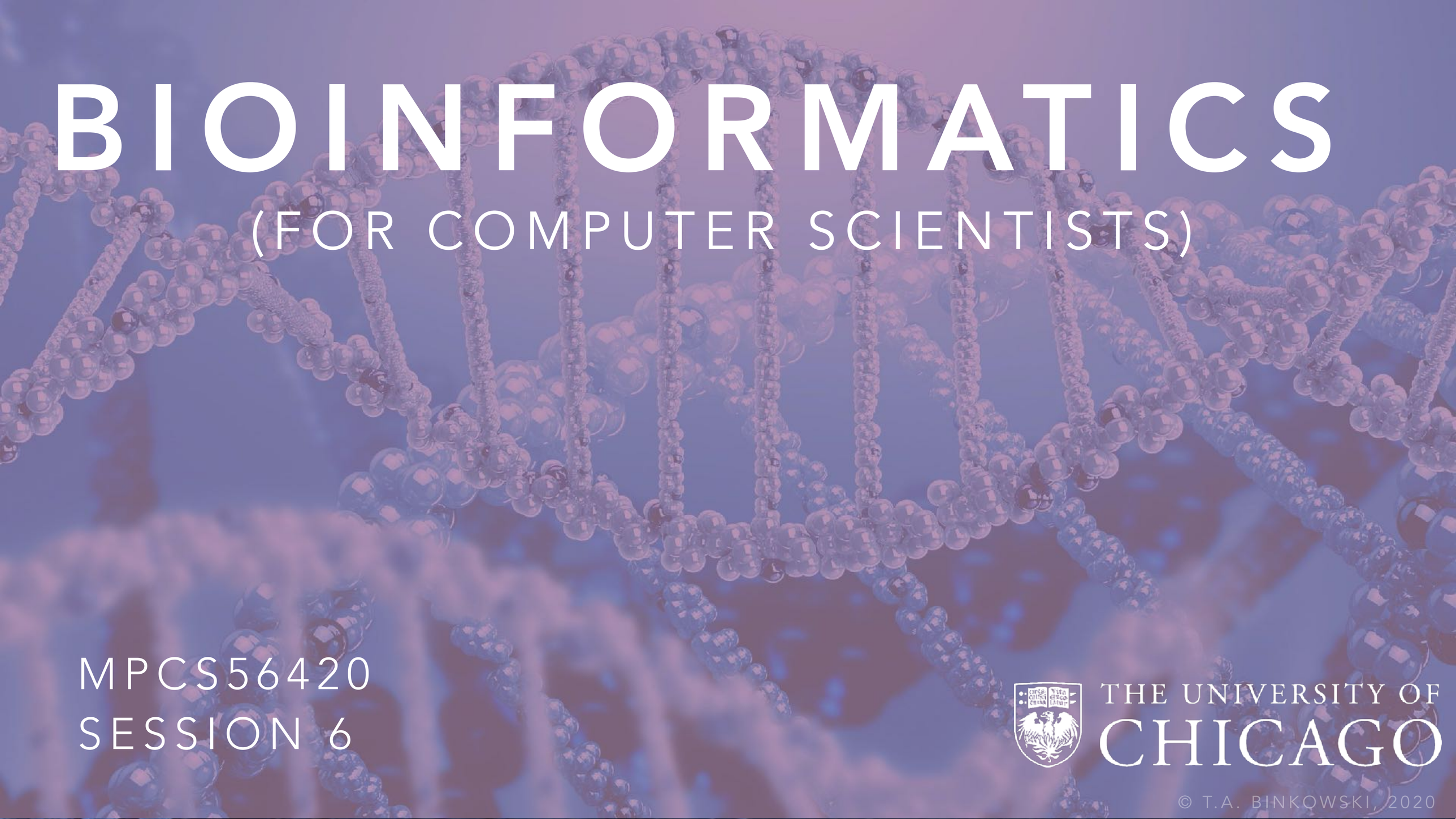
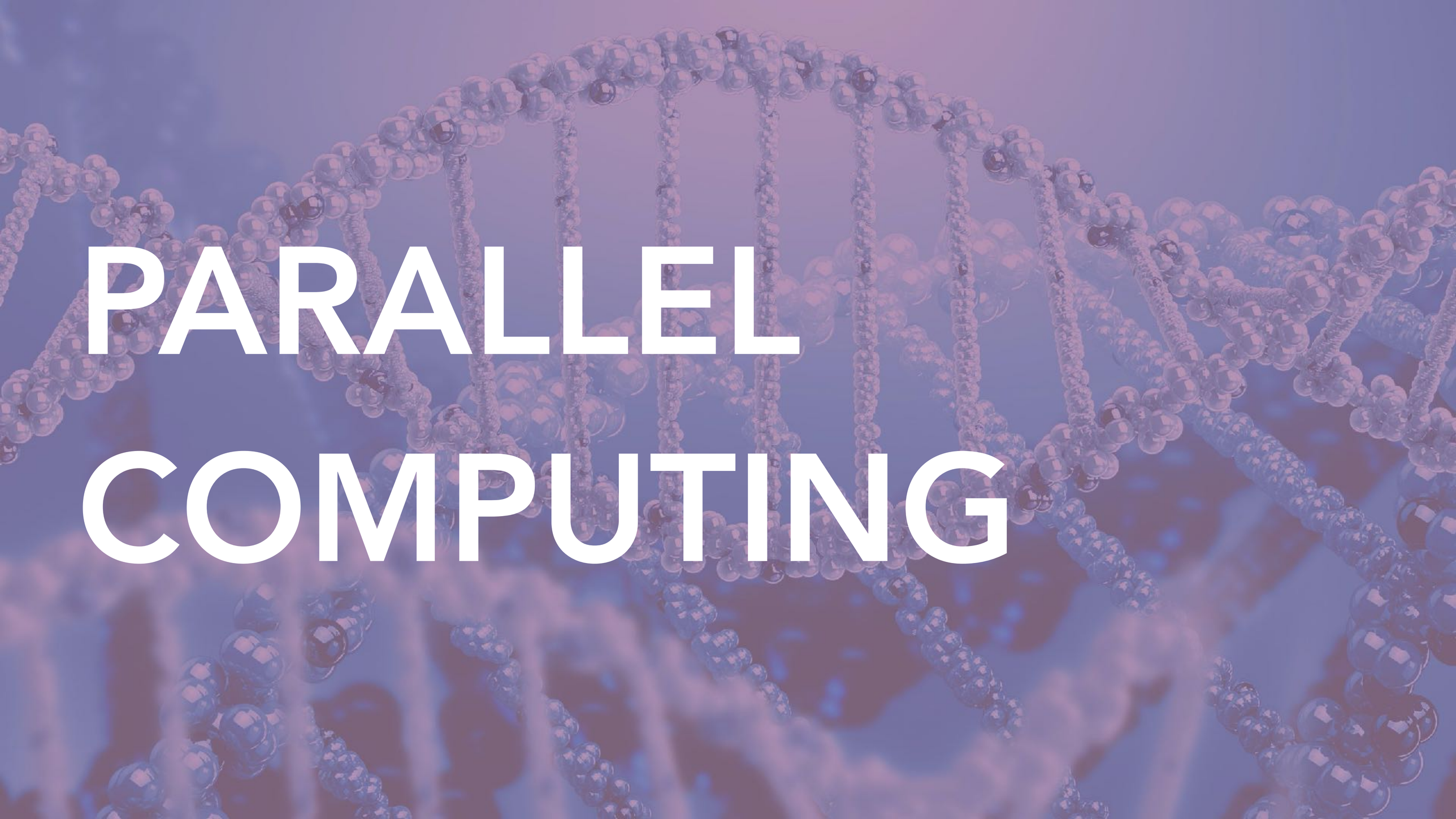


BIOINFORMATICS

(FOR COMPUTER SCIENTISTS)

MPCS56420
SESSION 6

THE UNIVERSITY OF CHICAGO


PARALLEL
COMPUTING

HPC

- My take
  - Clusters are great but unreliable
  - They are the only way to address some problems
  - Given the time scale of publication, experiments, etc. timely results can be obtained with loosely coupled machines and multicore systems
  - Reliances on big machines is dangerous
    - No time left to run
- Need strategies that take advantage of what resources you have!!!!

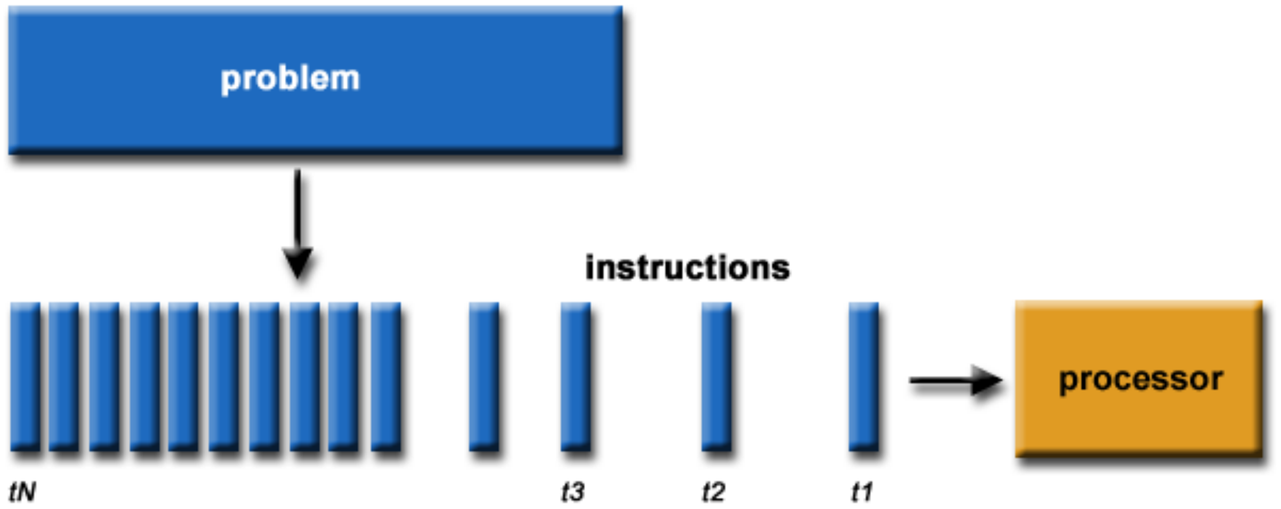
What is Parallel Computing?

BLAISE BARNEY, LAWRENCE LIVERMORE NATIONAL LABORATORY

WHAT IS PARALLEL COMPUTING?

SERIAL COMPUTING

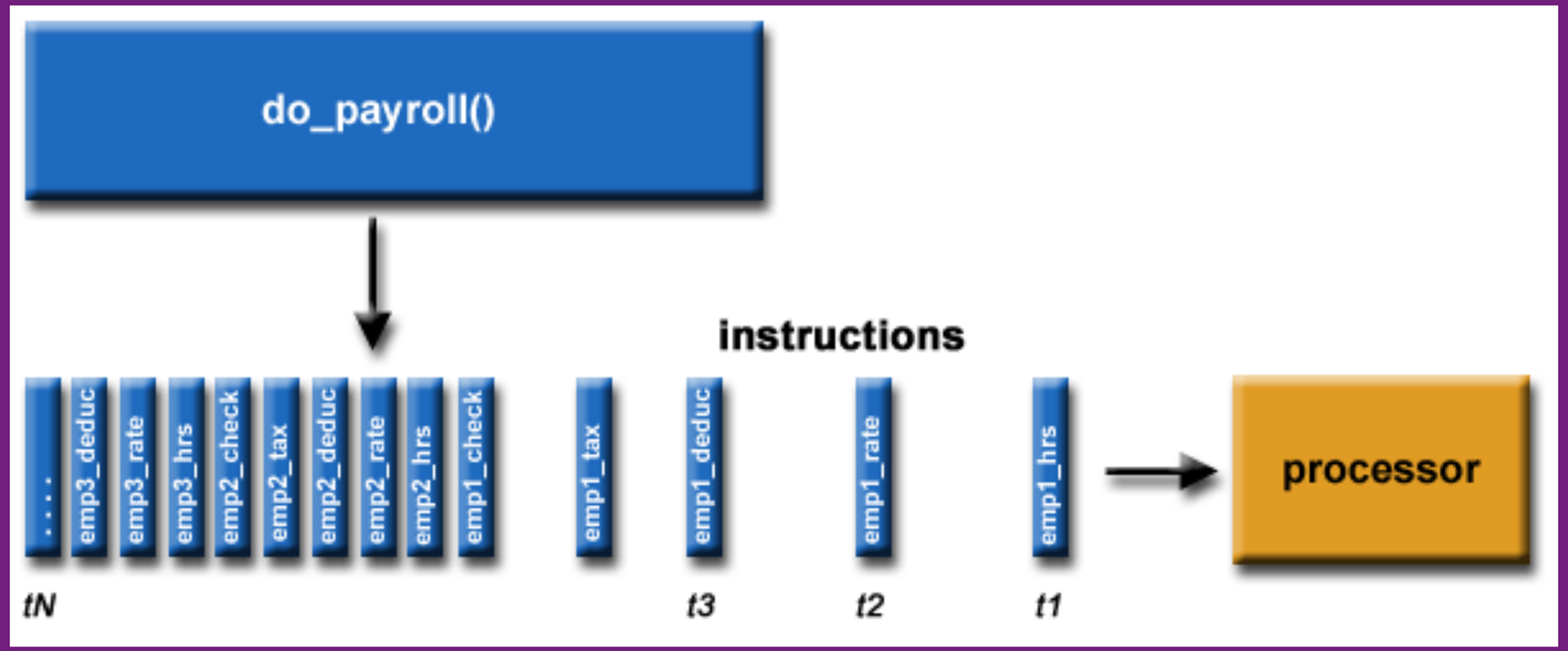
- Problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time



SERIAL COMPUTING

WHAT IS PARALLEL COMPUTING?

SERIAL COMPUTING

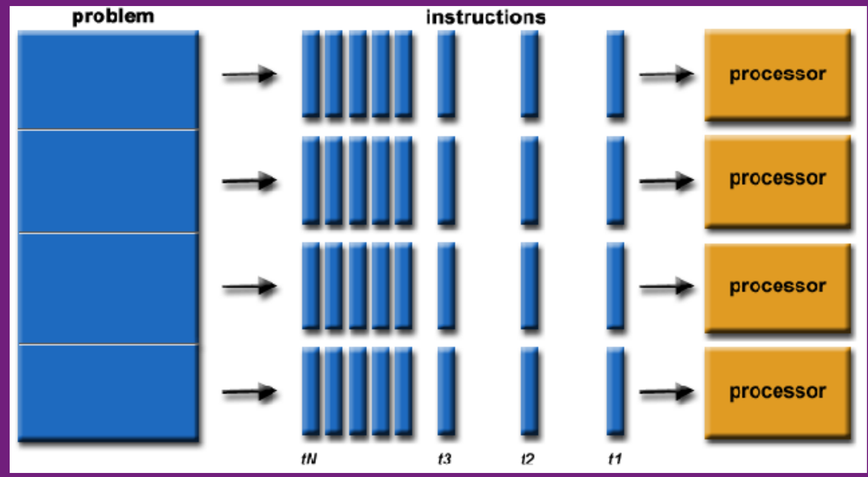


- Example: Doing payroll for employee

WHAT IS PARALLEL COMPUTING?

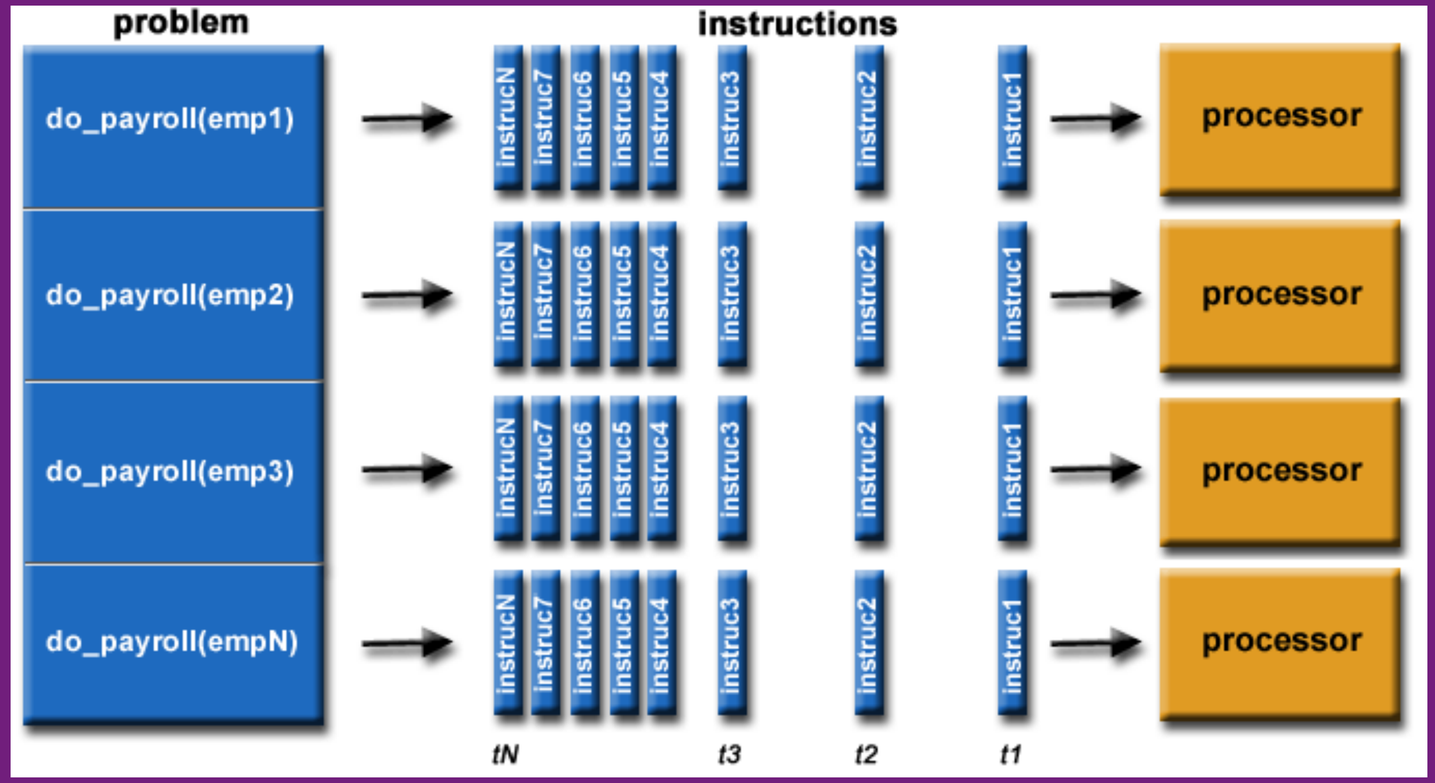
PARALLEL COMPUTING

- Simultaneous use of multiple compute resources to solve a computational problem
  - A problem is broken 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



WHAT IS PARALLEL COMPUTING?

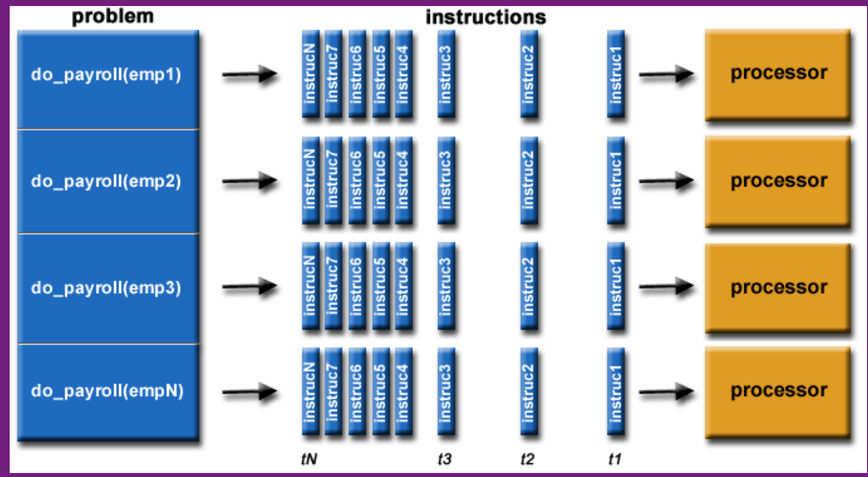
PARALLEL COMPUTING



- Example: Payroll for the entire company

WHAT IS PARALLEL COMPUTING?

- Parallel problems requirements
  - 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 PROBLEM DEFINES THE COMPUTING STRATEGY

Parallel Computers

PARALLEL COMPUTERS

- A single computer with multiple processors/cores
  - iPhone X - 4 cores
  - MacBook Pro - 16 cores
  - MacPro - 32 cores
- An arbitrary number of such computers connected by network
  - BlueGene/Q ~300,000 cores
  - Next generation BlueGene ~5 million cores

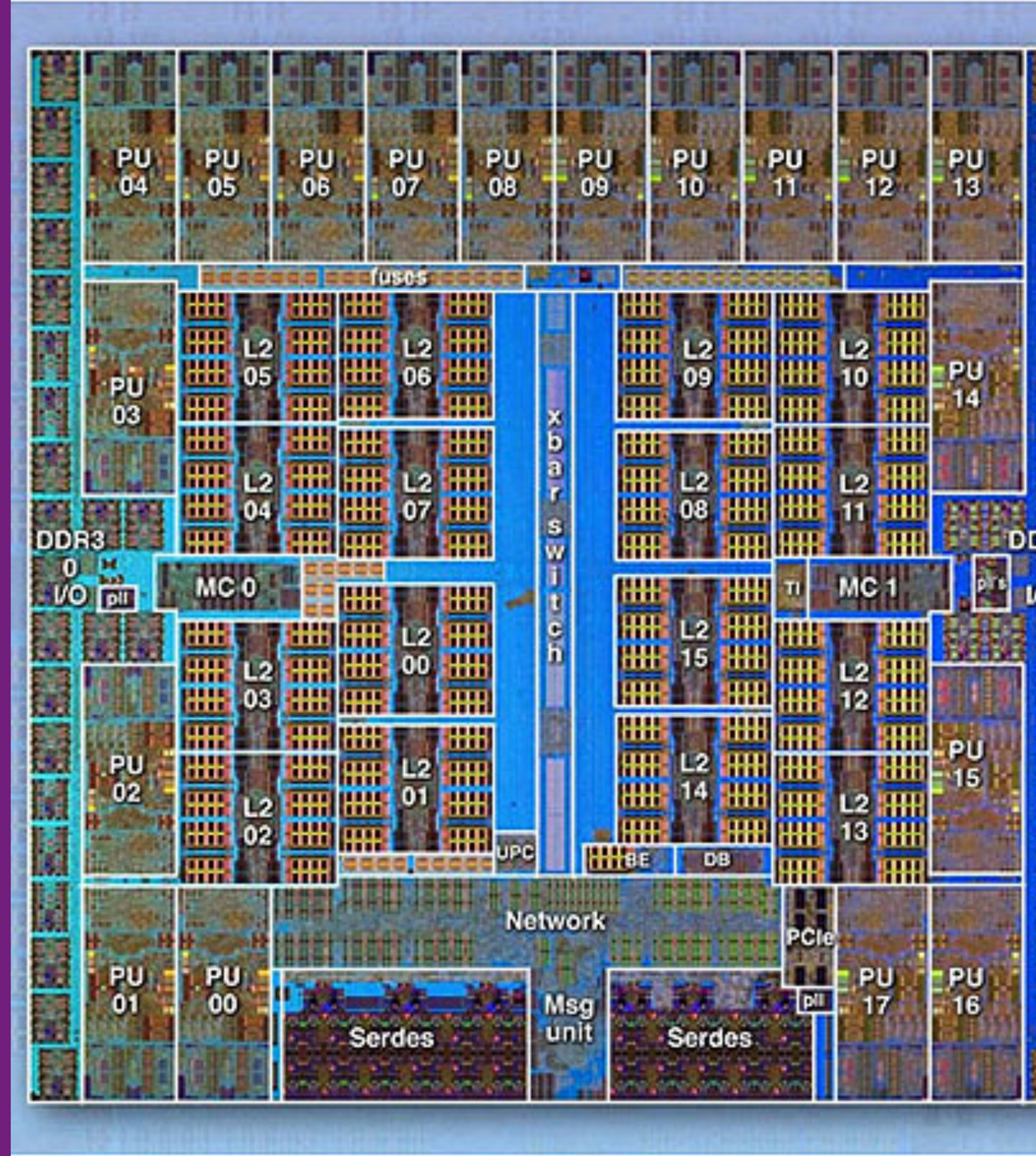
WHAT IS PARALLEL COMPUTING?

