	0.37	0.21
	S	1.00	1.99	3.93	7.50	12.91
	E=S/p	1.00	0.99	0.98	0.94	0.81
Double N=2.0E+09	T	5.47	2.74	1.39	0.72	0.39
	S	1.00	1.99	3.94	7.61	13.99
	E=S/p	1.00	1.00	0.98	0.95	0.87
Half N=5.0E+08	T	1.37	0.69	0.36	0.19	0.11
	S	1.00	1.97	3.84	7.10	11.93
	E=S/p	1.00	0.99	0.96	0.89	0.75
Quarter N=2.5E+08	T	0.69	0.35	0.19	0.10	0.07
	S	1.00	1.97	3.71	6.78	9.16
	E=S/p	1.00	0.99	0.93	0.85	0.57

Speedup


Speedup vs Threads for Different Problem Size
(Original Size N=10^8)
Speedup
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Original Size
Double Size
Half Size
Quarter Size
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Threads

Efficiency


Efficiency vs Threads for Different Problem Size
Speedup
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Threads
Original Size
Double Size
Half Size
Quarter Size

Effect of Overhead

- **Many parallel programs are developed by dividing the work of the serial program among the processes/threads and adding in the necessary "parallel overhead" such as mutual exclusion or communication.**
- ***$T_{overhead}$* denotes the parallel overhead:**

$$T_{parallel} = \frac{T_{serial}}{p} + T_{overhead}$$

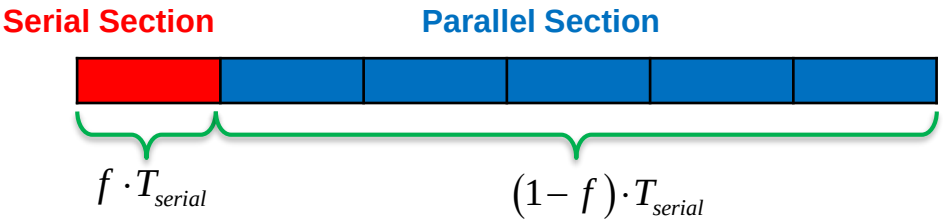
- **To increase Speedup and Efficiency with increasing problem size, what kind of condition should be satisfied for *T_{serial}* and *$T_{overhead}$*?**
  - See Exercise 2b)

Scalability of Parallel Systems

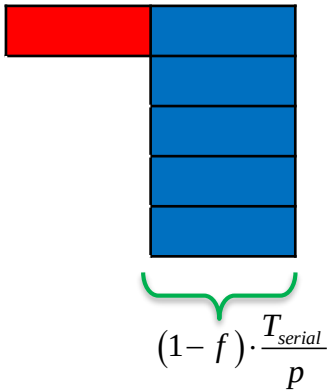
- **Amadahl's Law**
  - The size of the sequential fraction ***f*** determines how much speedup is achievable

$$S \le \frac{T_{serial}}{f \cdot T_{serial} + (1-f) \cdot \frac{T_{serial}}{p}} = \frac{p}{1 - f(1-p)} = \frac{1}{f + \frac{(1-f)}{p}}$$