PARALLEL COMPUTING RESOURCES

8 CORE GALAXY NOTE 7

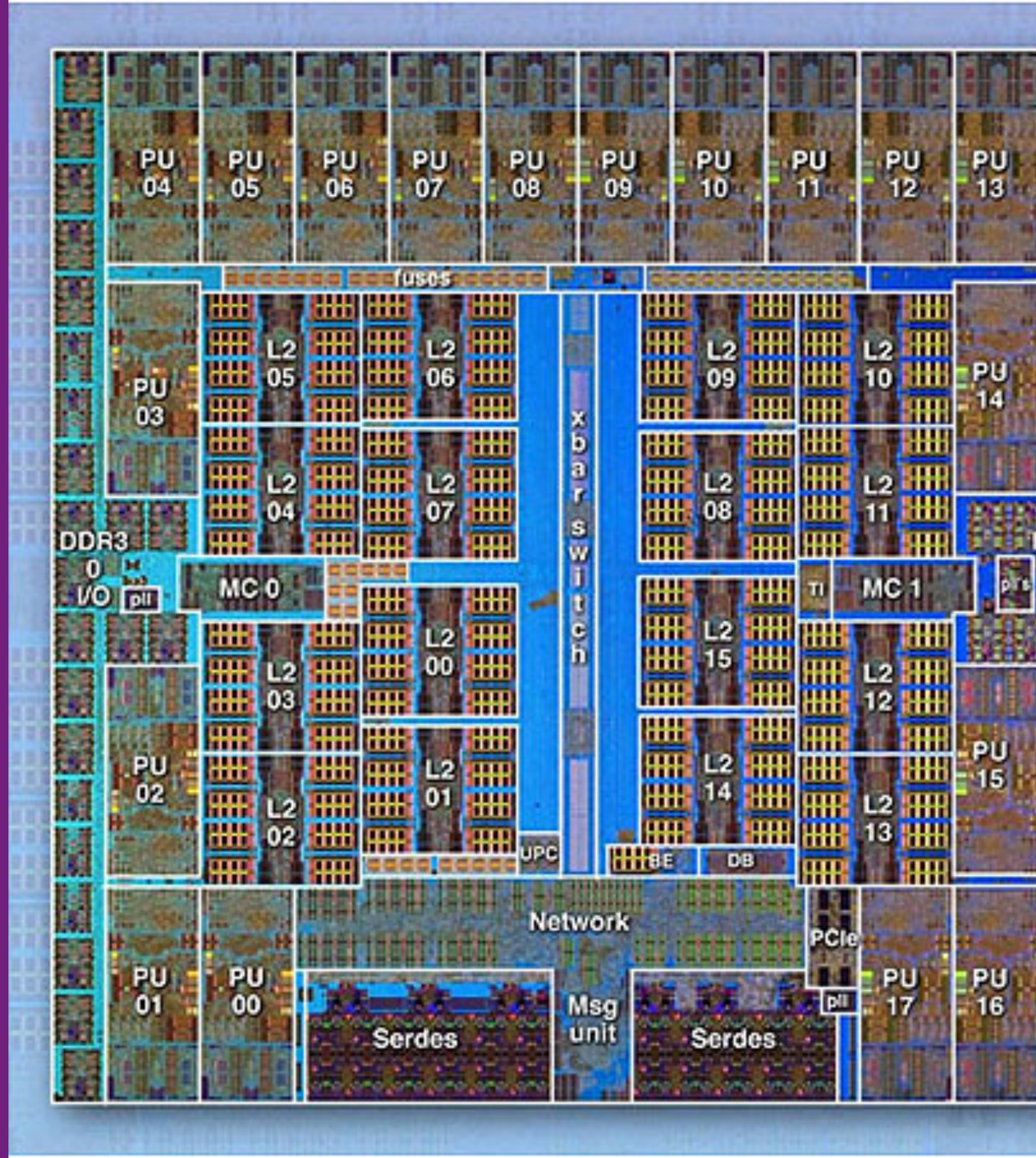
PARALLEL COMPUTERS

- Virtually all stand-alone computers today are parallel from a hardware perspective
  - Multiple functional units (L1 cache, L2 cache, branch, prefetch, decode, floating-point, graphics processing (GPU), integer, etc.)
  - Multiple execution units/cores
  - Multiple hardware threads



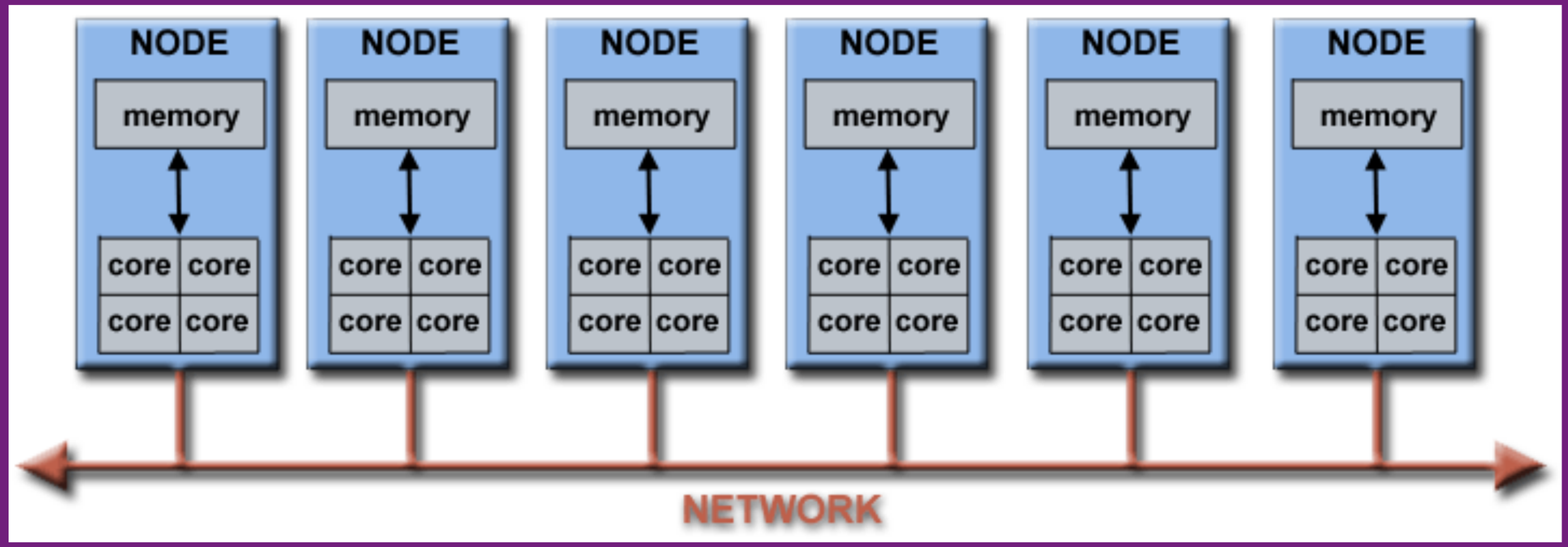
PARALLEL COMPUTERS

- IBM BG/Q Compute Chip
  - 18 cores (PU)
  - 16 L2 Cache units (L2)
- Next generation approaching 100 cores/chip



PARALLEL COMPUTERS

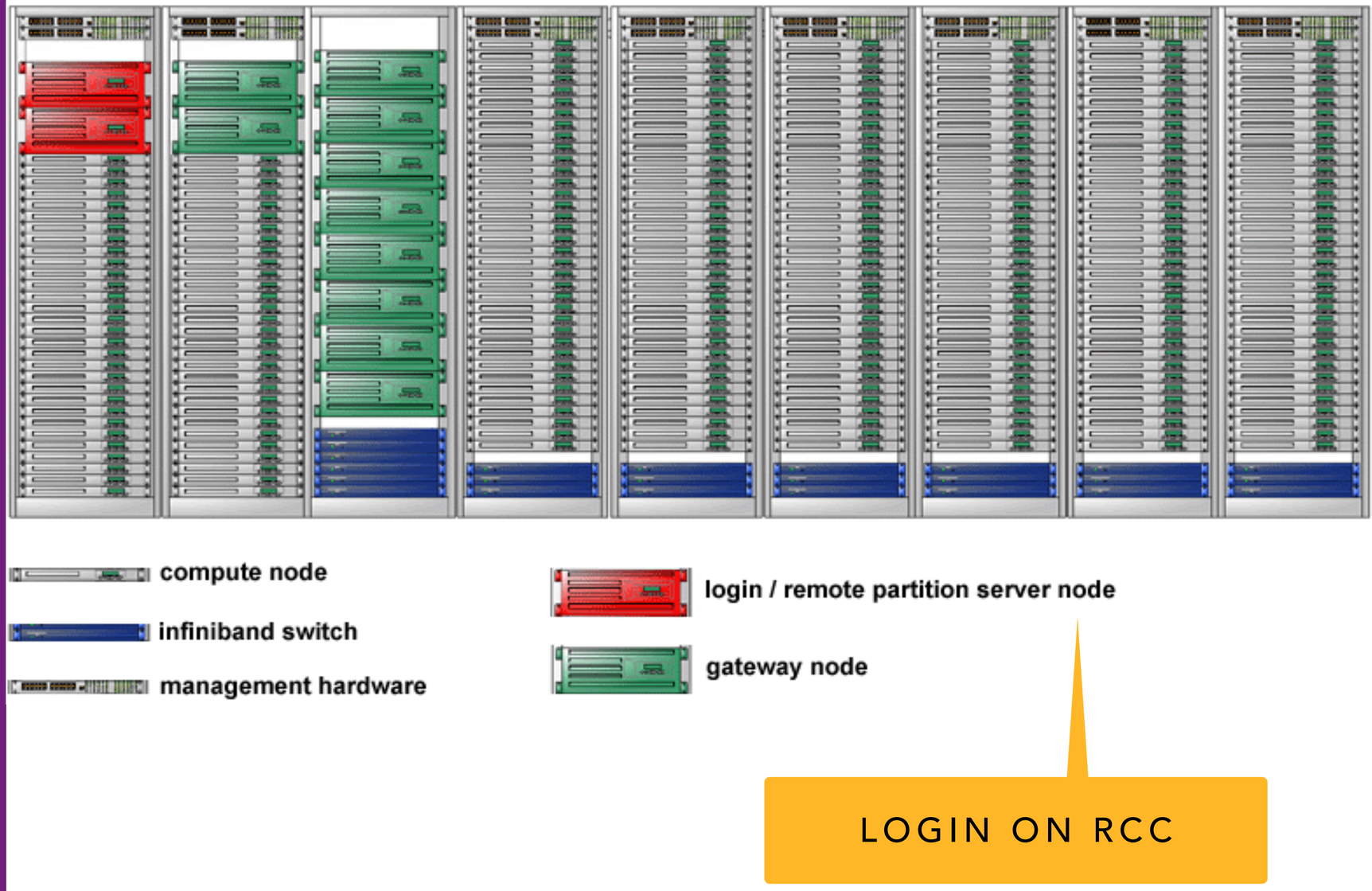
CAN BE THE BOTTLENECK



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

PARALLEL COMPUTERS

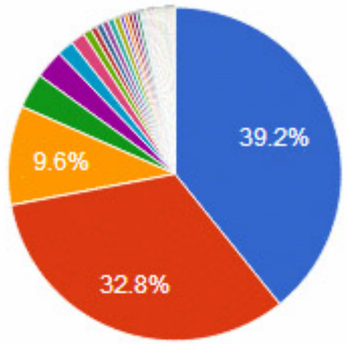
- Schematic shows a typical parallel computer cluster
  - Each compute node is a multi-processor parallel computer in itself
  - Multiple compute nodes are networked together with a network
  - Special purpose nodes, also multi-processor, are used for other purposes



PARALLEL COMPUTERS

- Majority of the world's large supercomputers are clusters of common (and cheap) hardware

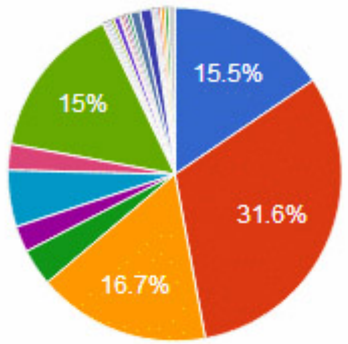
Vendors System Share



HP
IBM
Cray Inc.
SGI
Bull
Fujitsu
Dell

1/5

Vendors Performance Share



HP
IBM
Cray Inc.
SGI
Bull
Fujitsu
Dell

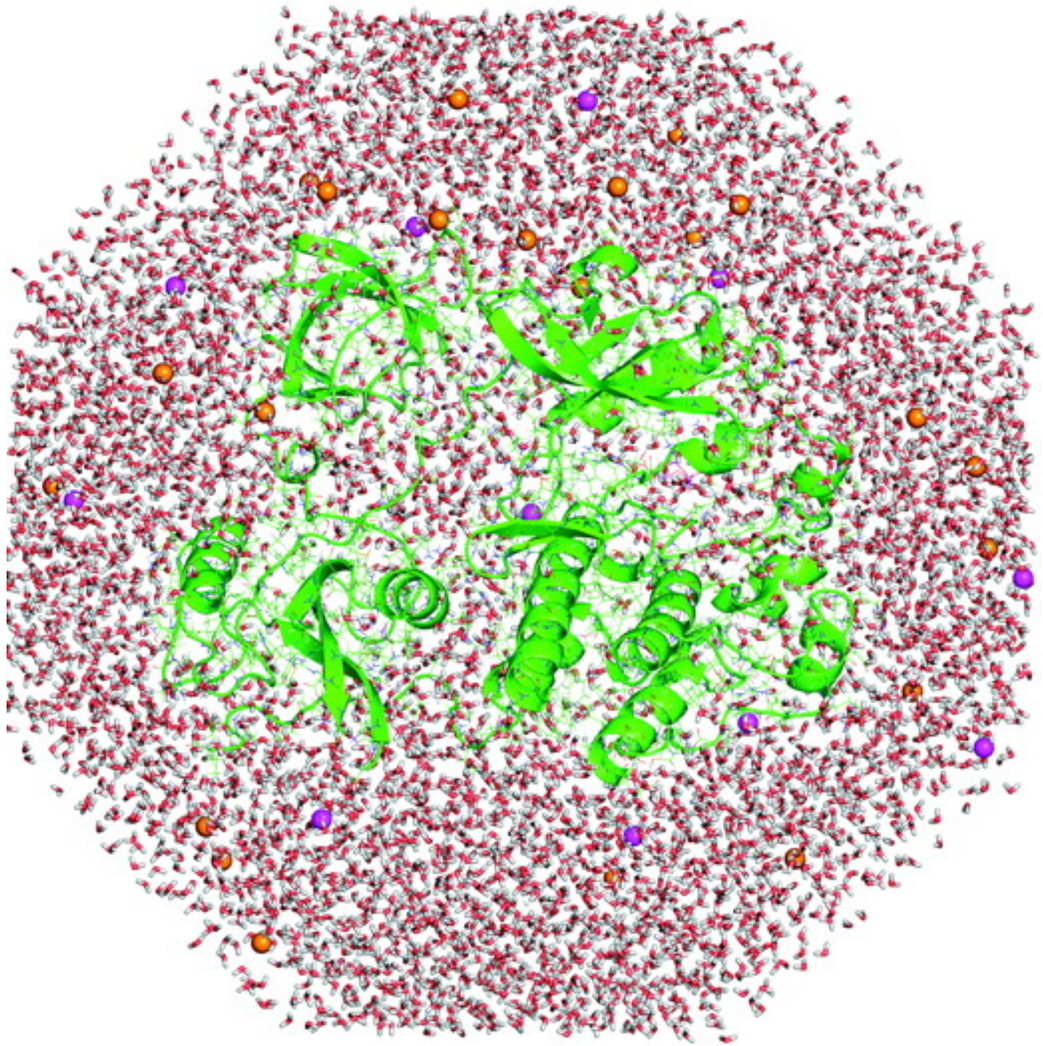
1/4

Vendors	Count	System Share (%)
HP	196	39.2
IBM	164	32.8
Cray Inc.	48	9.6
SGI	17	3.4
Bull	14	2.8
Fujitsu	8	1.6
Dell	7	1.4
NUDT	4	0.8
Supermicro	3	0.6
NEC	3	0.6
Megware	3	0.6
Hitachi	3	0.6
Oracle	3	0.6
Dawning	2	0.4
Itautec	2	0.4
RSC Group	2	0.4
Self-made	2	0.4
NRCPCET	2	0.4
Atipa	1	0.2
NEC/HP	1	0.2
Penguin Computing	1	0.2
Raytheon/Aspen Systems	1	0.2
Dell/Sun/IBM	1	0.2
Xenon Systems	1	0.2
Acer Group	1	0.2
HP/WIPRO	1	0.2
ClusterVision	1	0.2
Inspur	1	0.2
Clustervision/Supermicro	1	0.2
Netweb Technologies	1	0.2
IPE, Nvidia, Tyan	1	0.2
Adtech	1	0.2
Intel	1	0.2
T-Platforms	1	0.2
Hitachi/Fujitsu	1	0.2

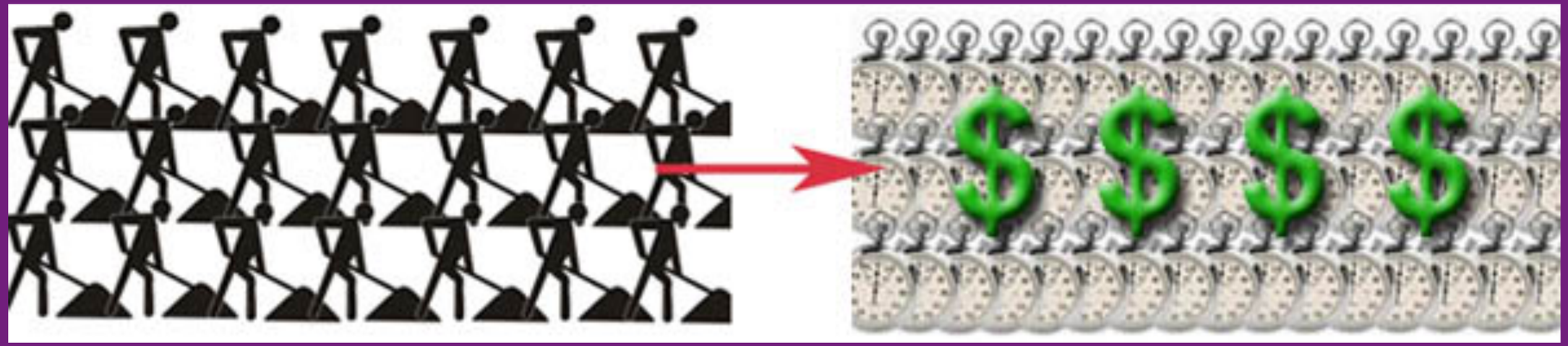
Motivation for Parallel Computing

MOTIVATION FOR PARALLEL COMPUTING

- The real world is parallel and complex
  - Interrelated events are happening at the same time
- Parallel computing is better suited for
  - Mdeling, simulating and understanding complex, real world phenomena



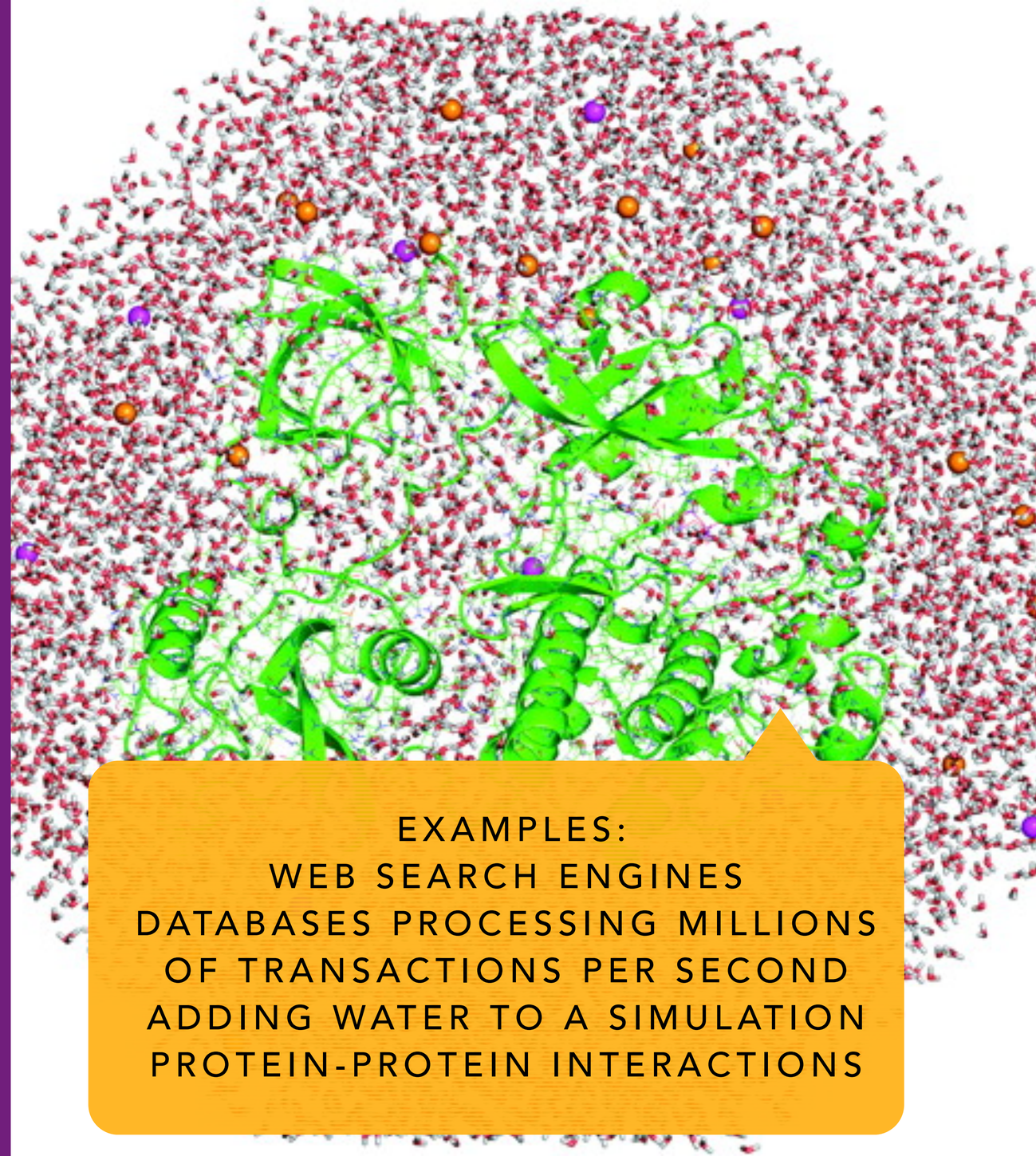
MOTIVATION FOR PARALLEL COMPUTING



- Save time/money/resources
  - Shorten its time to completion, with potential cost savings
  - Parallel computers can be built from cheap, commodity components