  - One processor

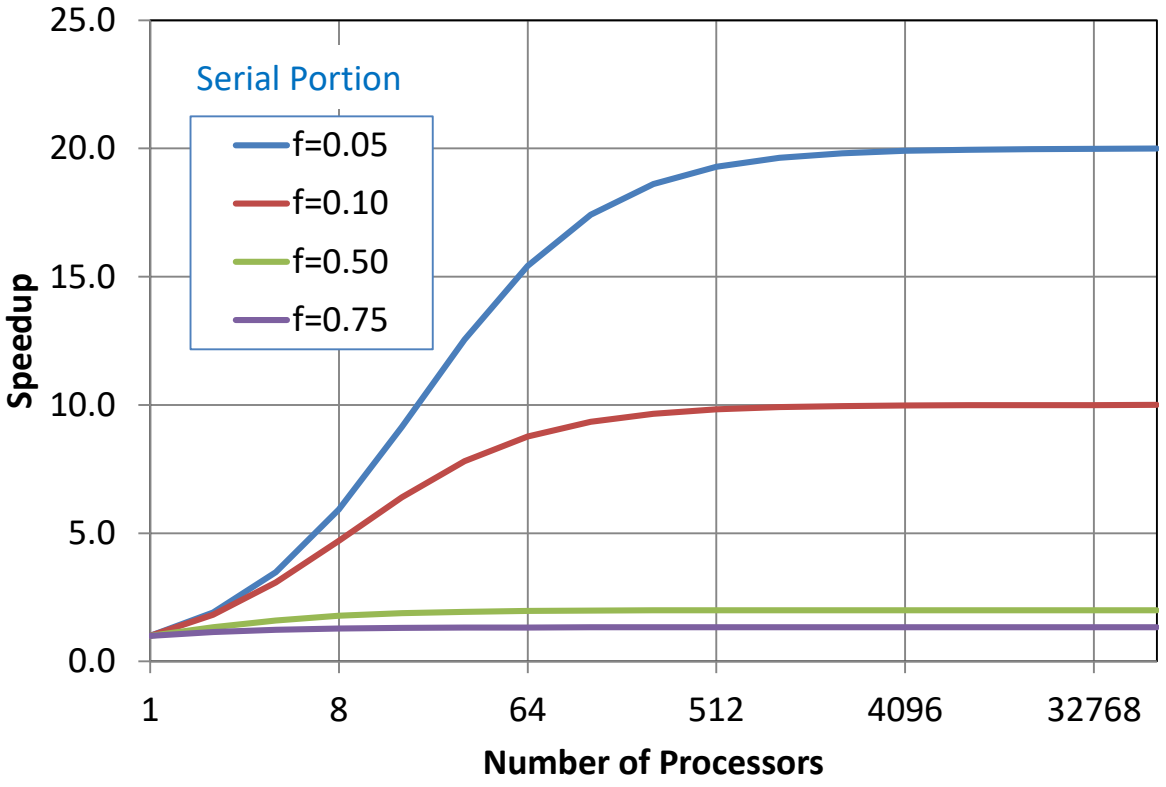


  - *p* processors



Effect of Amdahl's Law

- **Plot of Speedup vs Number of processors at different *f* (parallel portion)**



Amdahl's Law

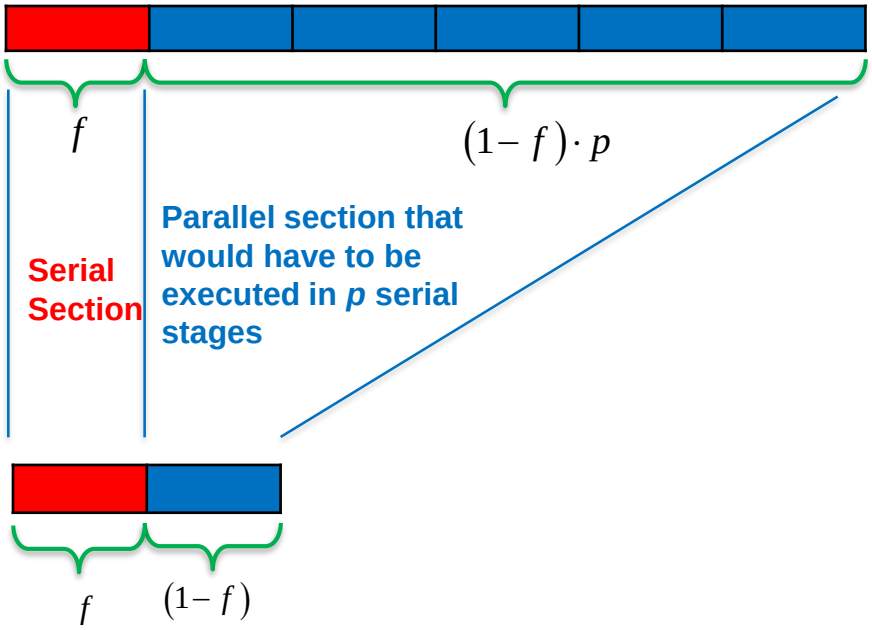
- **Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.**

- **This is pretty daunting. Should we give up and go home?**

Gustafson's Law

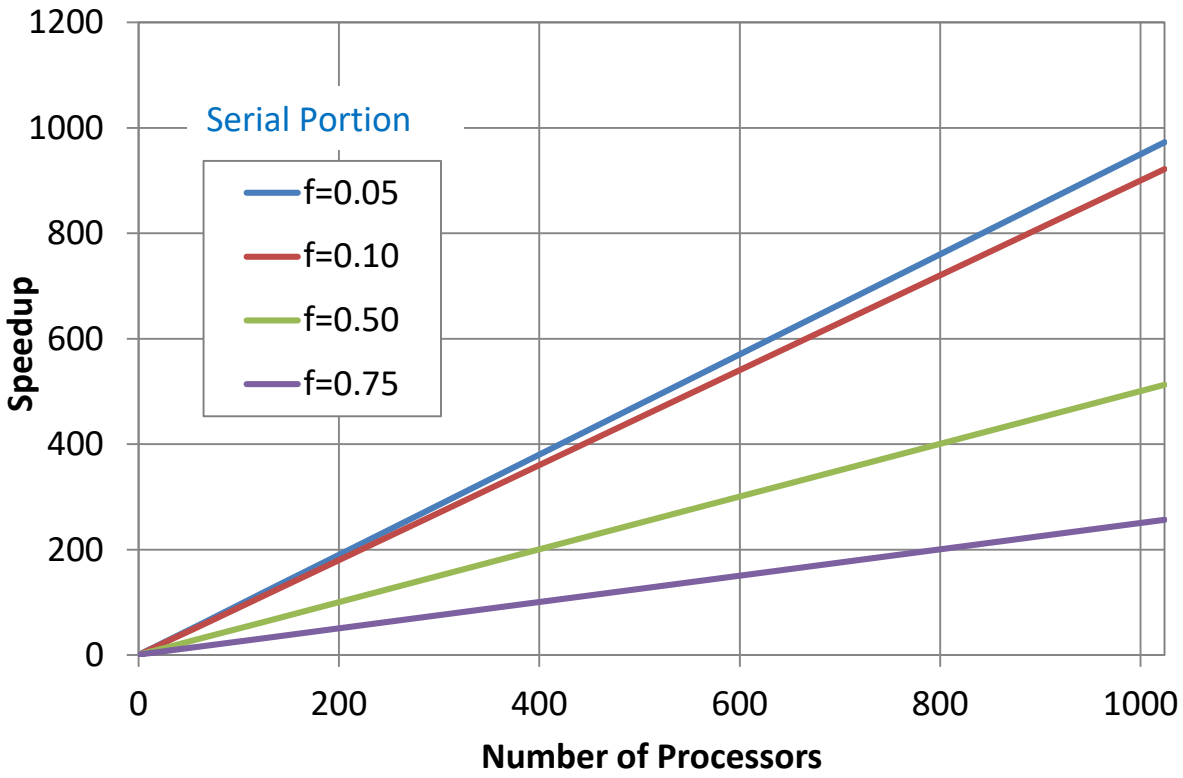
- **Addresses the shortcomings of Amdahl's law, which is based on the assumption of a fixed problem size.**
  - Apply *p* processors to a task that has serial fraction ***f*** , scaling the task to take the same amount of time as before, the speedup is:

$$S = f + (1 - f) \cdot p = p - f\,(p - 1)$$



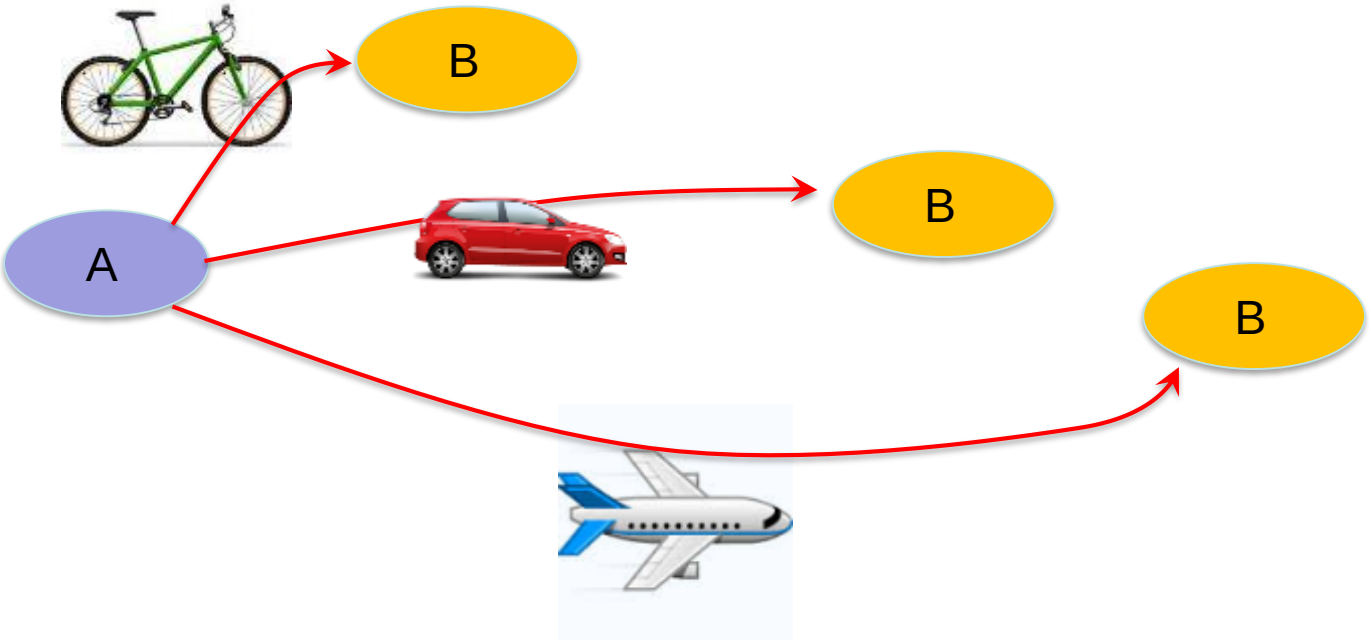
Effect of Gustafson's Law

- **Plot of Speedup vs Number of processors at different f (serial portion)**



Gustafson's Law - Analogy

Commuting Time:

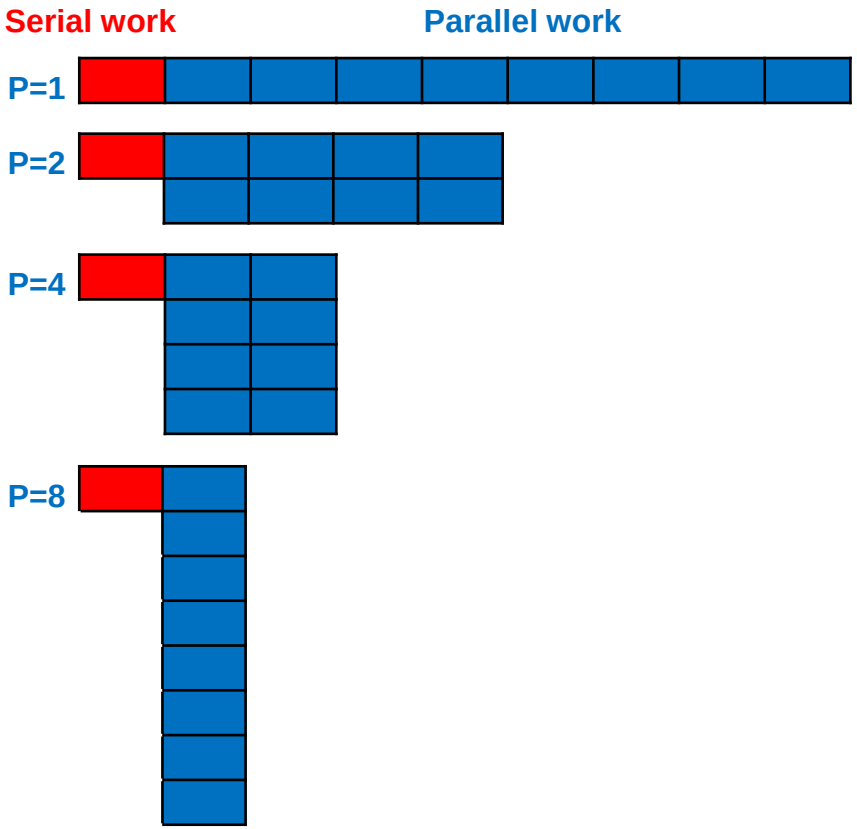


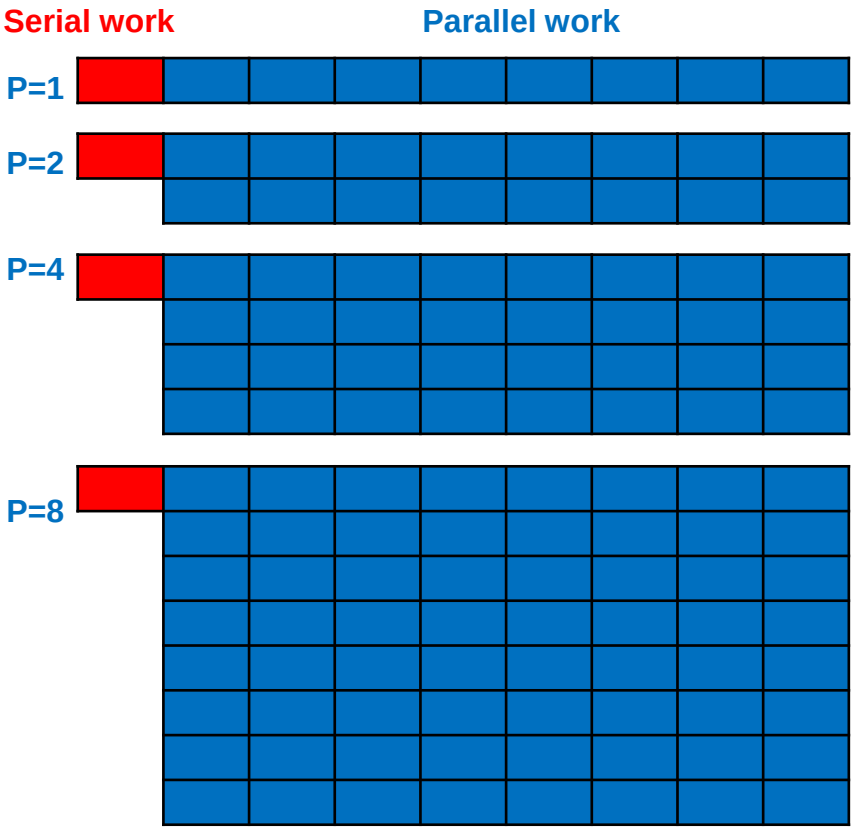
Biological Brains:

v.s.