MOTIVATION FOR PARALLEL COMPUTING

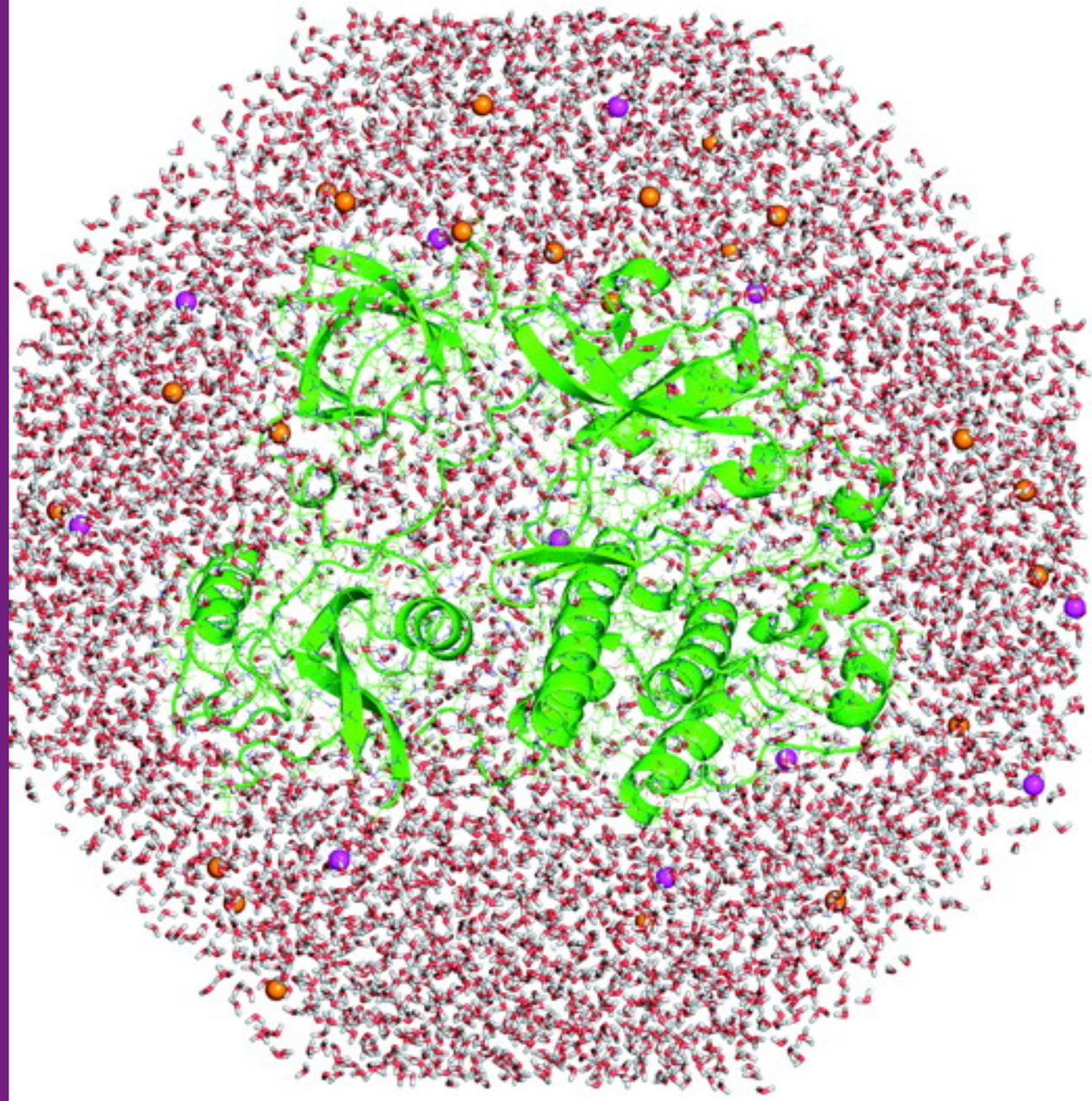
- Solve larger/more complex problems
  - Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer
  - Memory is often a major limitation



EXAMPLES:
WEB SEARCH ENGINES
DATABASES PROCESSING MILLIONS OF TRANSACTIONS PER SECOND
ADDING WATER TO A SIMULATION
PROTEIN-PROTEIN INTERACTIONS

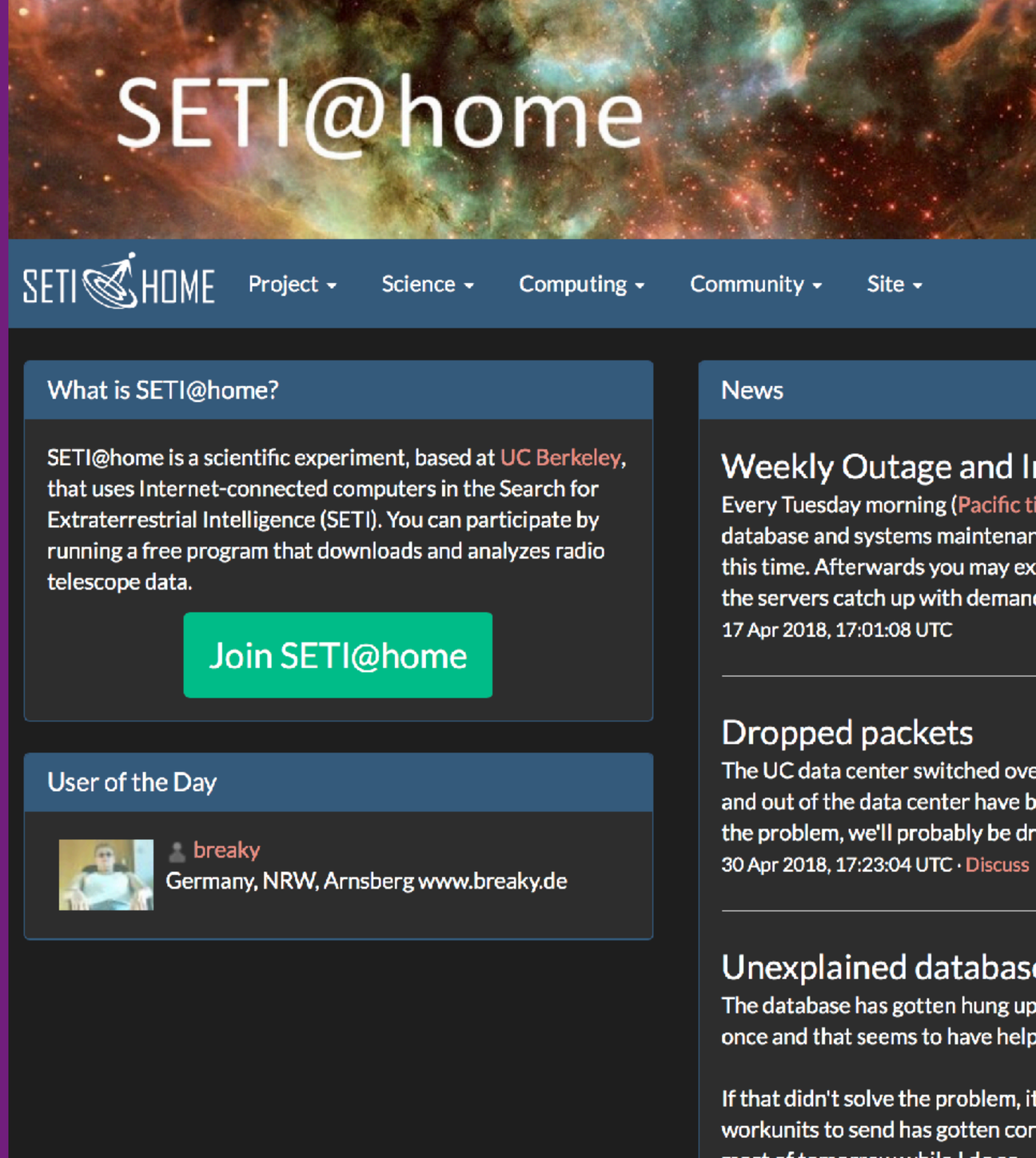
MOTIVATION FOR PARALLEL COMPUTING

- Provide concurrency
  - A single compute resource can only do one thing at a time
  - Multiple compute resources can do many things simultaneously



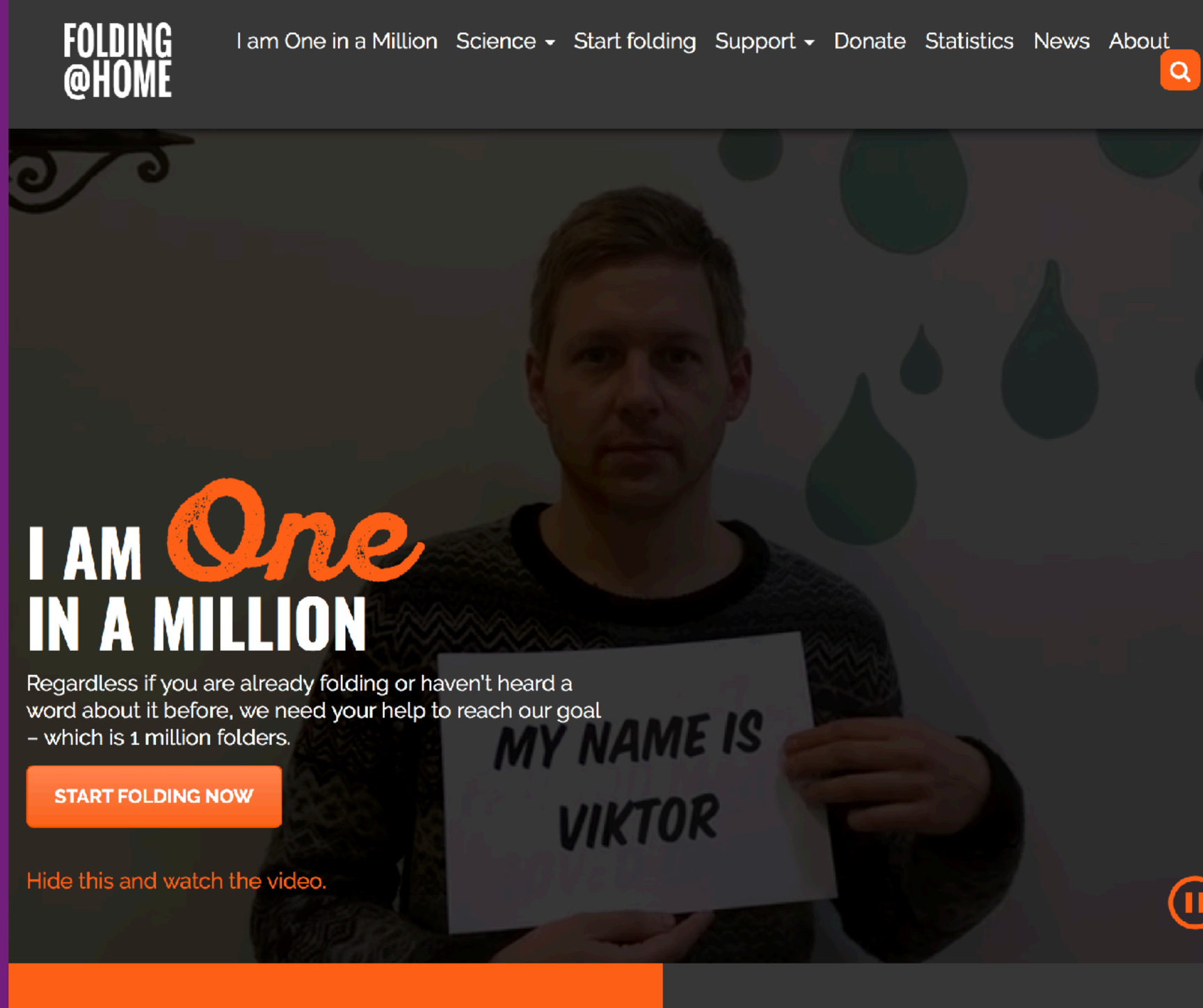
MOTIVATION FOR PARALLEL COMPUTING

- Take advantage of non-local resources
  - Using compute resources on a wide area network, or even the Internet
  - Example:
    - SETI@home (setiathome.berkeley.edu) over 1.3 million users, 3.4 million computers in nearly every country in the world

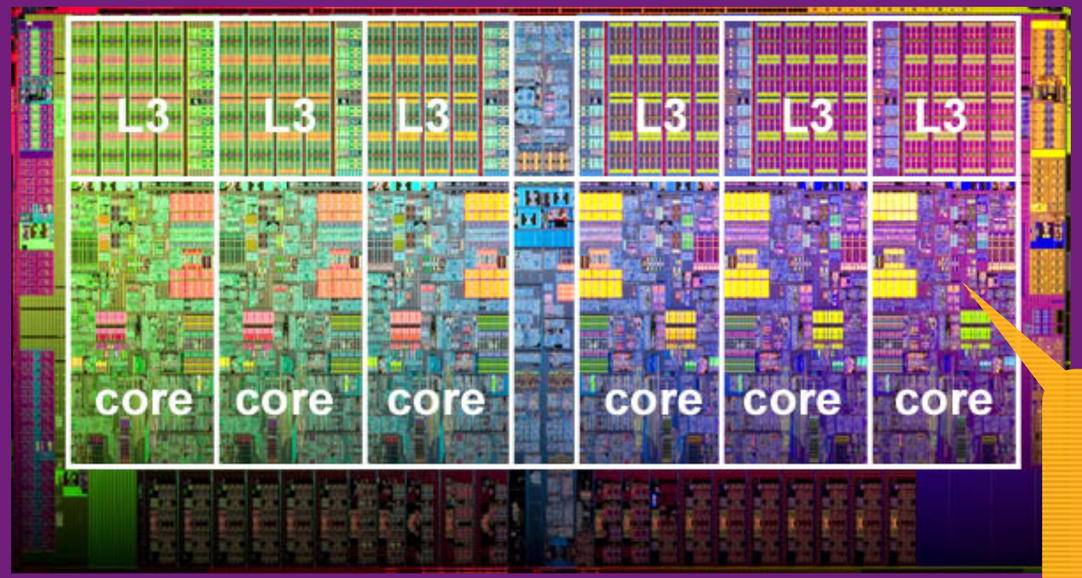


MOTIVATION FOR PARALLEL COMPUTING

- Folding@home (folding.stanford.edu) uses over 320,000 computers globally



MOTIVATION FOR PARALLEL COMPUTING



INTEL XEON PROCESSOR WITH 6 CORES AND 6 L3 CACHE UNITS

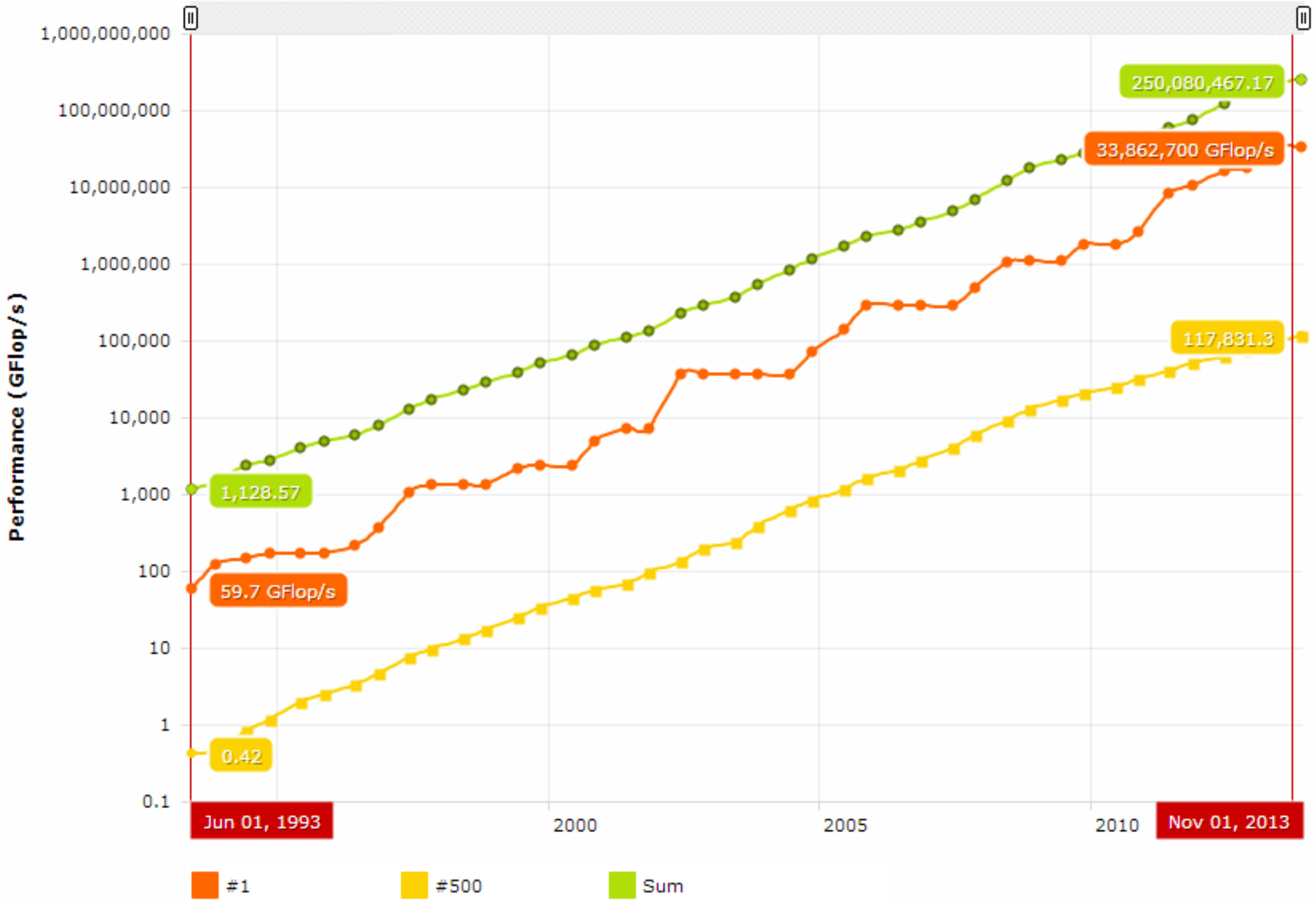
- Make better use of hardware
  - Modern computers are parallel in architecture with multiple processors/cores
  - Parallel software is intended for parallel hardware with multiple cores, threads, etc
  - Serial programs run on modern computers "waste" potential computing power

MOTIVATION FOR PARALLEL COMPUTING

- In 20 years there has been a greater than 500,000x increase in supercomputer performance

Performance Development

Exponential growth of supercomputing power as recorded by the TOP500.



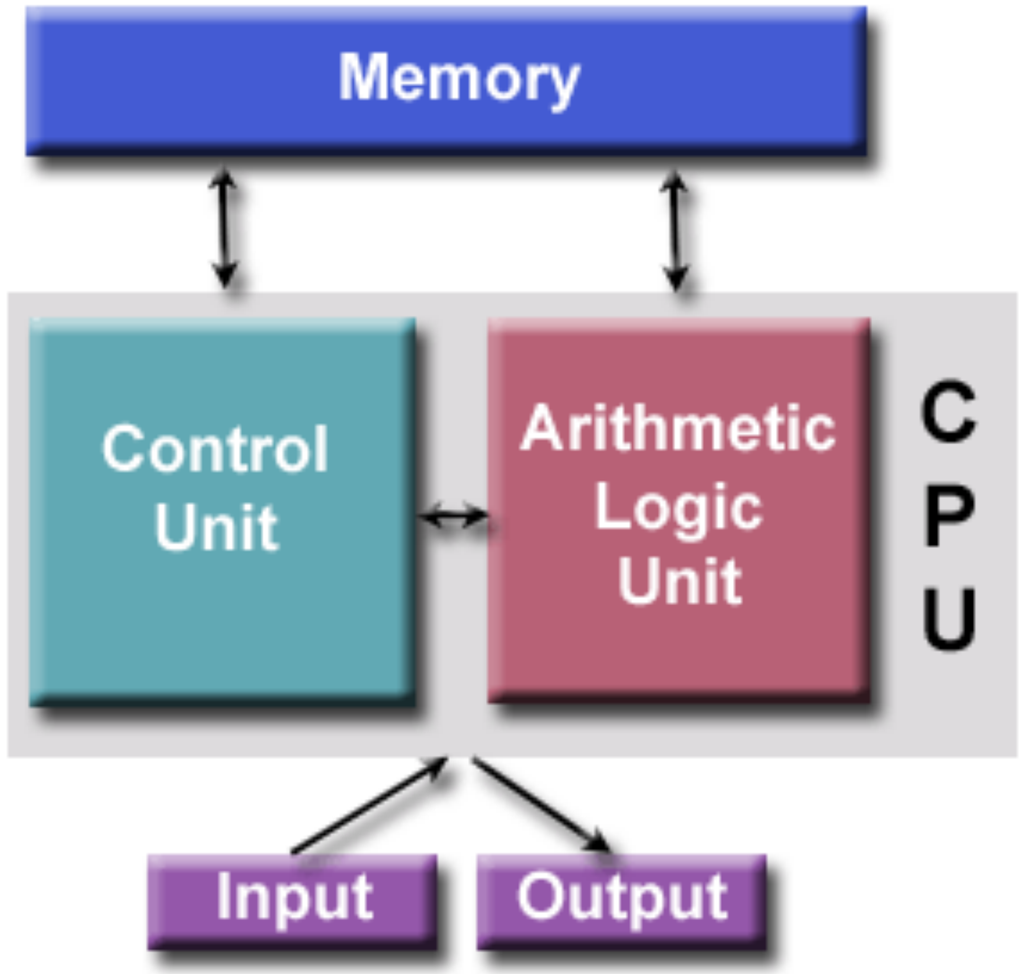
Concepts

CONCEPTS AND TERMINOLOGY

- von Neumann Architecture
  - Named after the Hungarian mathematician John von Neumann
    - Authored the general requirements for an electronic computer in his 1945 papers
  - Also known as "stored-program computer"
  - Both program instructions and data are kept in electronic memory
  - Differs from earlier computers which were programmed through "hard wiring"