Amdahl v.s. Gustafson

- **Amdahl's law based on fixed problem size.**
- **Gustafson's law based on fixed run time.**





Strong and Weak Scaling

- **If we can find a corresponding rate of increase in the problem size so that the program always has efficiency E, then the program is *scalable*.**
  - Strong Scaling: Keeping the problem size fixed and pushing in more workers or processors
    - Goal: Minimize time to solution for a given problem
  - Weak Scaling: Keeping the work per worker fixed and adding more workers/processors (the overall problem size increases)
    - Goal: solve the larger problems

Scalability Example

- **Serial execution:** $T_{serial} = n$
- **Parallel execution:** $T_{parallel} = \frac{n}{p} + 1$
- **Efficiency:** $E = \frac{T_{serial}}{p \cdot T_{parallel}} = \frac{n}{p(n/p + 1)} = \frac{n}{n + p}$
- **To see if the program is scalable, we increase the number of processors by a factor of *k*, and we want to find the factor *x* that we need to increase the problem size by so that *E* is unchanged:**

$$E' = \frac{xn}{xn + kp} = E = \frac{n}{n + p}$$

- **Solve the above equation, we have:**

$$x = k$$

- **Thus the program is scalable (Efficiency will be unchanged)**

Parallel Programming Concepts

Designing Parallel Programs

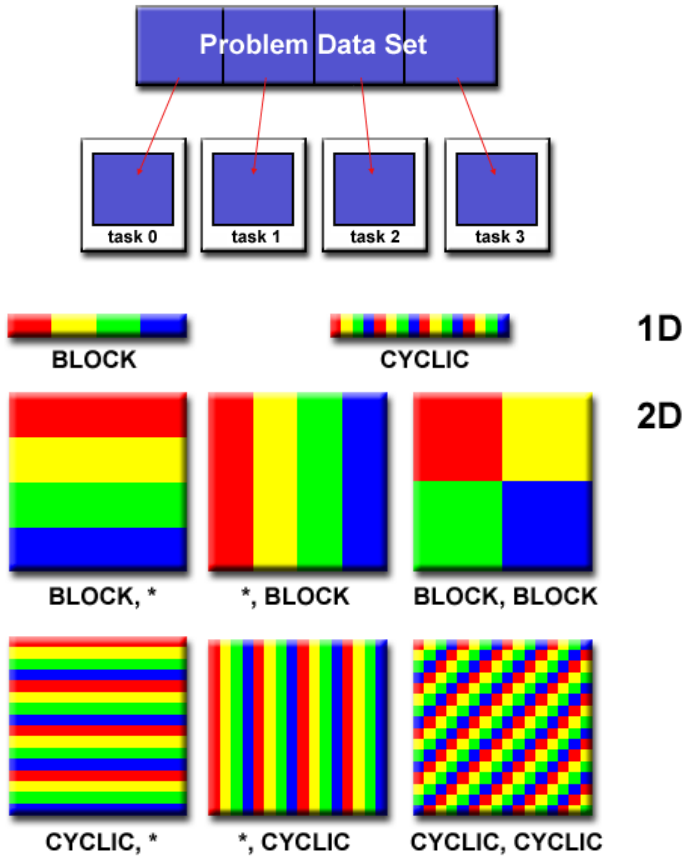
Design a Parallel Program

- **How to design a parallel program**
  - Partitioning
    - Divide the work among the processes/threads so that
      - Each process/thread gets roughly the same amount of work;
      - Communication is minimized.
  - Communication
    - Arrange for the processes/threads to synchronize.
  - Synchronization
    - Arrange for communication among processes/threads