CONCEPTS AND TERMINOLOGY

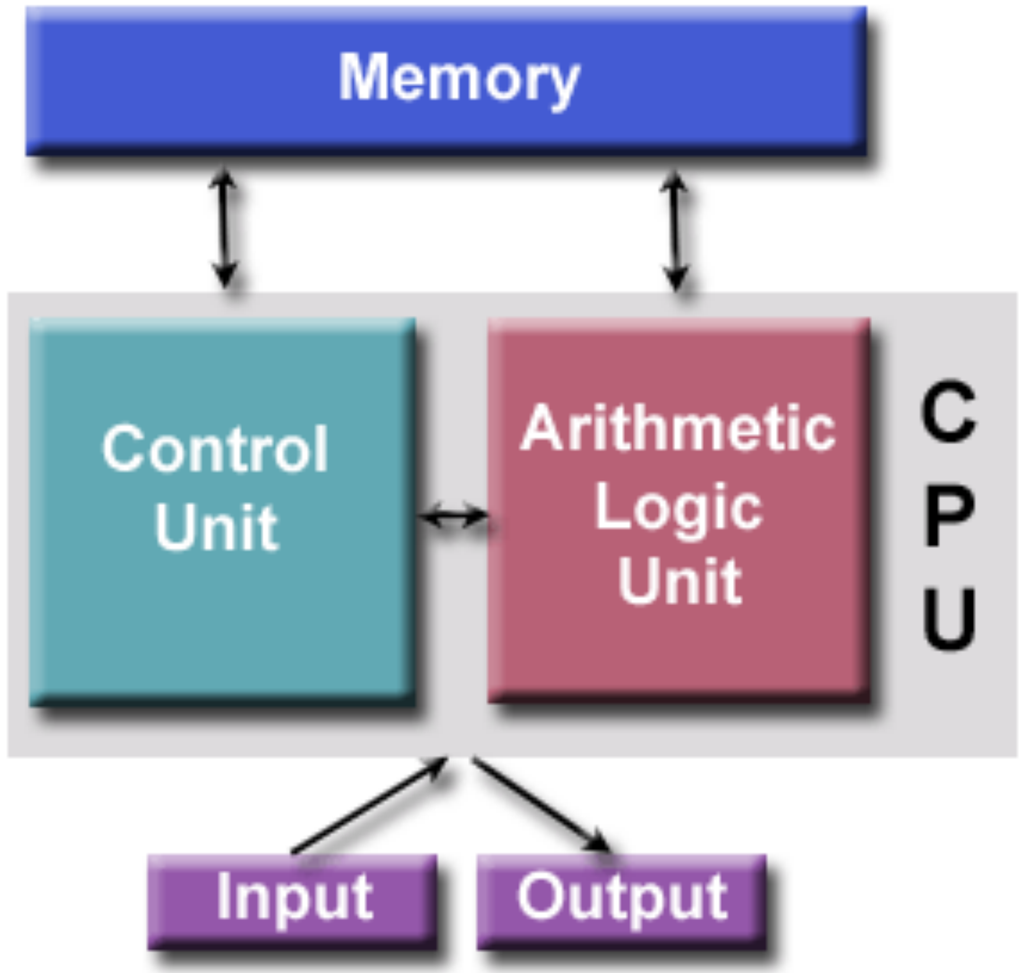
- Virtually all computers follow basic design:
  - Memory
  - Control Unit
  - Arithmetic Logic Unit
  - Input/Output



PARALLEL COMPUTERS STILL FOLLOW THIS BASIC DESIGN, JUST MULTIPLIED IN UNITS

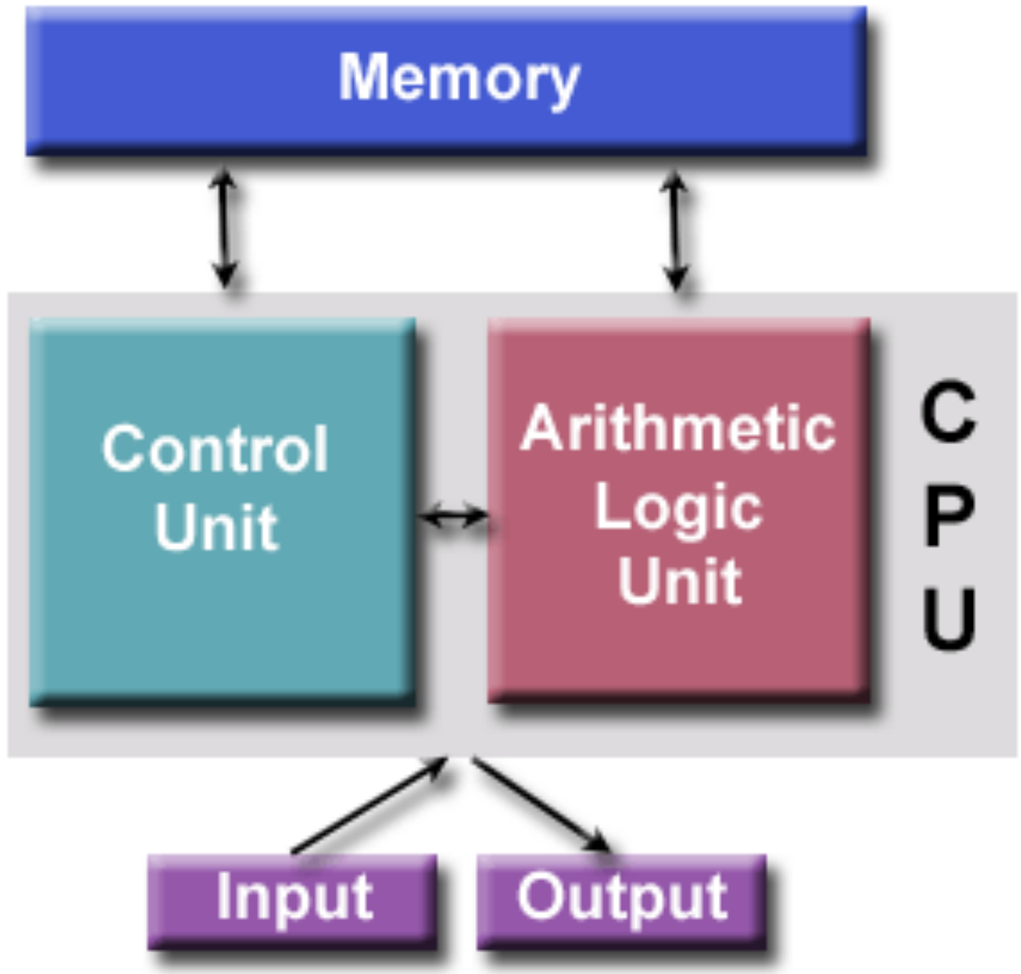
CONCEPTS AND TERMINOLOGY

- Read/write, random access memory is used to store both program instructions and data
  - Program instructions are coded data which tell the computer to do something
  - Data is information to be used by the program



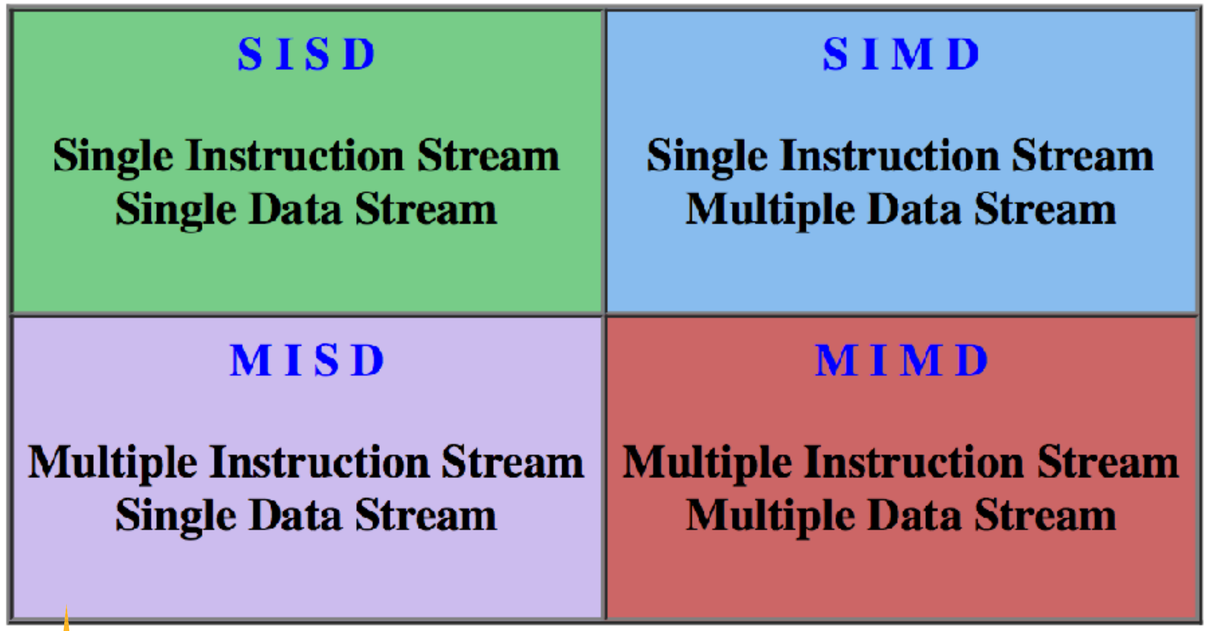
CONCEPTS AND TERMINOLOGY

- Control unit
  - Fetches instructions/data from memory
  - Decodes
  - Sequentially coordinates operations to accomplish the programmed task
- Arithmetic Unit
  - Performs basic arithmetic operations
- Input/Output
  - Interface to the human



CONCEPTS AND TERMINOLOGY

- There are different ways to classify parallel computers
- Flynn's Classical Taxonomy
  - Instruction Stream
  - Data Stream
  - Only one of two possible states
    - Single
    - Multiple



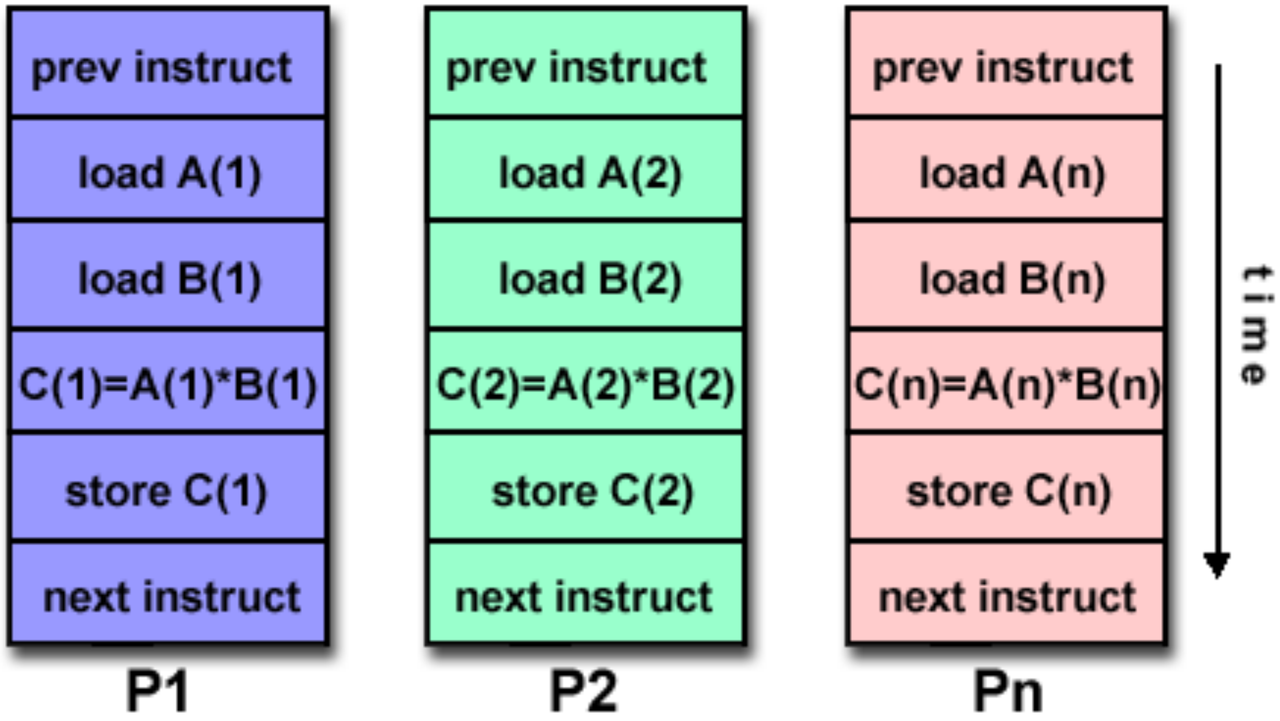
THE MATRIX BELOW DEFINES THE 4 POSSIBLE CLASSIFICATIONS ACCORDING TO FLYNN TAXONOMY (1966)

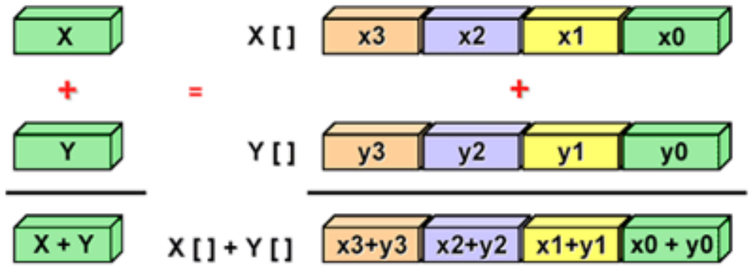
CONCEPTS AND TERMINOLOGY

- Single Instruction, Single Data (SISD)
  - A serial computer

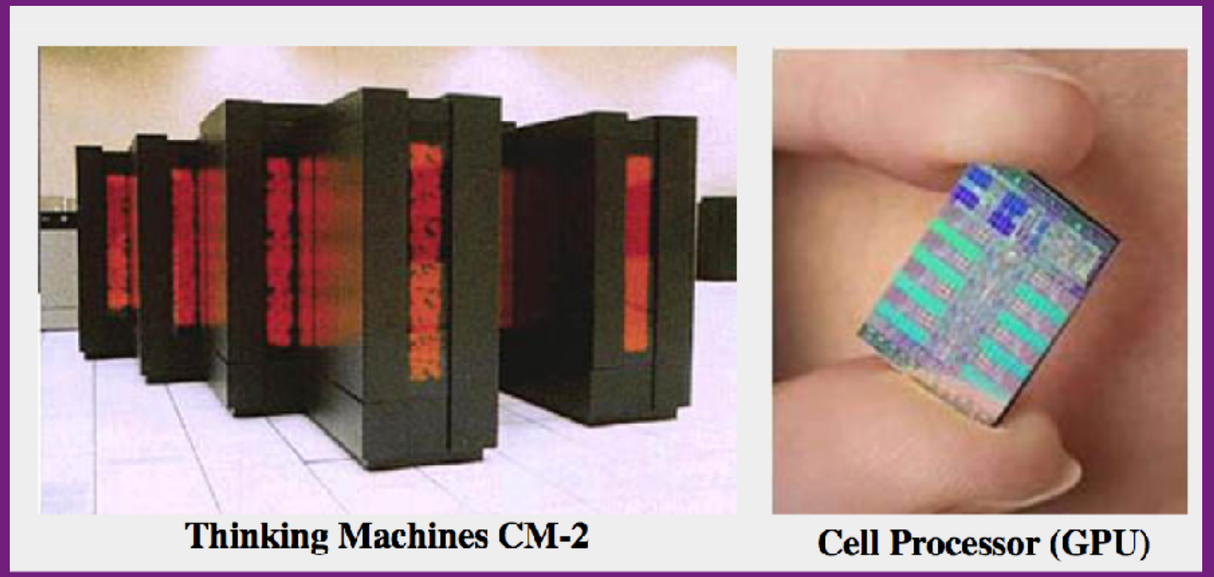
CONCEPTS AND TERMINOLOGY

- Single Instruction, Multiple Data (SIMD)
  - A parallel computer
  - Single Instruction:
    - All processing units execute the same instruction at any given clock cycle
  - Multiple Data:
    - Each processing unit can operate on a different data element
  - Suited for specialized problems characterized by a high degree of regularity
  - Synchronous and deterministic execution
  - Two varieties
    - Processor Arrays and Vector Pipelines





CONCEPTS AND TERMINOLOGY



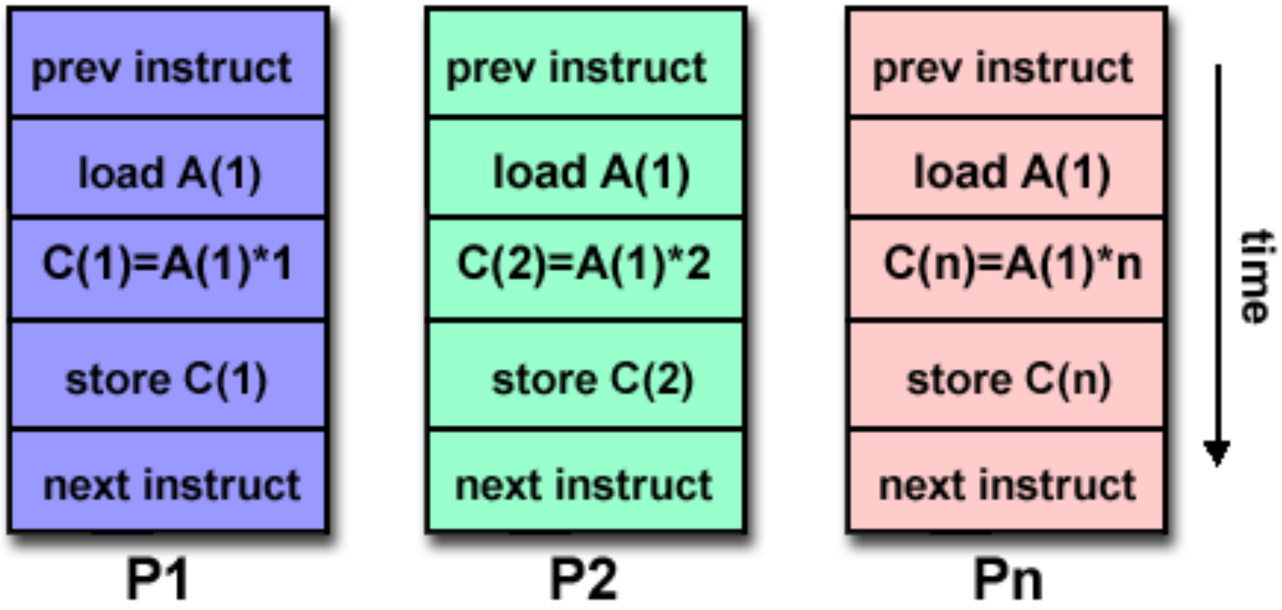
Thinking Machines CM-2

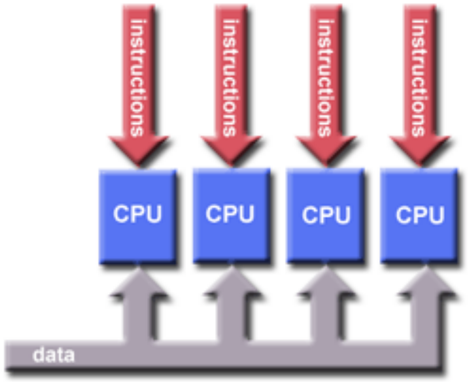
Cell Processor (GPU)

- SIMD examples:
  - Processor Arrays: Thinking Machines CM-2, MasPar MP-1 & MP-2, ILLIAC IV
  - Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10
  - Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.

CONCEPTS AND TERMINOLOGY

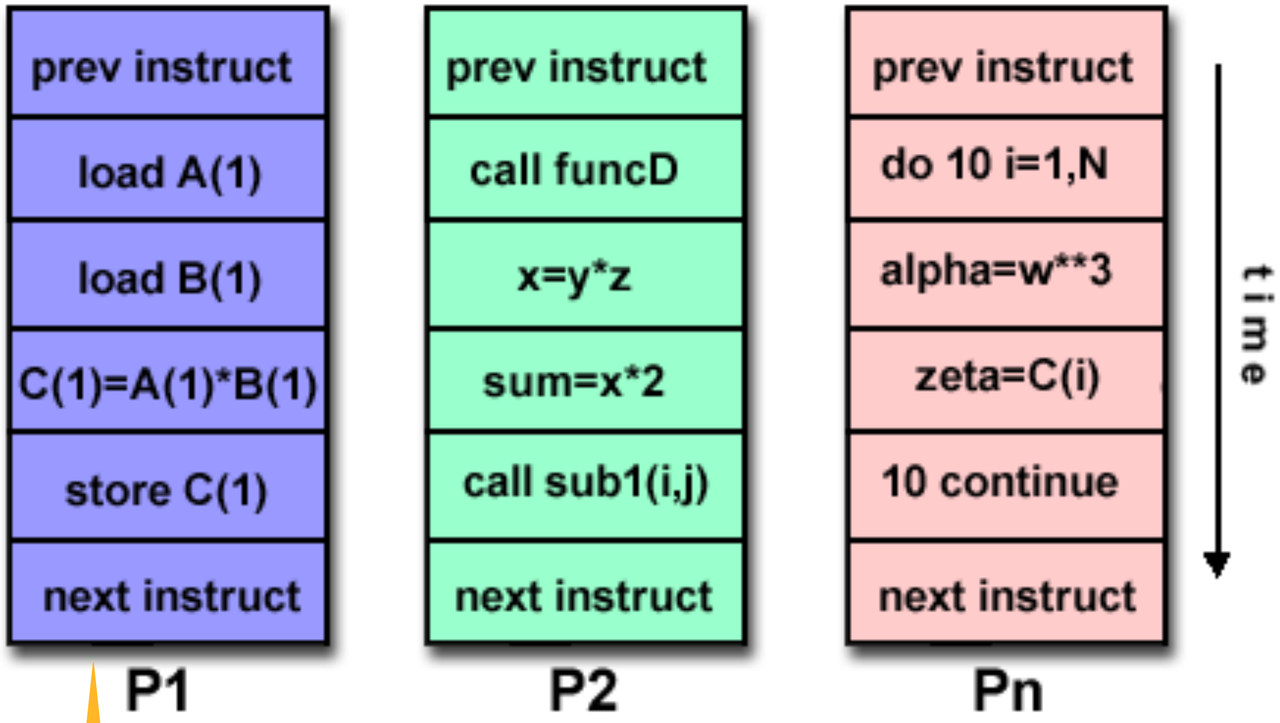
- Multiple Instruction, Single Data (MISD)
  - A type of parallel computer
  - Multiple Instruction
    - Each processing unit operates on the data independently via separate instruction streams.
  - Single Data
    - A single data stream is fed into multiple processing units.
  - Few actual examples of this class of parallel computer
  - Some conceivable uses might be:
    - Multiple cryptography algorithms attempting to crack a single coded message





CONCEPTS AND TERMINOLOGY

- Multiple Instruction, Multiple Data (MIMD):
  - A type of parallel computer
  - Multiple Instruction
    - Every processor may be executing a different instruction stream
  - Multiple Data
    - Every processor may be working with a different data stream
  - Execution can be synchronous or asynchronous, deterministic or non-deterministic
  - The most common type of parallel computer



MANY MIMD ARCHITECTURES ALSO INCLUDE SIMD EXECUTION SUB-COMPONENTS

CONCEPTS AND TERMINOLOGY

- Multiple Instruction, Multiple Data (MIMD) parallel computers
  - Most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs

IBM POWER5

HP/Compaq Alphaserver

Intel IA32

AMD Opteron

Cray XT3

IBM BG/L

Some General Parallel Terminology

CONCEPTS AND TERMINOLOGY

- Supercomputing / High Performance Computing (HPC)
  - Using the world's fastest and largest computers to solve large problems
- Node
  - A standalone "computer in a box”
  - Usually comprised of multiple CPUs/processors/cores, memory, network interfaces, etc.
  - Nodes are networked together to comprise a supercomputer

CONCEPTS AND TERMINOLOGY

- CPU / Socket / Processor / Core
  - Smallest unit of execution component (varies on vendors)
  - Node have multiple CPUs, each containing multiple cores

CONCEPTS AND 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
  - A parallel program consists of multiple tasks running on multiple processors

CONCEPTS AND TERMINOLOGY

- Pipeline
  - Breaking a task into steps performed by different processor units
  - Inputs stream to outputs and used as inputs
  - Scripting "glue"

CONCEPTS AND TERMINOLOGY

- Shared Memory
  - Hardware:
    - Computer architecture where all processors have direct access to common physical memory
  - Programming:
    - Model where parallel tasks all have the same "picture" of memory
    - Can directly address and access the same logical memory locations regardless of where the physical memory actually exists

CONCEPTS AND TERMINOLOGY

- Distributed Memory
  - Hardware
    - Refers to network based memory access for physical memory that is not common
  - 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

CONCEPTS AND TERMINOLOGY

- Communications
  - Parallel tasks typically need to exchange data

- 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

CONCEPTS AND TERMINOLOGY

- Granularity
  - A qualitative measure of the ratio of computation to communication
  - Coarse:
    - Large amounts of computational work are done between communication events
  - Fine:
    - Small amounts of computational work are done between communication events

CONCEPTS AND TERMINOLOGY

- Observed Speedup
  - Observed speedup of a code which has been parallelized
    - wall clock serial/wall clock parallel
  - One of the simplest and most widely used indicators for a parallel program's performance

CONCEPTS AND TERMINOLOGY

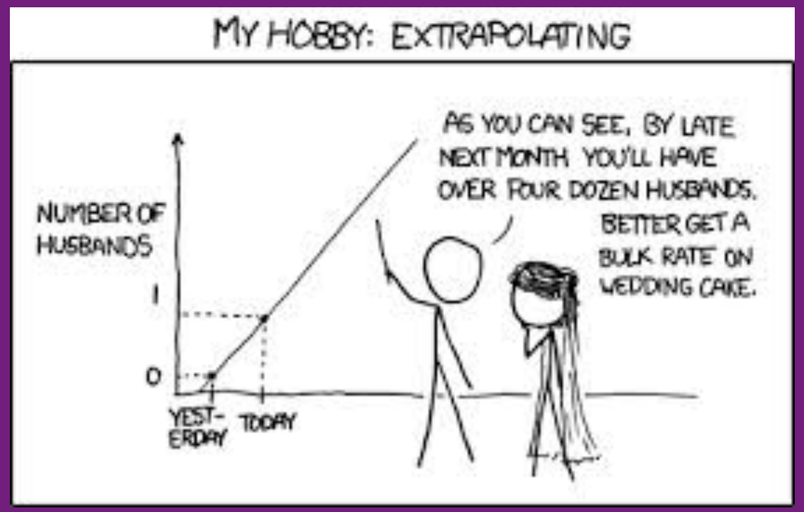
- Parallel Overhead
  - The amount of time required to coordinate parallel tasks (as opposed to doing useful work)
  - Parallel overhead factors
    - Task start-up time, synchronizations, data communications
    - Software overhead imposed by parallel languages, libraries, operating system, etc.
    - Task termination time

CONCEPTS AND TERMINOLOGY

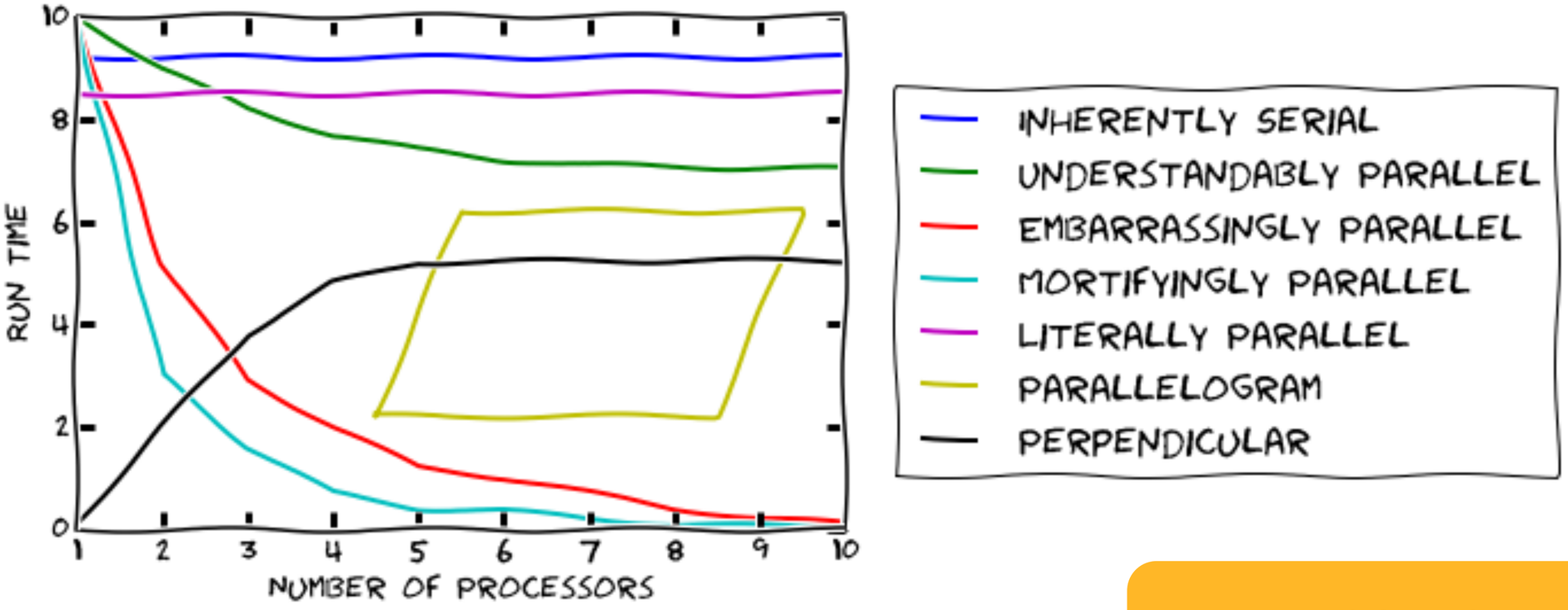
- Scalability
  - Refers to a parallel system's ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources
  - Factors that contribute to scalability include:
    - Hardware - particularly memory-cpu bandwidths and network communication properties
    - Application algorithm
    - Parallel overhead related
    - Characteristics of your specific application

CONCEPTS AND TERMINOLOGY

- Massively Parallel
  - Refers to the hardware that comprises a given parallel system - having many processors
- Embarrassingly Parallel
  - Solving many similar, but independent tasks simultaneously
  - Little to no need for coordination between the tasks



CONCEPTS AND TERMINOLOGY



A COMIC ABOUT EMBARRASSINGLY PARALLEL...

Limits and Costs of Parallel Programming

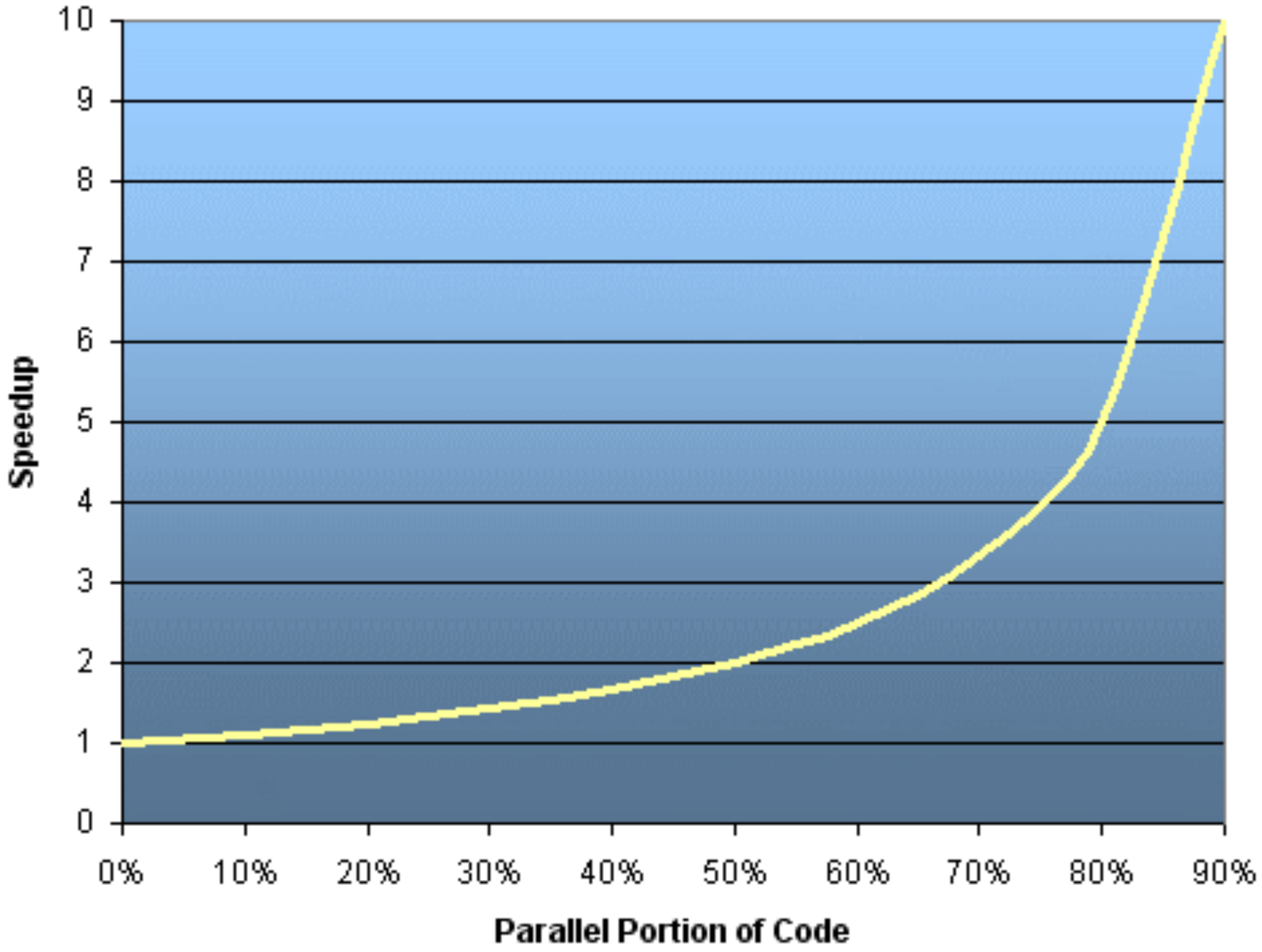
LIMITS AND COSTS OF PARALLEL PROGRAMMING

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

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

- If no code can be parallelized, P = 0 and the speedup = 1 (no speedup)
- If all of the code is parallelized, P = 1 and the speedup is infinite (in theory)
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast

LIMITS AND COSTS OF PARALLEL PROGRAMMING



LIMITS AND COSTS OF PARALLEL PROGRAMMING

- Amdahl's Law
  - Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by
    - P = parallel fraction
    - N = number of processors
    - S = serial fraction

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

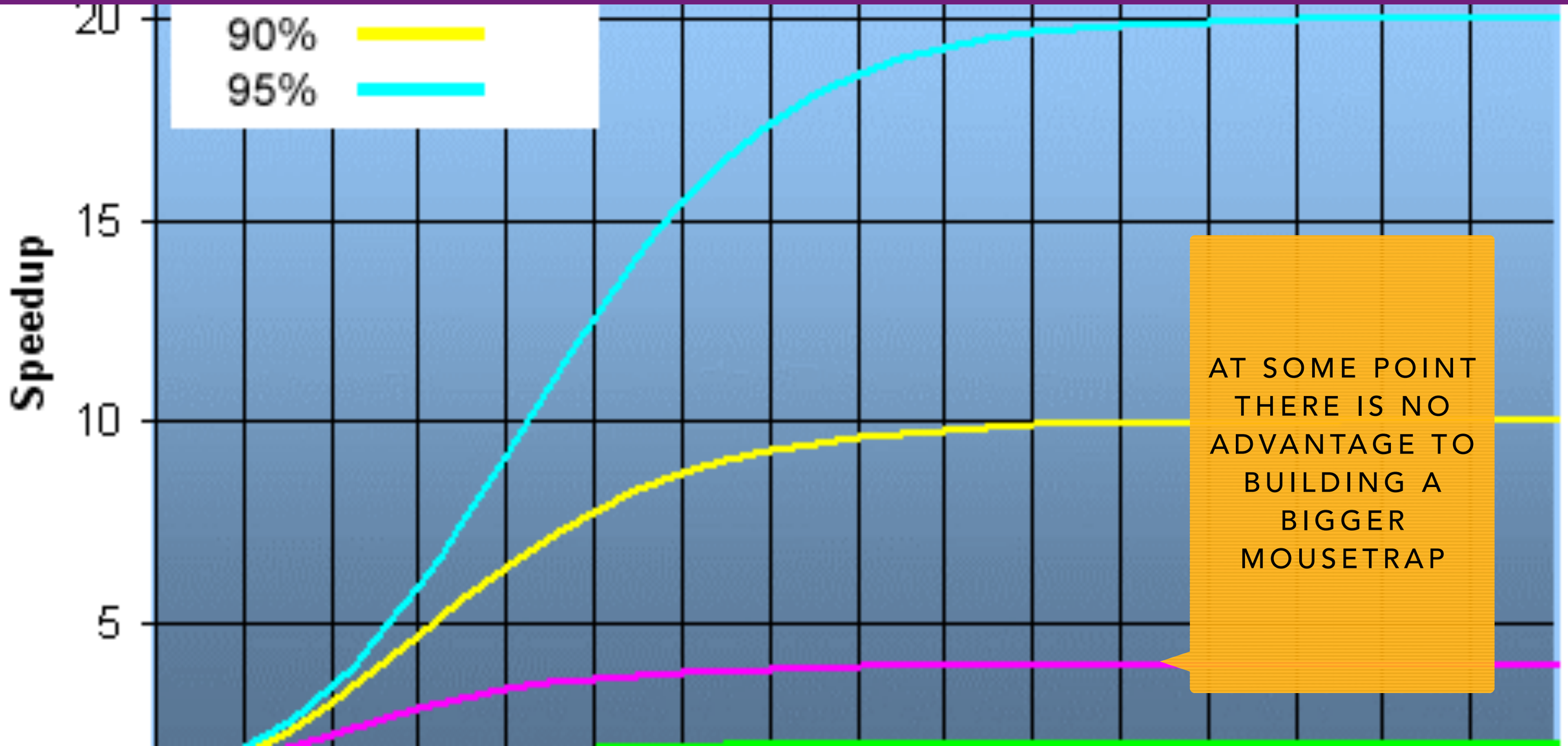
LIMITS AND COSTS OF PARALLEL PROGRAMMING

- There are limits to the scalability of parallelism
- Problems that increase the percentage of parallel time with their size are more **scalable** than problems with a fixed percentage of parallel time

	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1,000	1.99	9.91	90.99
10,000	1.99	9.91	99.02
100,000	1.99	9.99	99.90

AT SOME POINT THERE IS NO ADVANTAGE TO PARALLELISM

LIMITS AND COSTS OF PARALLEL PROGRAMMING



LIMITS AND COSTS OF PARALLEL PROGRAMMING

COMPLEXITY

- Parallel applications are much more complex than corresponding serial applications
  - Multiple instruction/data streams executing at the same time
- The costs of complexity are measured in programmer time
  - Design, Coding , Debugging, Tuning, Maintenance
  - Adhering to "good" software development practices is essential

LIMITS AND COSTS OF PARALLEL PROGRAMMING

PORTABILITY

- Used to be a huge problem (Standardization in several APIs has made this better)
  - Vendor "enhancements" to Fortran, C or C++
  - Even though standards exist for several APIs, implementations will differ in a number of details
    - Operating systems
    - Hardware architectures

EVEN IBM/DEVELOPER COULDN'T GET ROSETTA TO COMPILE ON BLUEGENE

LIMITS AND COSTS OF PARALLEL PROGRAMMING

RESOURCE REQUIREMENTS

- Intent of parallel programming is to decrease execution wall clock time, however in order to accomplish this, more CPU time is required
- For example, a parallel code that runs in 1 hour on 8 processors actually uses 8 hours of CPU time.

LIMITS AND COSTS OF PARALLEL PROGRAMMING

RESOURCE REQUIREMENTS

- The amount of memory is typically greater for parallel codes
  - Replicate data
  - Overheads associated with parallel support libraries and subsystems

LIMITS AND COSTS OF PARALLEL PROGRAMMING

RESOURCE REQUIREMENTS

- For short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation
  - The overhead costs associated with setting up the parallel environment
  - Task creation
  - Communication
  - Task termination
- Can comprise a significant portion of the total execution time for short runs

BLAST??

LIMITS AND COSTS OF PARALLEL PROGRAMMING

SCALABILITY

- Two types of scaling based on time to solution
  - Strong scaling: The total problem size stays fixed as more processors are added
  - Weak scaling: The problem size per processor stays fixed as more processors are added

LIMITS AND COSTS OF PARALLEL PROGRAMMING

SCALABILITY

- The ability of a parallel program's performance to scale is a result of a number of interrelated factors
  - Simply adding more processors is rarely the answer
  - The algorithm may have inherent limits to scalability
- At some point, adding more resources causes performance to decrease
  - Many parallel solutions demonstrate this characteristic at some point

LIMITS AND COSTS OF PARALLEL PROGRAMMING

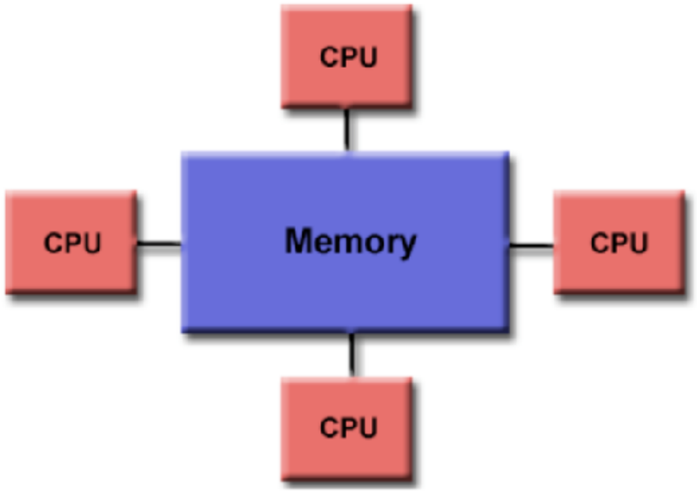
SCALABILITY

- Hardware factors play a significant role in scalability
  - Memory-cpu bus bandwidth on an SMP machine
  - Communications network bandwidth
  - Amount of memory available on any given machine or set of machines
  - Processor clock speed
- Parallel support libraries and subsystems software can limit scalability independent of your application

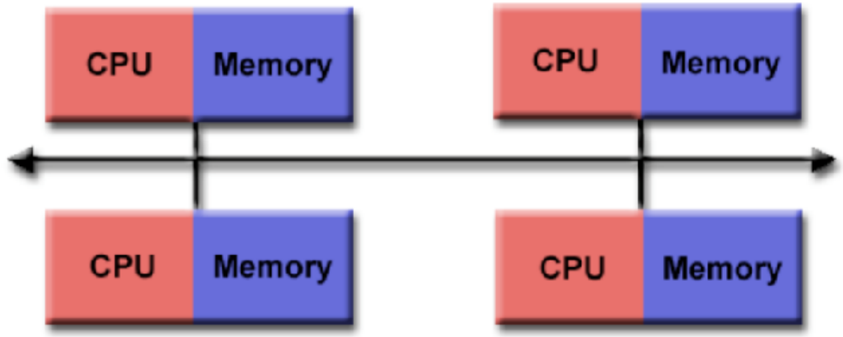
Parallel Computer Memory Architectures - Shared

PARALLEL COMPUTER MEMORY ARCHITECTURES

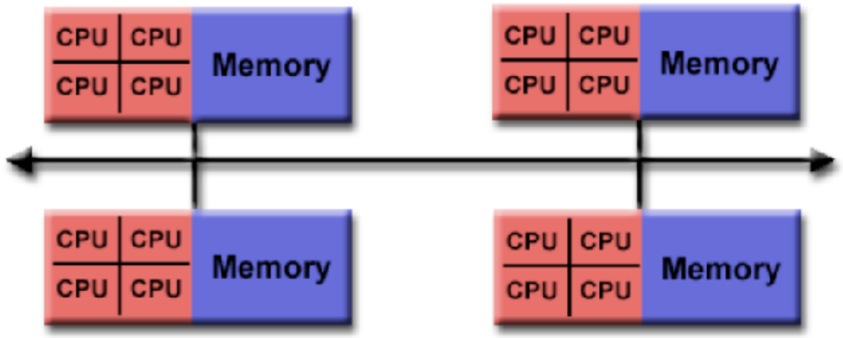
Shared Memory



Distributed Memory



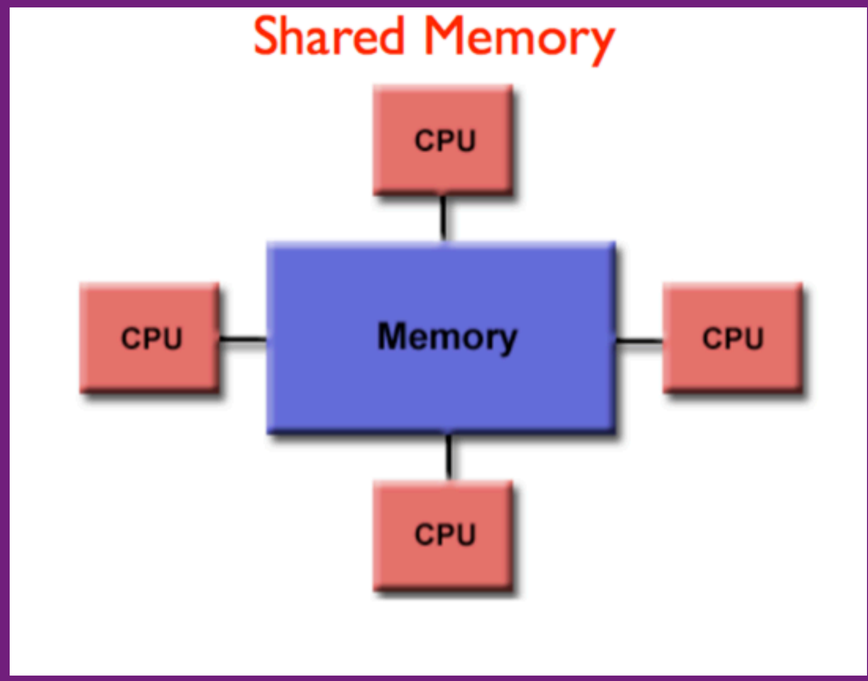
Hybrid Distributed Shared Memory



PARALLEL COMPUTER MEMORY ARCHITECTURES

SHARED MEMORY

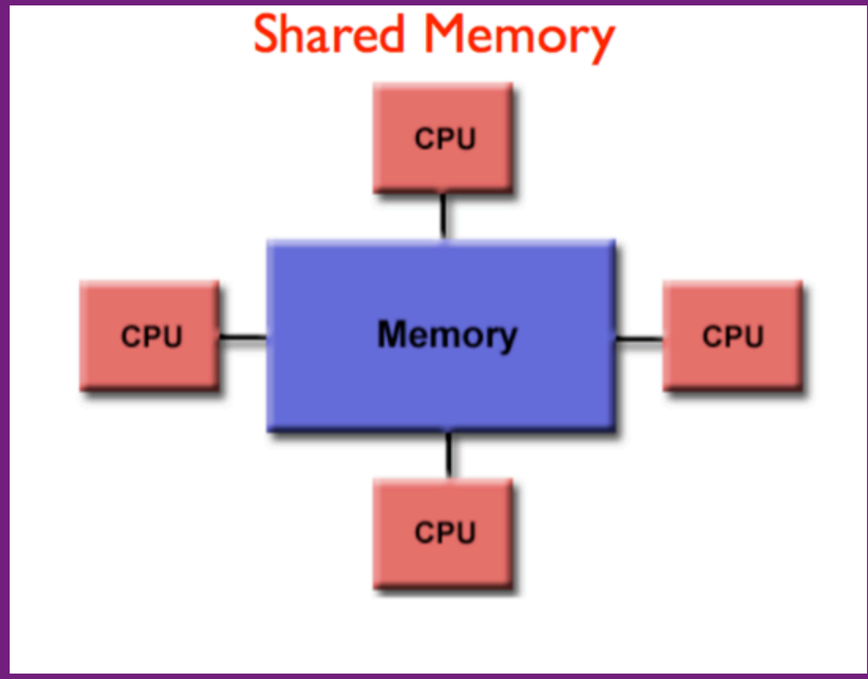
- Multiple processors can operate independently but share the same memory resources
    - Changes in a memory location effected by one processor are visible to all other processors