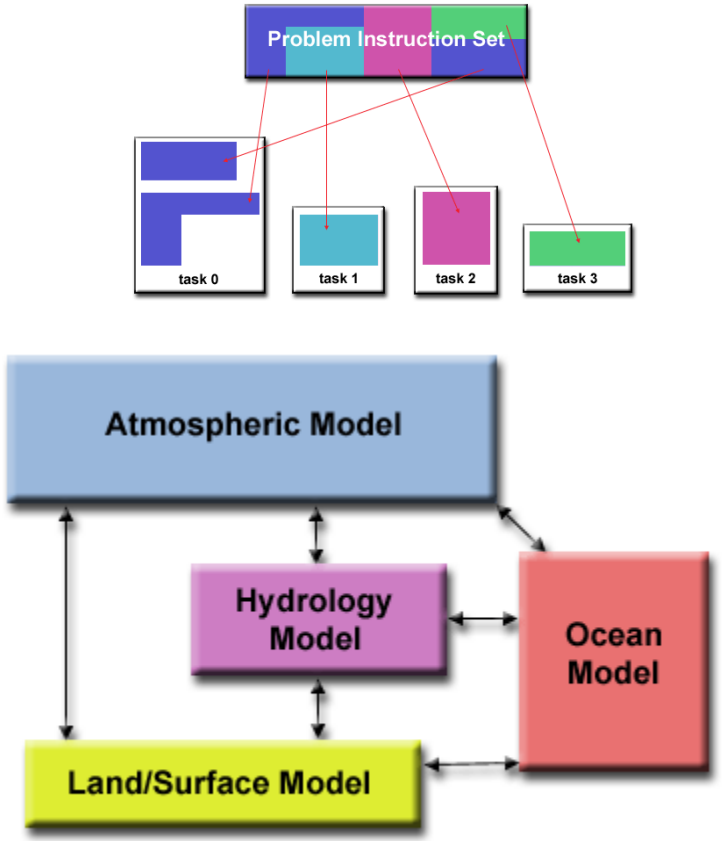
Partitioning

- **There are two basic ways to partition computational work among parallel tasks:**

domain decomposition

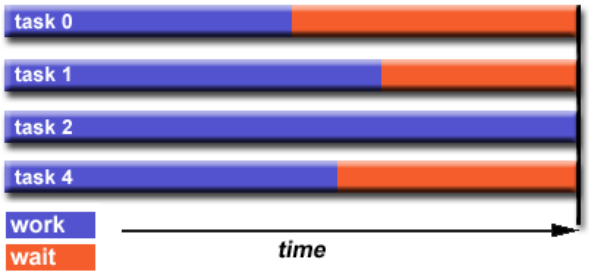


functional decomposition



Load Balancing

- **Load balancing**
  - Distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time.
  - Important to parallel programs for performance reasons.



- **How to Achieve Load Balance:**
  - Equally partition the work each task receives, e.g.:
    - For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
    - For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
  - Use dynamic work assignment
    - It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

Communications

- **Factors to Consider:**
  - Cost of communications
  - Latency vs. Bandwidth
    - latency is the time it takes to send a minimal (0 byte) message from point A to point B.
    - bandwidth is the amount of data that can be communicated per unit of time.
  - Synchronous vs. asynchronous communications
  - Scope of communications
    - Point-to-point
    - Collective
  - Efficiency of communications
  - Overhead and Complexity

- **Overall, try to minimize the communication!**

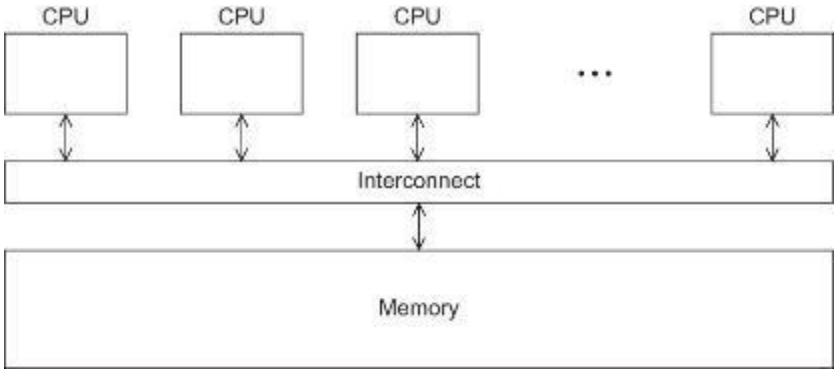
Synchronization

- **Managing the sequence of work and the tasks performing it is a critical design consideration for most parallel programs.**
  - Can be a significant factor in program performance (or lack of it)
  - Often requires "serialization" of segments of the program.

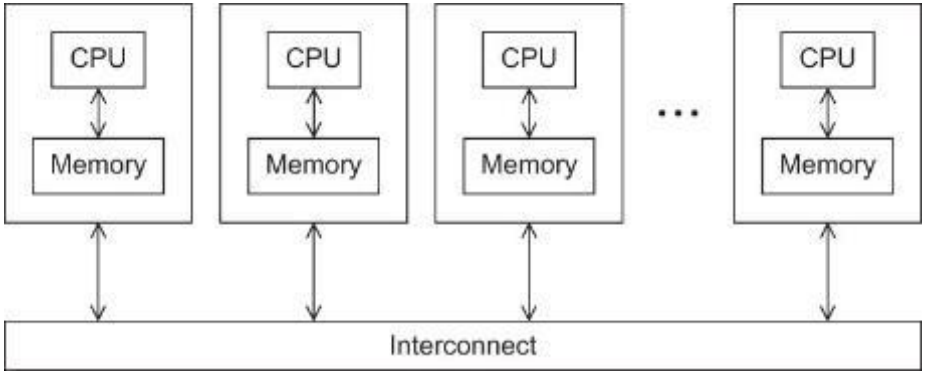
- **Types of Synchronization:**
  - Barrier
  - Lock/semaphore
  - Synchronous communication operations

Communication and Synchronization

- **In shared-memory programs, we often implicitly communicate among the threads by synchronizing them.**
- **In distributed-memory programs, we often implicitly synchronize the processes by communicating among them.**



A shared-memory system



A distributed-memory system

Questions?

Parallel Programming Concepts

Lab and Exercise Session

Quiz

- **Quiz link**
  - https://goo.gl/forms/pokKf4suYVosi2MH3

Exercise 0 - Log onto Cluster

- **Open a terminal and log onto SuperMike2, download the git repository, compile the examples:**

```
# log onto SuperMike2, hpctrn?? is your training account
[user@locallaptop ]$ ssh -X hpctrn??@mike.hpc.lsu.edu
# submit an interactive job to shelob queue
[hpctrn01@mike2]$ qsub -I -X -l nodes=1:ppn=16,walltime=8:00:00 -q shelob -A hpc_train_2018
qsub: waiting for job 658202.mike3 to start
...job welcome message...
[hpctrn01@shelob001 ~]$ git clone https://github.com/lsuhpchelp/loniworkshop2018.git
Initialized empty Git repository in /home/hpctrn01/loniworkshop2018/.git/
...
Unpacking objects: 100% (47/47), done.
[hpctrn01@shelob001 ~]$ cd loniworkshop2018/day1morning/exercise/
# build all executables
[hpctrn01@shelob001 exercise]$ make
icc -openmp -I./ floatoper_omp_ser.c -o eg_serial.out
...
mpicc hello_mpi.c -o hello_mpi.out
# check python version
[hpctrn01@shelob001 exercise]$ which python
/usr/local/packages/python/3.6.4-anaconda/bin/python
```

Exercise 1

- **Compile and run OpenMP, MPI programs**
  - There is an example OpenMP and MPI programs provided for you, compile the two programs using the intel (recommended) or gcc compiler

a) Observe the nondeterminism of the thread/process output

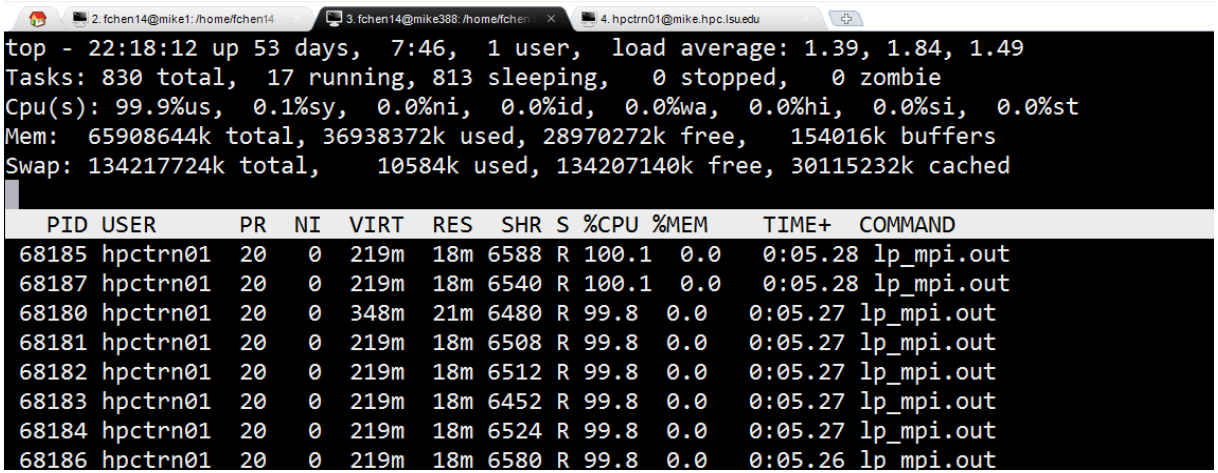
```
# compile and run the hello world openmp code
$ icc -openmp hello_omp.c -o  hello_omp.out
$ ./hello_openmp.out
# compile and run the hello world mpi code
$ mpicc hello_mpi.c -o hello_mpi.out
$ mpirun -np 16 ./hello_mpi.out
```

b) Observe the running multiple processes/threads using the “top” utility (with the H option for openmp codes)