PARALLEL COMPUTER MEMORY ARCHITECTURES

SHARED MEMORY

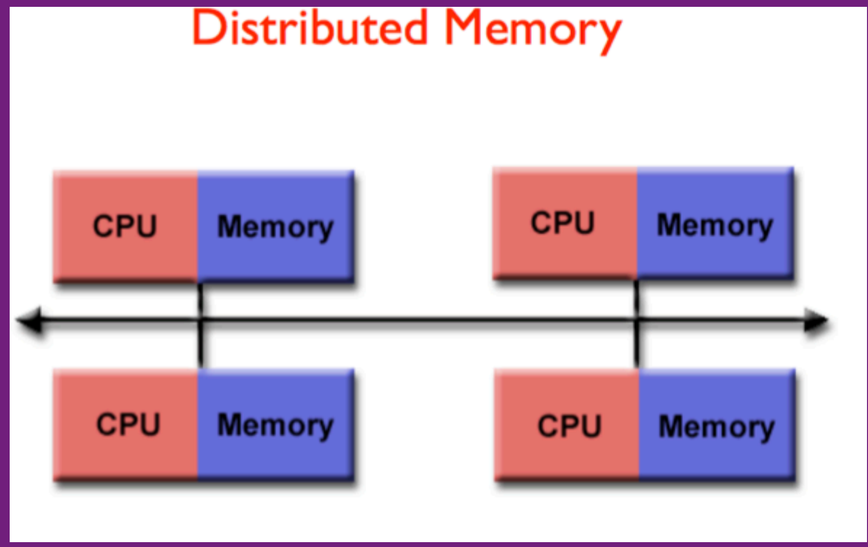
- Advantages:
  - User-friendly programming perspective to memory
  - Data sharing between tasks is both fast and uniform due
- Disadvantages:
  - Adding more CPUs can geometrically increases traffic on the shared memory-CPU path
  - Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.



PARALLEL COMPUTER MEMORY ARCHITECTURES

DISTRIBUTED MEMORY

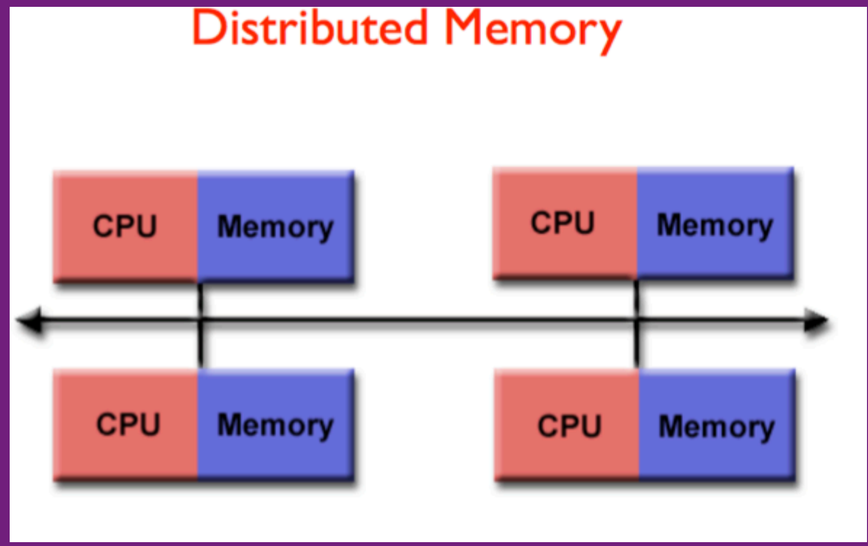
- Processors have their own local memory
  - No global address space
- Require a communication network to connect inter-processor memory



PARALLEL COMPUTER MEMORY ARCHITECTURES

DISTRIBUTED MEMORY

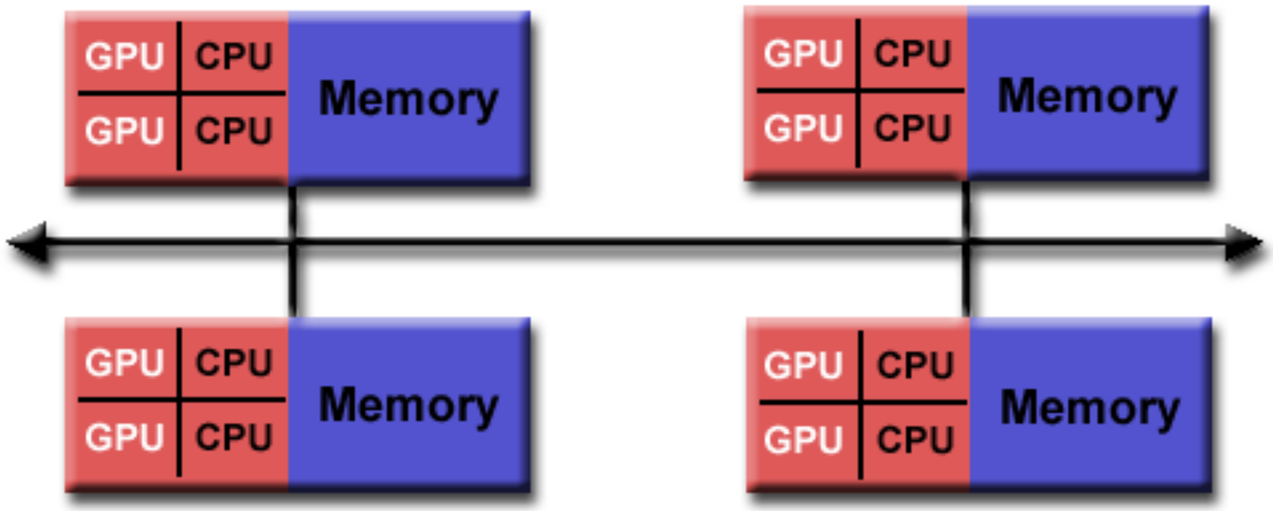
- Advantages
  - Memory is scalable with the number of processors
  - Cost effectiveness: commodity, off-the-shelf processors and networking
- Disadvantages
  - Programmer is responsible for many of the details associated with data communication between processors
  - It may be difficult to map existing data structures to this memory organization

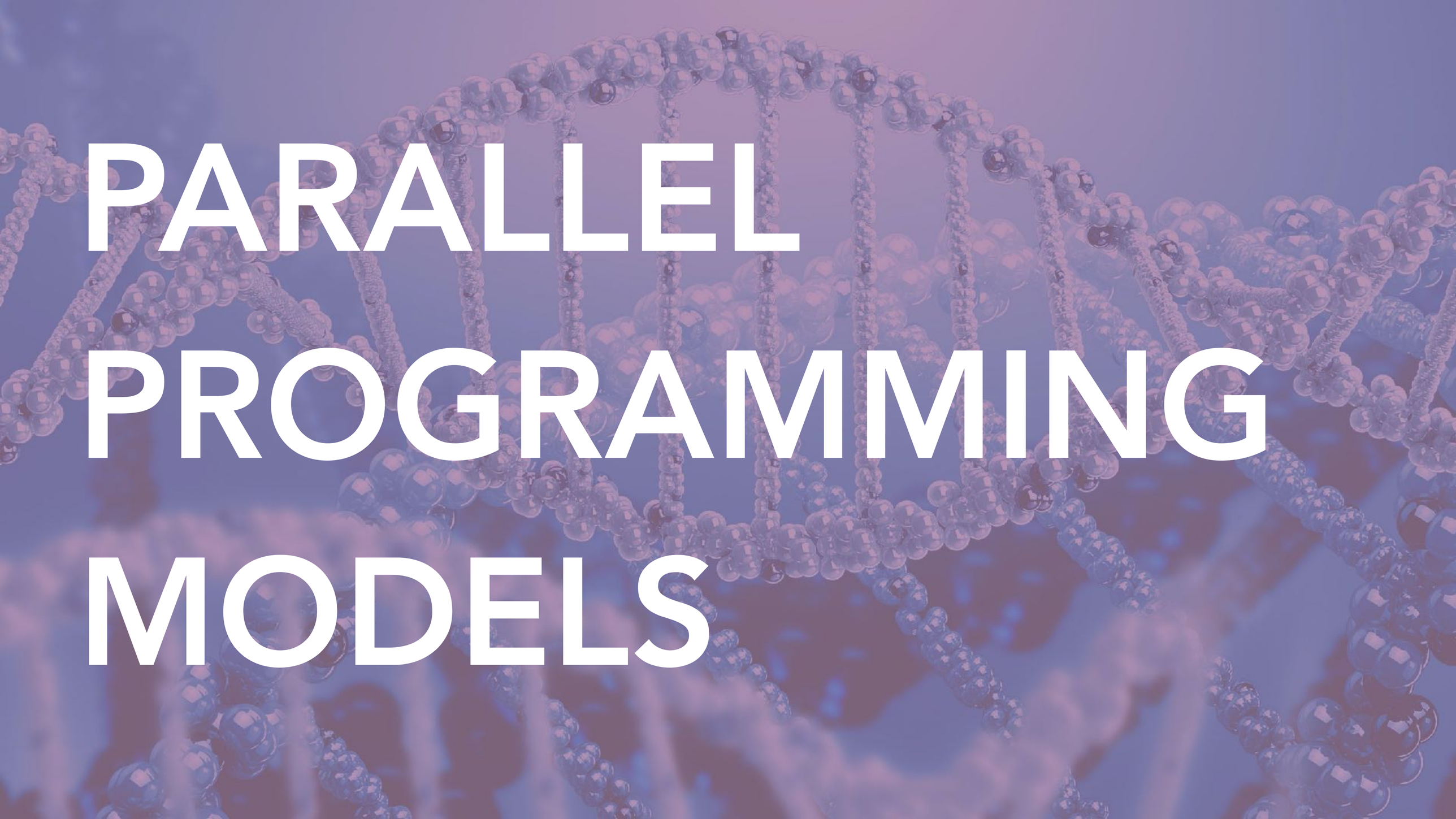


PARALLEL COMPUTER MEMORY ARCHITECTURES

HYBRID MEMORY

- Employ both shared and distributed memory architectures
- Advantage
  - Increased scalability
- Disadvantage
  - Increased programmer complexity




PARALLEL
PROGRAMMING
MODELS

PARALLEL PROGRAMMING MODELS

- Parallel programming models exist as an abstraction above hardware and memory architectures
- These models are NOT specific to a particular type of machine or memory architecture
  - Any of these models can (theoretically) be implemented on any underlying hardware

PARALLEL PROGRAMMING MODELS

- There are several parallel programming models in common use
  - Shared Memory (without threads)
  - Threads
  - Distributed Memory / Message Passing
  - Data Parallel
  - Hybrid
  - Single Program Multiple Data (SPMD)
  - Multiple Program Multiple Data (MPMD)

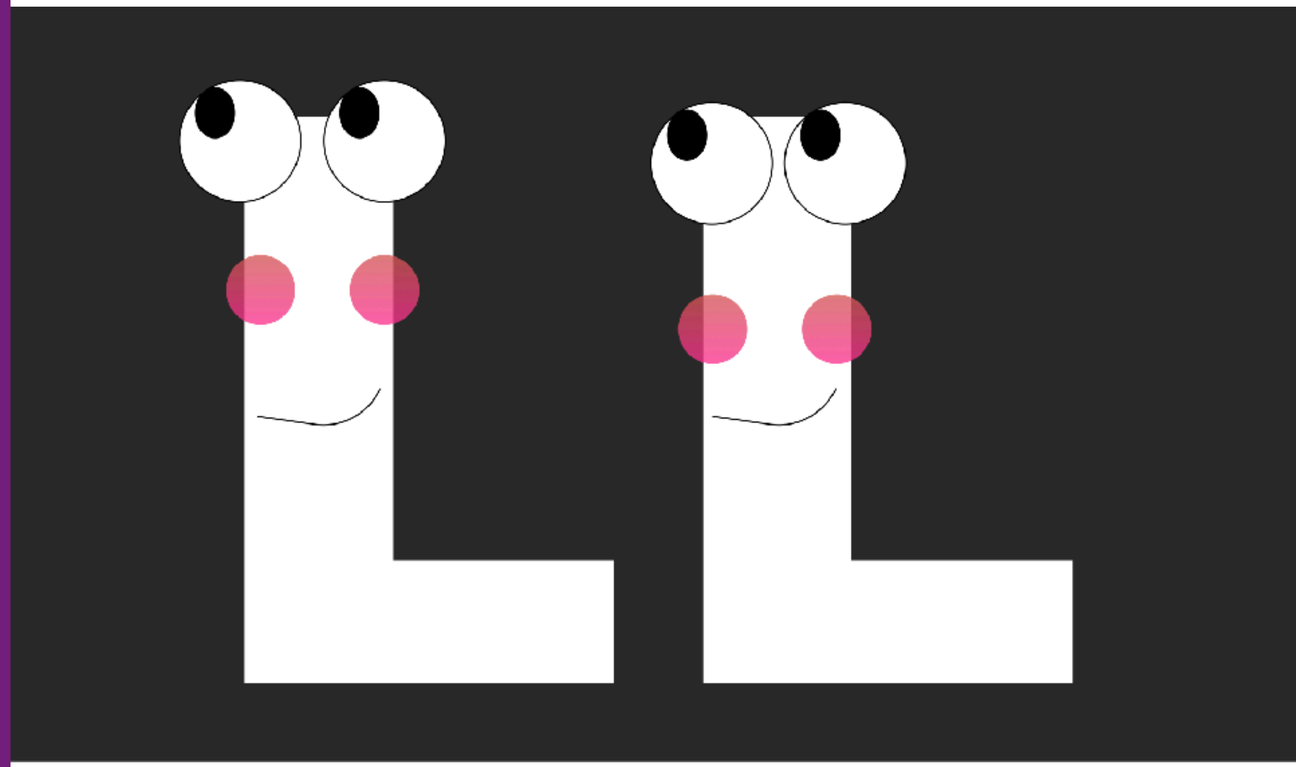
PARALLEL PROGRAMMING MODELS

- Which model to use?
  - This is often a combination of what is available and personal choice
  - There is no "best" model, although there certainly are better implementations of some models over others

Embarrassing Parallel (Many Task)

PARALLEL PROGRAMMING MODELS

- Embarrassingly Parallel
  - Solving many similar but independent tasks simultaneously
  - Little to no coordination (no communication) between tasks
  - Each task can be a simple serial program

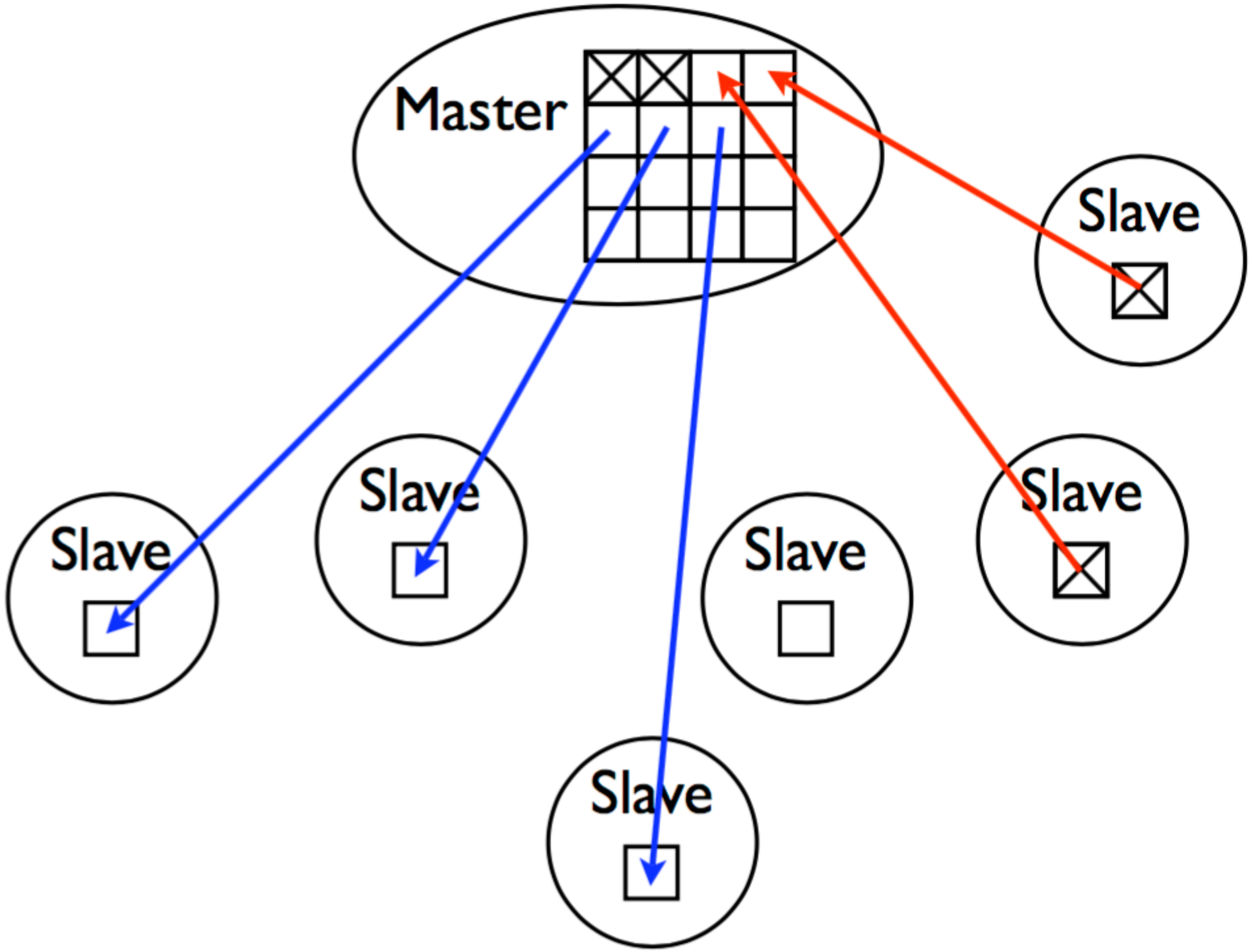


PARALLEL PROGRAMMING MODELS

- This is the “easiest” type of problem to implement in a parallel manner
  - Automatically coordinating many independent calculations
  - Collating the results
- Widely and “proudly” used in many fields

PARALLEL PROGRAMMING MODELS

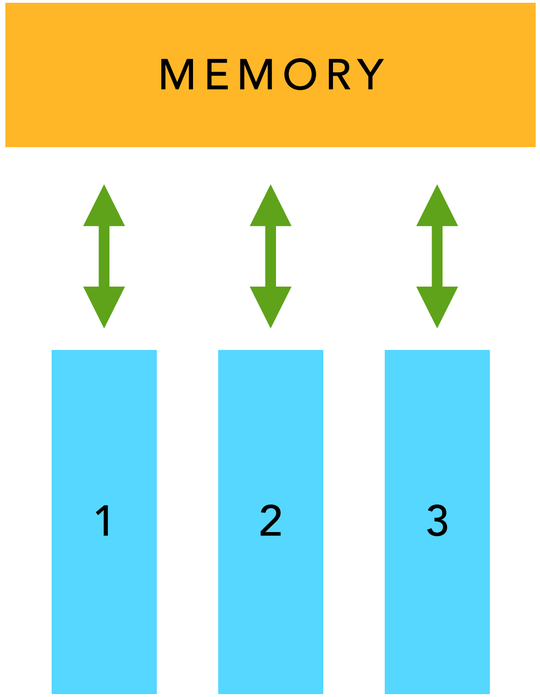
- Master-Slave: Master task assigns slave task
  - Useful version of many-task



Shared Memory Model (without threads)

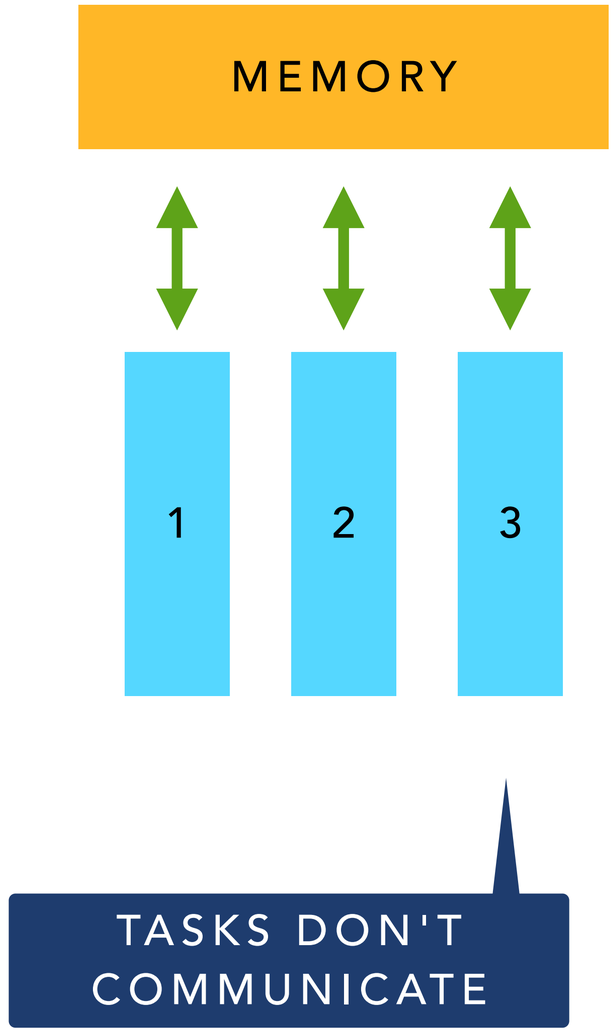
SHARED MEMORY MODEL (WITHOUT THREADS)

- Tasks share a common address space, which they read and write to asynchronously
  - Various mechanisms such as locks / semaphores may be used to control access to the shared memory



SHARED MEMORY MODEL (WITHOUT THREADS)

- Notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks
- Program development can often be simplified
- Difficult to understand and manage data

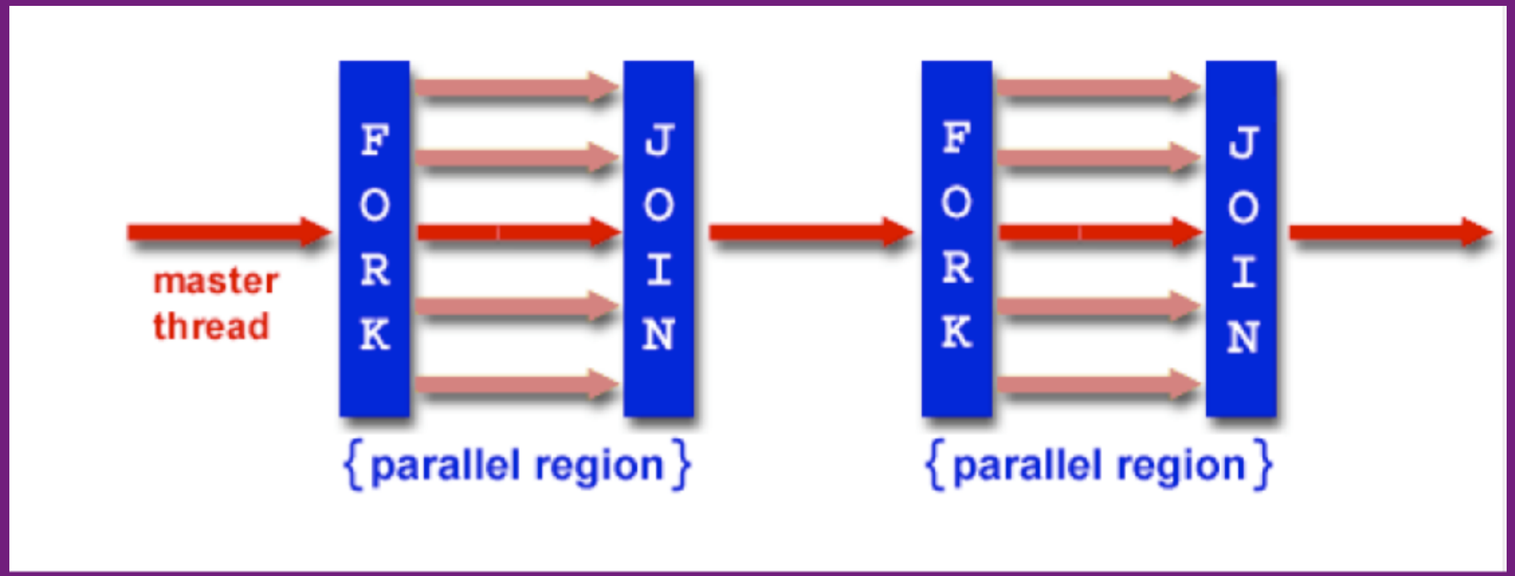


SHARED MEMORY MODEL (WITHOUT THREADS)

- Implementations
- Stand-alone shared memory machines
    - Native operating systems, compilers and/or hardware provide support for shared memory programming
    - Distributed shared memory machines
        - Memory is physically distributed across a network of machines, but made global through specialized hardware and software

Threads Model

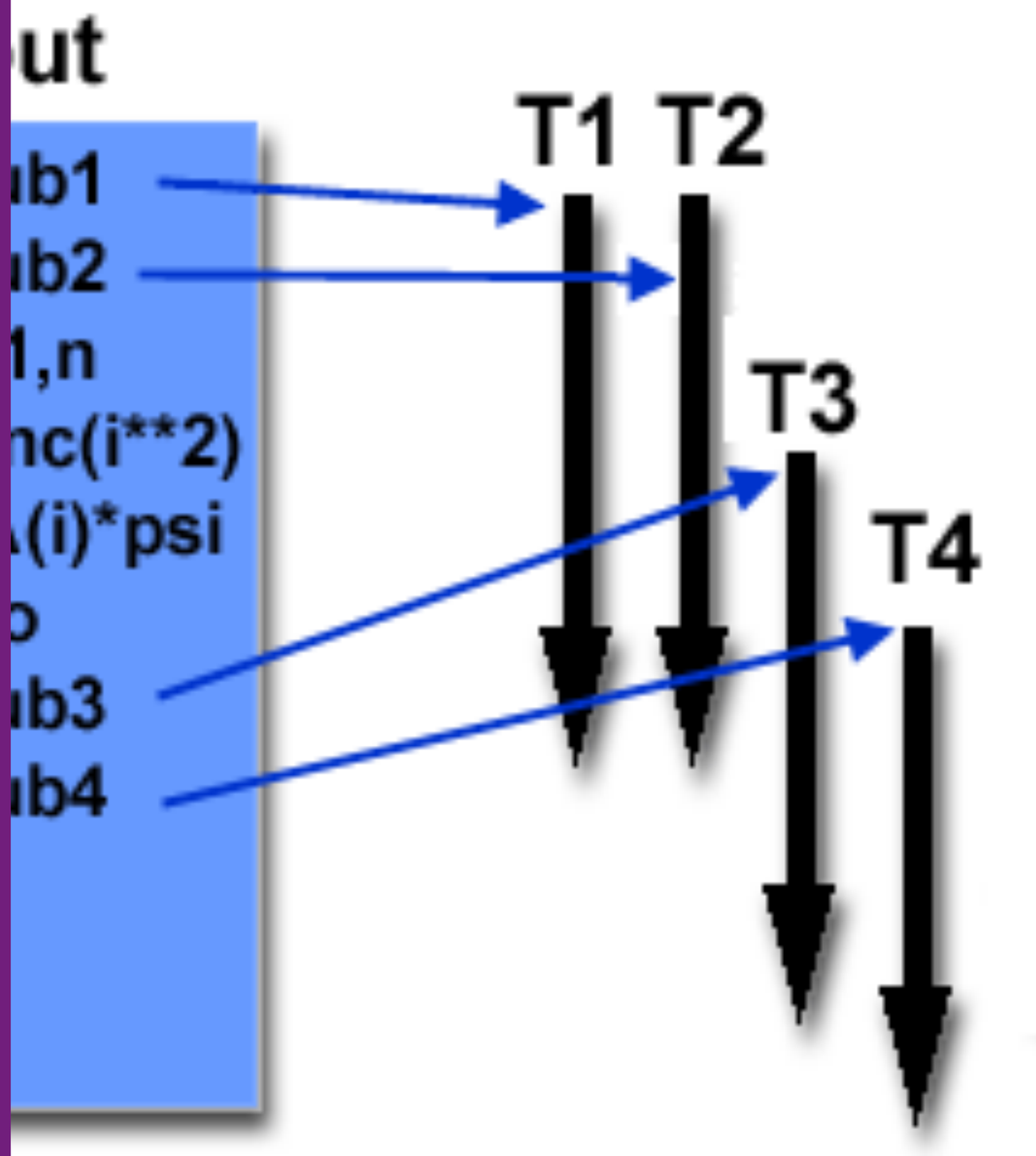
THREADS MODEL



- In the threads model a single "heavy weight" process can have multiple "light weight", concurrent execution paths
  - Threads communicate through global memory

THREADS MODEL

- Example:
  - Main program a.out is scheduled to run by the native operating system ‘heavy weight’
    - a.out loads and acquires all of the necessary system and user resources to run.
    - a.out performs some serial work
    - Then creates a number of tasks (light weight threads) that can be scheduled and run by the operating system concurrently
  - Each thread has local data, but also, shares the entire resources of a.out



THREADS MODEL

- Threads implementations commonly comprise
  - A library of subroutines called from within parallel source code
  - A set of compiler directives imbedded in either serial or parallel source code
- Programmer is responsible for determining the parallelism (although compilers can sometimes help)

THREADS MODEL

- Unrelated standardization efforts have resulted in two very different implementations of threads
  - POSIX Threads
  - OpenMP

Threads Model

- POSIX Threads
  - Library based; requires parallel coding
  - Specified by the IEEE POSIX 1003.1c standard (1995)
  - Commonly referred to as Pthreads
  - Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations
  - Very explicit parallelism
    - Requires significant programmer attention to detail

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
      fprintf(stderr, "usage: threads <loops>\n");
      exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

THREADS MODEL

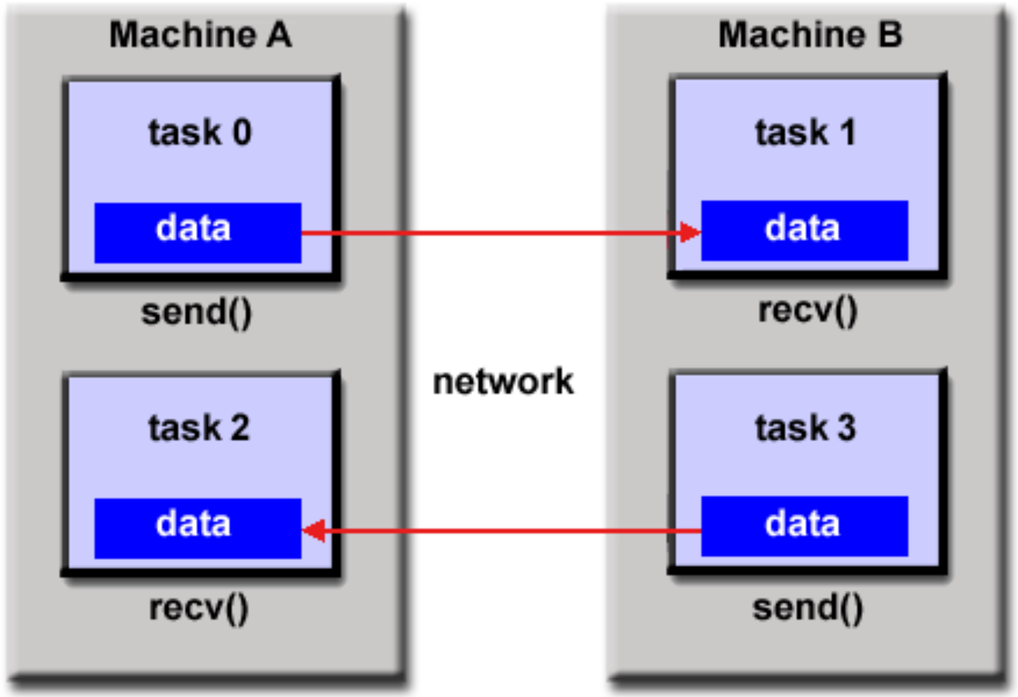
- OpenMP
  - Compiler directive based; can use serial code
  - Jointly defined and endorsed by major vendors
  - Portable / multi-platform
  - Can be very easy and simple to use
    - Provides for "incremental parallelism"

```
serial code
        .
        .
        .

!$OMP PARALLEL DO
do i = 1,N
  A(i)=B(i)+C(i)
enddo
!$OMP END PARALLEL DO

        .
        .
        .
serial code
```

Distributed Memory / Message Passing

DISTRIBUTED MEMORY / MESSAGE PASSING MODEL

- A set of tasks that use their own local memory during computation
- Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines
- Tasks exchange data through communications by sending and receiving messages

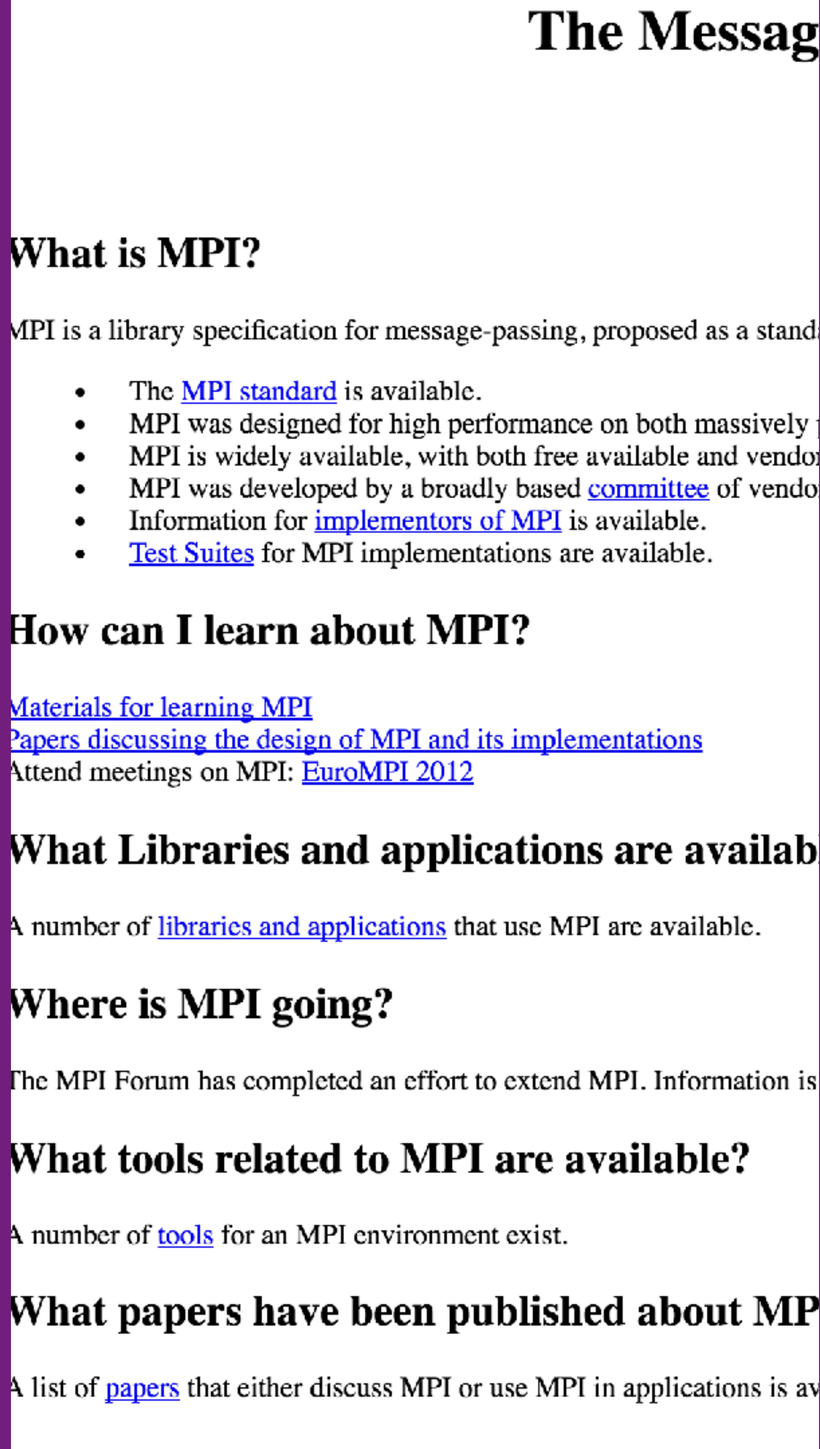


DISTRIBUTED MEMORY / MESSAGE PASSING MODEL

- Implementations
  - Message passing implementations usually comprise a library of subroutines
  - Calls to these subroutines are imbedded in source code
  - Programmer responsible for determining all parallelism
- Message passing libraries have been available since the 1980s
  - Differed substantially from each other making it difficult for programmers to develop portable applications

DISTRIBUTED MEMORY / MESSAGE PASSING MODEL

- MPI is the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work
  - In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations
  - Part 1 of the Message Passing Interface (MPI) was released in 1994
  - Part 2 (MPI-2) was released in 1996 and MPI-3 in 2012
- MPI implementations exist for virtually all popular parallel computing platforms
  - Not all implementations include everything in MPI-1, MPI-2 or MPI-3



The Messag

What is MPI?

MPI is a library specification for message-passing, proposed as a stand

- The MPI standard is available.
- MPI was designed for high performance on both massively
- MPI is widely available, with both free available and vendo
- MPI was developed by a broadly based committee of vendo
- Information for implementors of MPI is available.
- Test Suites for MPI implementations are available.

How can I learn about MPI?

Materials for learning MPI
Papers discussing the design of MPI and its implementations
Attend meetings on MPI: EuroMPI 2012

What Libraries and applications are availab

A number of libraries and applications that use MPI are available.

Where is MPI going?

The MPI Forum has completed an effort to extend MPI. Information is

What tools related to MPI are available?

A number of tools for an MPI environment exist.

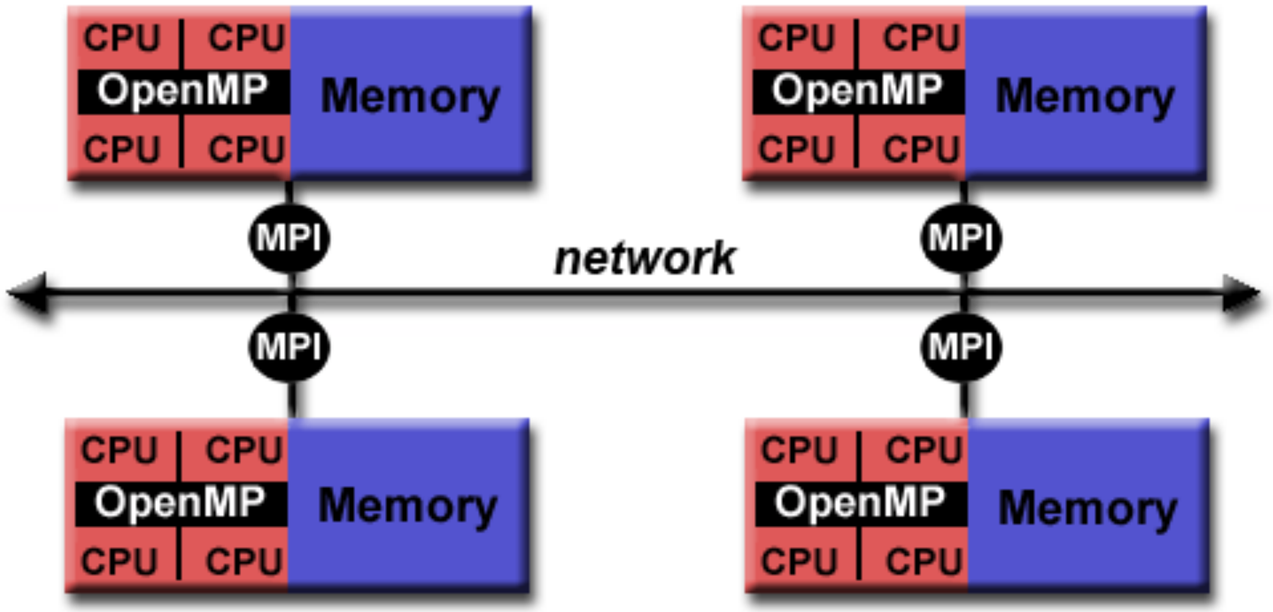
What papers have been published about MP

A list of papers that either discuss MPI or use MPI in applications is av

Hybrid Model

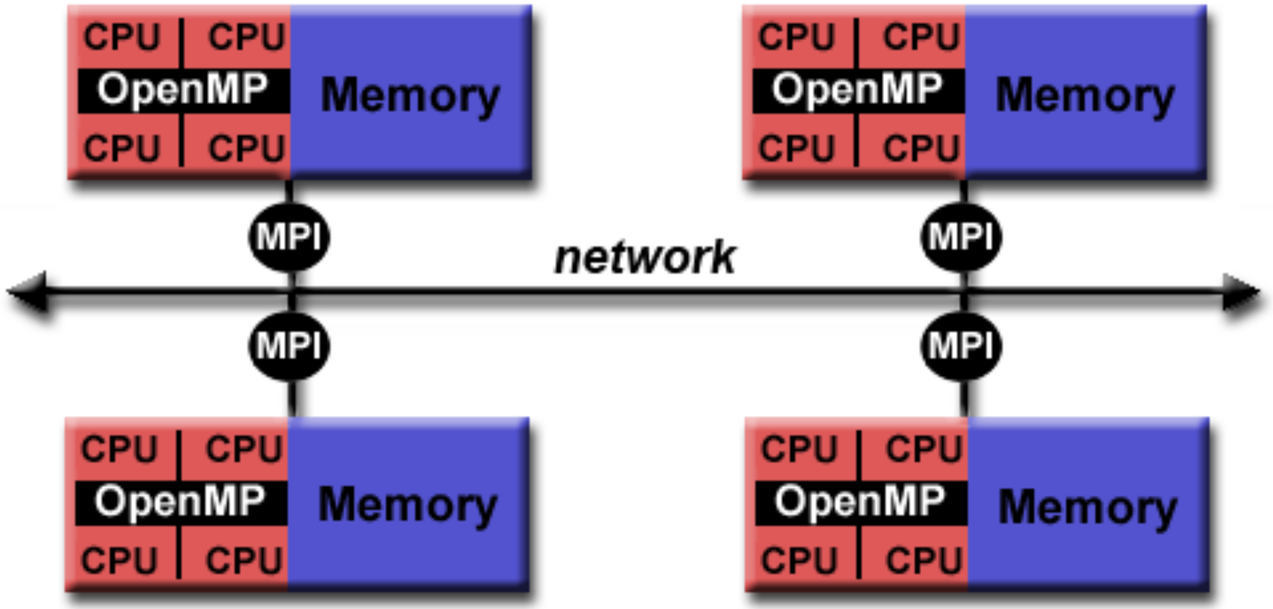
HYBRID MODEL

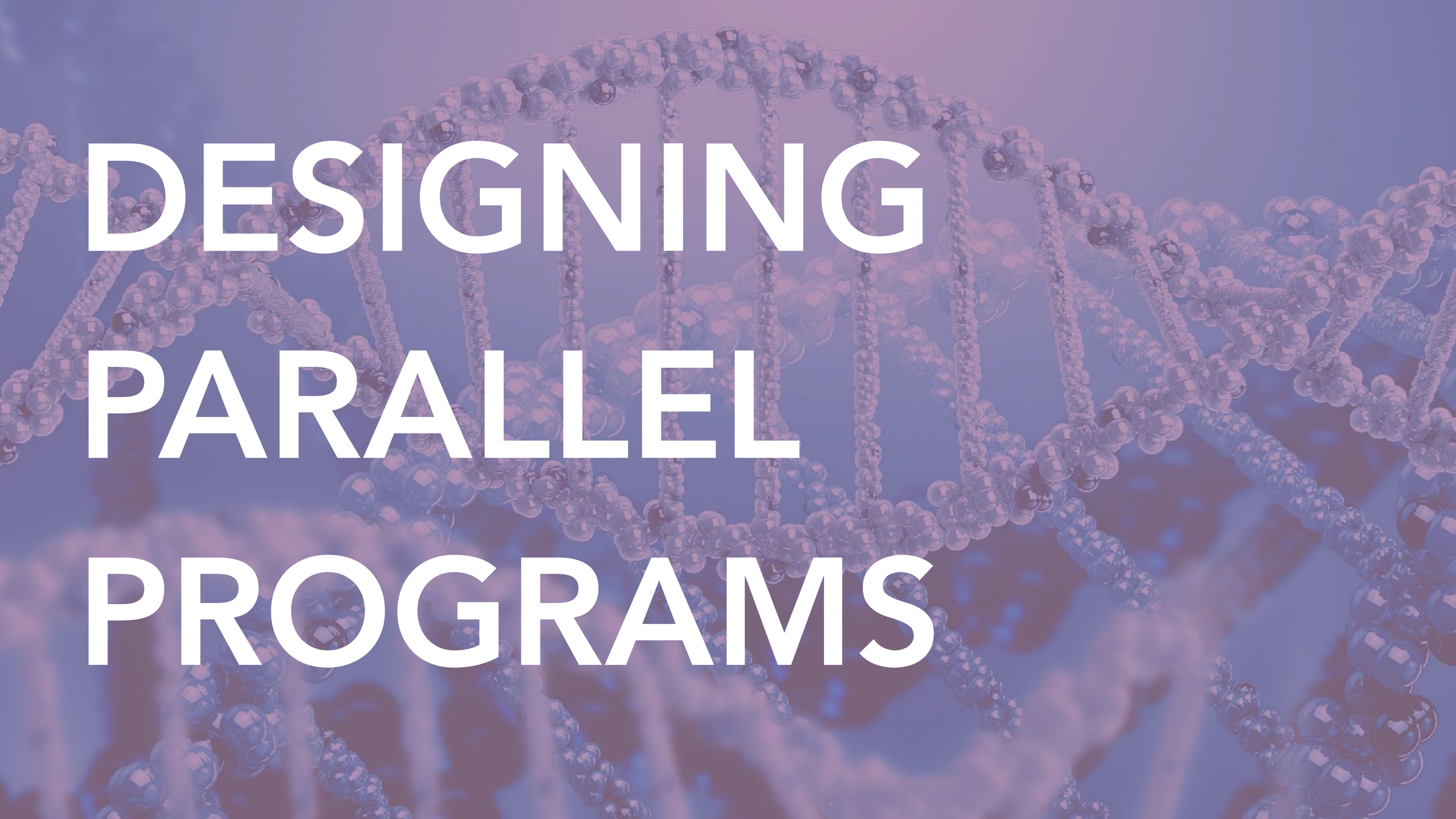
- A hybrid model combines more than one of the previously described programming models
- Common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP)
  - Threads perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI