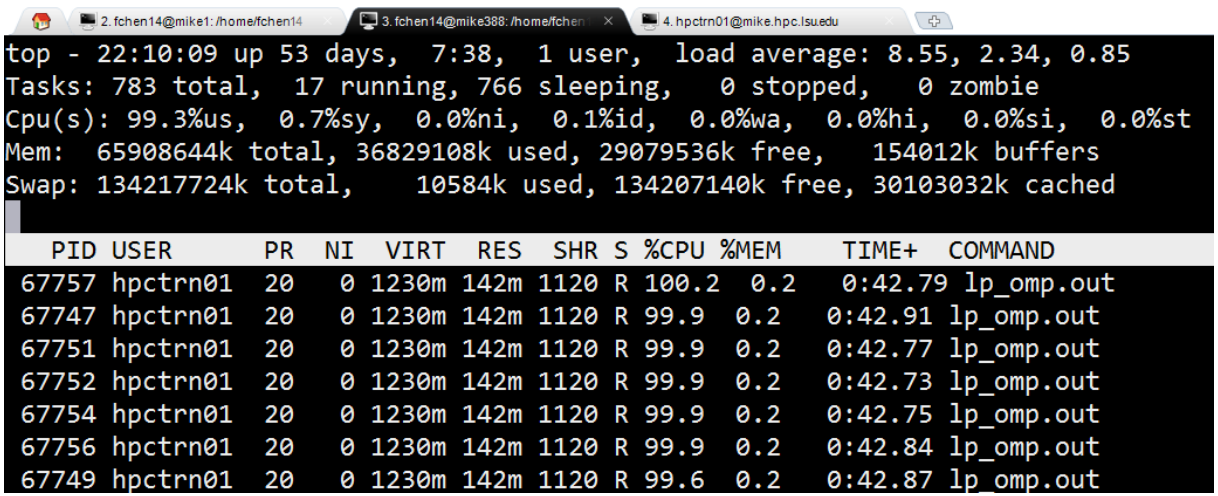
```
# run the mpi laplacian solver
$ ./run_lp_mpi.sh
# open another terminal and ssh to the compute node, then type “top”
# run the openmp laplacian solver
$ ./run_lp_omp.sh
# open another terminal and go to the compute node, then type “top -H”
```

Exercise 1

- b) Expected results from the “top”
  - For the MPI version, you should see 16 processes:



  - For the OpenMP version, you should see 16 threads (using `top -H`):



Exercise 2 Performance

- **a)**
  - Suppose the run-time of a serial program is given by $T_{serial} = n^2$, where the units of the run-time are in microseconds. Suppose that a parallelization of this program has run-time

$$T_{parallel} = n^2 / p + \log_2 (p)$$

  - Write a program that plots the speedups and efficiencies of this program for various values of ***n*** and ***p***. Run your program with ***n* = 10, 20, 40, 80, 160, 320**, and ***p* = 1, 2, 4, 8, 16, 32, 64, 128**.
  - You can use the python template script provided for you (`ex2a.py`), complete the portion (in `FIXME`) that calculates the Speedup ***S*** and Efficiency ***E***, or you can use your most comfortable plotting tool), based on the plot, observe:
    - What happens to the speedups and efficiencies as p is increased and n is held fixed?
    - What happens when p is fixed and n is increased?

Exercise 2 Performance

- **b)**
  - Suppose that T_{serial} and $T_{overhead}$ are both functions of the problem size ***n***:

$$T_{parallel} = T_{serial} / p + T_{overhead}$$

  - Also suppose that we fix ***p*** and increase the problem size ***n***.
    - To increase Speedup and Efficiency with increasing problem size, what condition should be satisfied for T_{serial} and $T_{overhead}$?

Exercise 3 - Scalability (*Optional*)

- **Compile and run the OpenMP version of the floating point sum operation (`floatoper_omp_par.c`). Perform strong scaling and weak scaling test using this code:**
  - For strong scaling, the Efficiency E:

$$E_{strong} = T_{serial} \Big/ \left(p \cdot T_{parallel} \right)$$

  - For weak scaling, the Efficiency E:

$$E_{weak} = T_{serial} \Big/ \left(T_{parallel} \right)$$

  - Try to change the problem size and see the difference in efficiency
  - Hint: You can use the bash scripts provided for you, and try to change the variable `BASE(the problem size N in the for loop)` in `perf_omp_strong.sh` and `perf_omp_weak.sh,` observe the Efficiency v.s. Number of processors curve at different problem size N:

```
[hpctrn01@shelob001 solution]$ ./perf_omp_strong.sh
[hpctrn01@shelob001 solution]$ ./perf_omp_weak.sh
[hpctrn01@shelob001 solution]$ ./ex3_perf_omp.py
```

Exercise 4 Load Balancing

- **An MPI code need to process *n* data elements (indexed from `0 to n-1`) using *p* processes (indexed from `0 to p-1`) , in order to achieve load balancing, each core (process) should be assigned roughly the same number of elements of computations.** `(loadbalance_mpi.c)`
  - Try to complete the code (in `FIXME`) so that each process (identified by `pid`) prints the start and end index of the elements, for the below example the output should be similar to:

```
To compile the program:
[hpctrn01@shelob001 exercise]$ mpicc loadbalance_mpi.c
# start 3 processes to process 13 elements
[hpctrn01@shelob001 exercise]$ mpirun -np 3 ./a.out 13
pid:0,  my_start=0,     my_size=5,      my_end=4
pid:1,  my_start=5,     my_size=4,      my_end=8
pid:2,  my_start=9,     my_size=4,      my_end=12
```