HYBRID MODEL

- Lends itself well to the increasingly common hardware environment of clustered multi/ many-core machines
- Another similar and increasingly popular example of a hybrid model is using MPI with GPU (Graphics Processing Unit) programming
  - GPUs perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI





DESIGNING PARALLEL PROGRAMS

Automatic Parallelization

AUTOMATIC PARALLELIZATION

- Designing and developing parallel programs has characteristically been a very manual process
- The programmer is typically responsible for both identifying and actually implementing parallelism
  - Manually developing parallel codes is a time consuming, complex, error-prone and iterative process
  - Tools exist to convert serial programs into parallel programs 🤔

AUTOMATIC PARALLELIZATION

- A parallelizing compiler generally works in two different ways
  - Programmer Directed
    - Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
    - May be able to be used in conjunction with some degree of automatic parallelization also.
    - The most common compiler generated parallelization is done using on-node shared memory and threads (such as OpenMP)

AUTOMATIC PARALLELIZATION

- A parallelizing compiler generally works in two different ways
  - Fully Automatic
    - Compiler analyzes the source code and identifies opportunities for parallelism
    - The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance
    - Loops (do, for) are the most frequent target for automatic parallelization

Automatic Parallelization

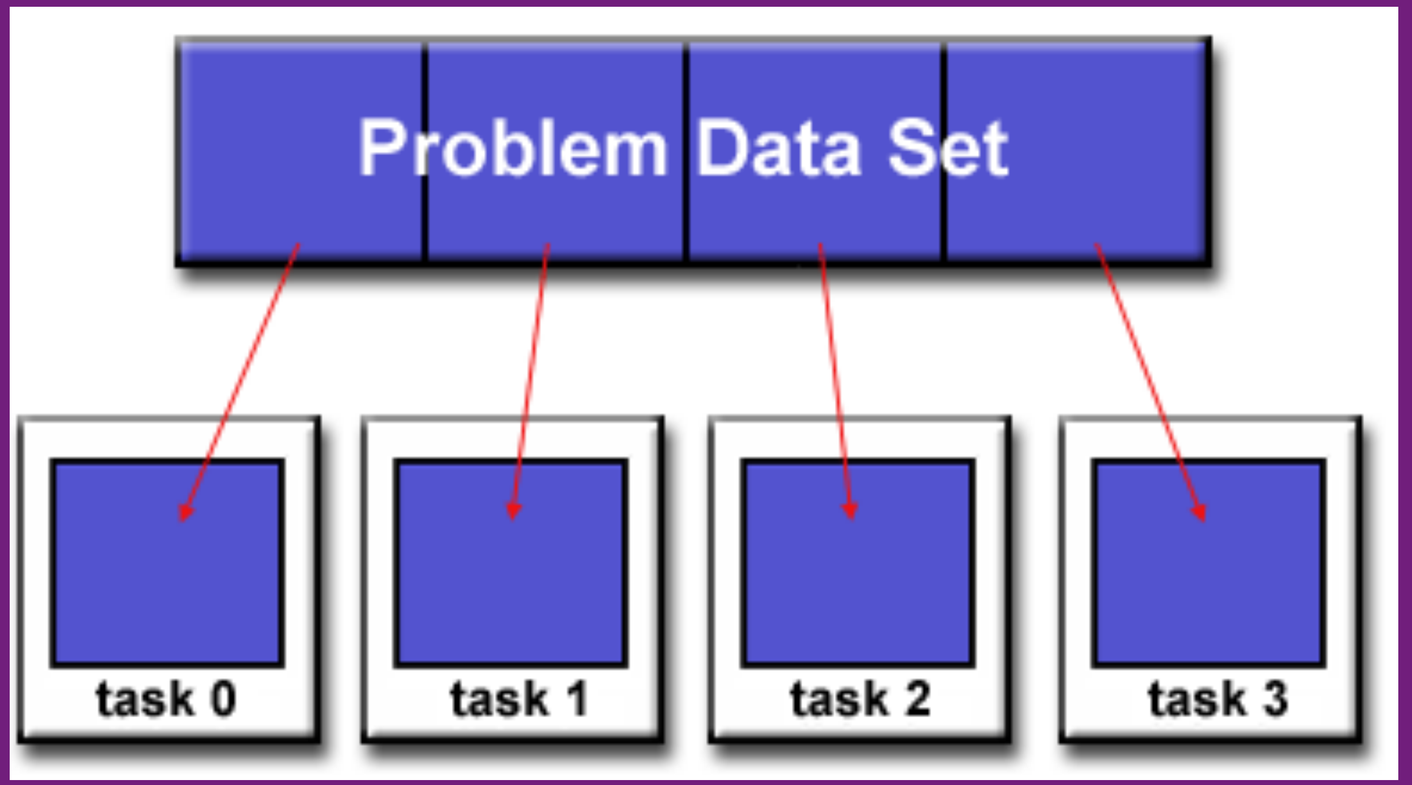
- Important caveats that apply to automatic parallelization
  - Wrong results may be produced
  - Performance may actually degrade
  - Much less flexible than manual parallelization
  - Limited to a subset (mostly loops) of code
  - May actually not parallelize code if the compiler analysis suggests there are inhibitors or the code is too complex

Manual Parallelization

MANUAL PARALLELIZATION

- Understand the problem and the program
  - Can it be parallelized?
  - Identify the program's hotspots
    - Where is work being done?
  - Identify bottlenecks in the program
    - Is the program waiting for data? Other I/O issues?
  - Is data dependence an issue?
  - New algorithm?

MANUAL PARALLELIZATION



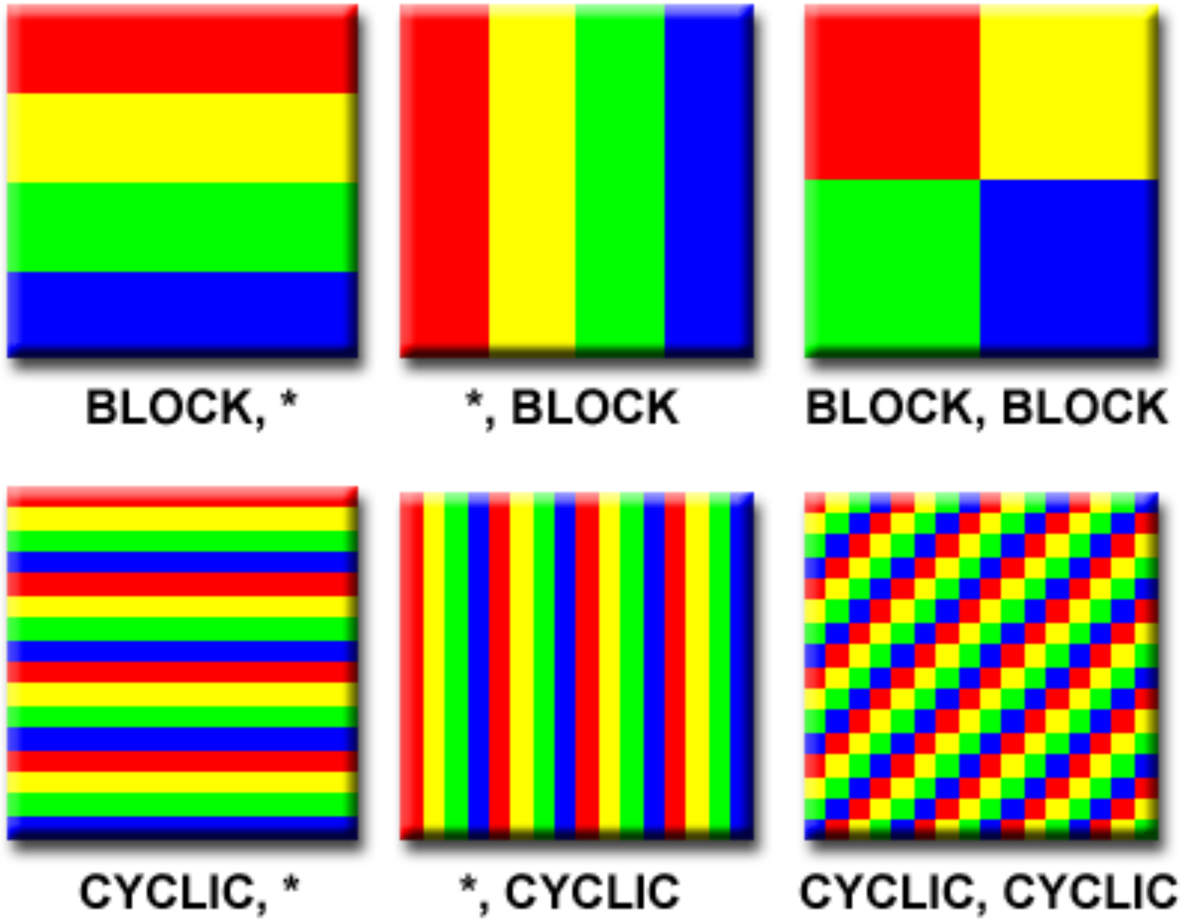
- Partitioning the problem

MANUAL PARALLELIZATION

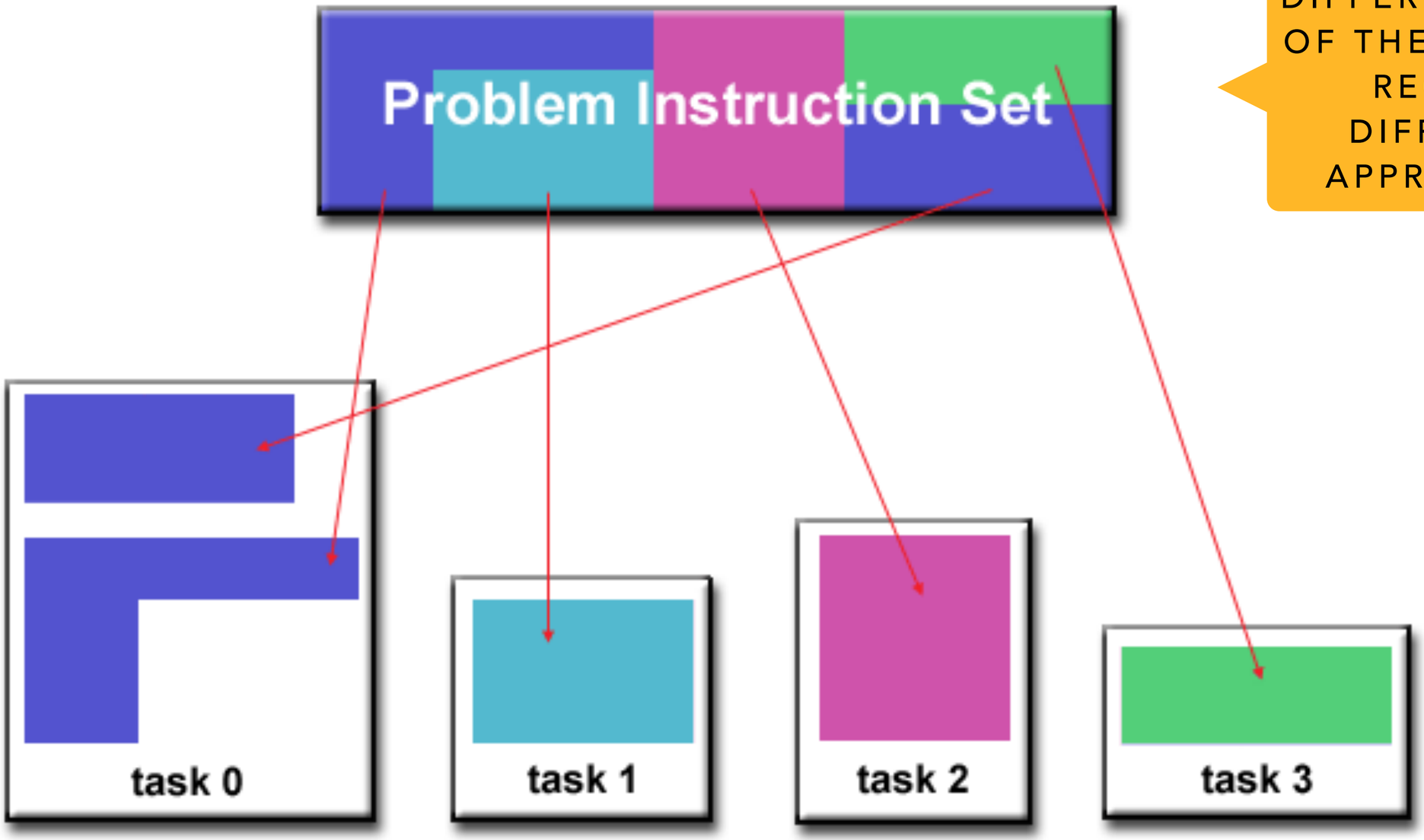
STRATEGIES FOR MANAGING THE EXECUTION AND DATA SEGMENTATION

1D

2D



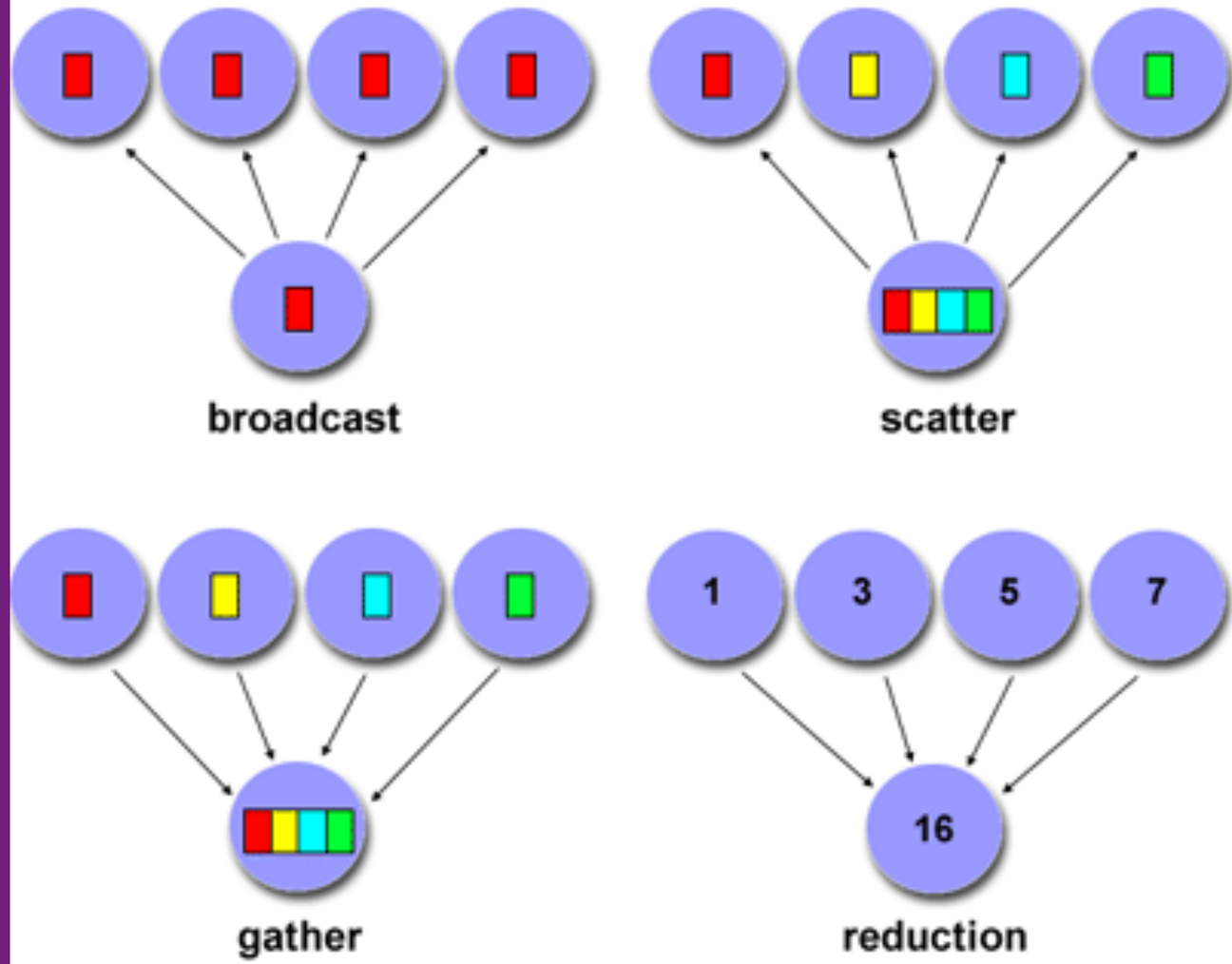
MANUAL PARALLELIZATION



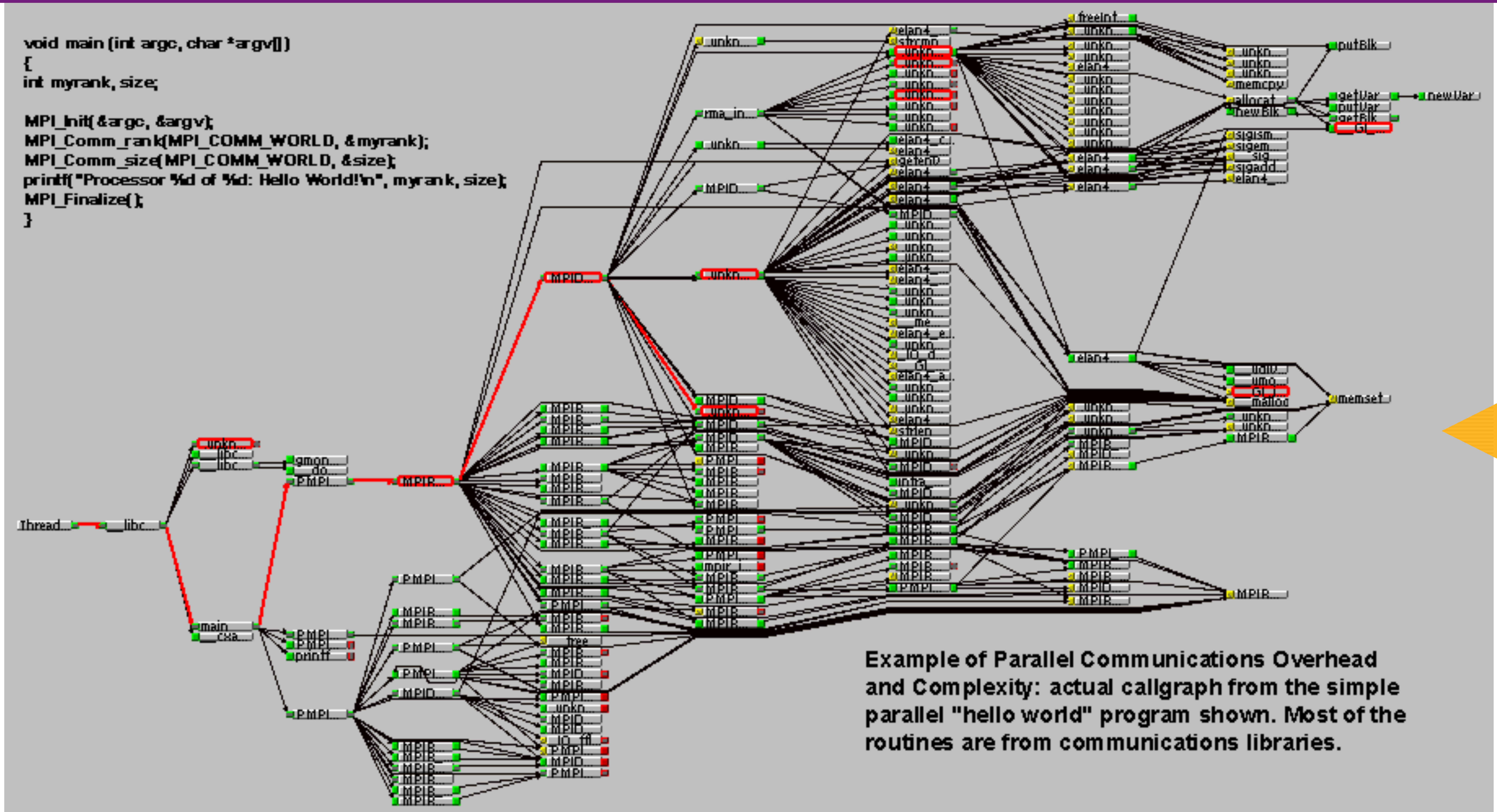
DIFFERENT PARTS OF THE PROBLEM REQUIRE DIFFERENT APPROACHES

MANUAL PARALLELIZATION

- Identify need for communications between tasks
  - You DON'T need communications
    - Many task
  - You DO need communications
    - Threads, MPI
    - Cost of communications
    - Scope of communications



MANUAL PARALLELIZATION



Example of Parallel Communications Overhead and Complexity: actual callgraph from the simple parallel "hello world" program shown. Most of the routines are from communications libraries.

OVERHEAD AND COMPLEXITY

Dependencies

DEPENDENCIES

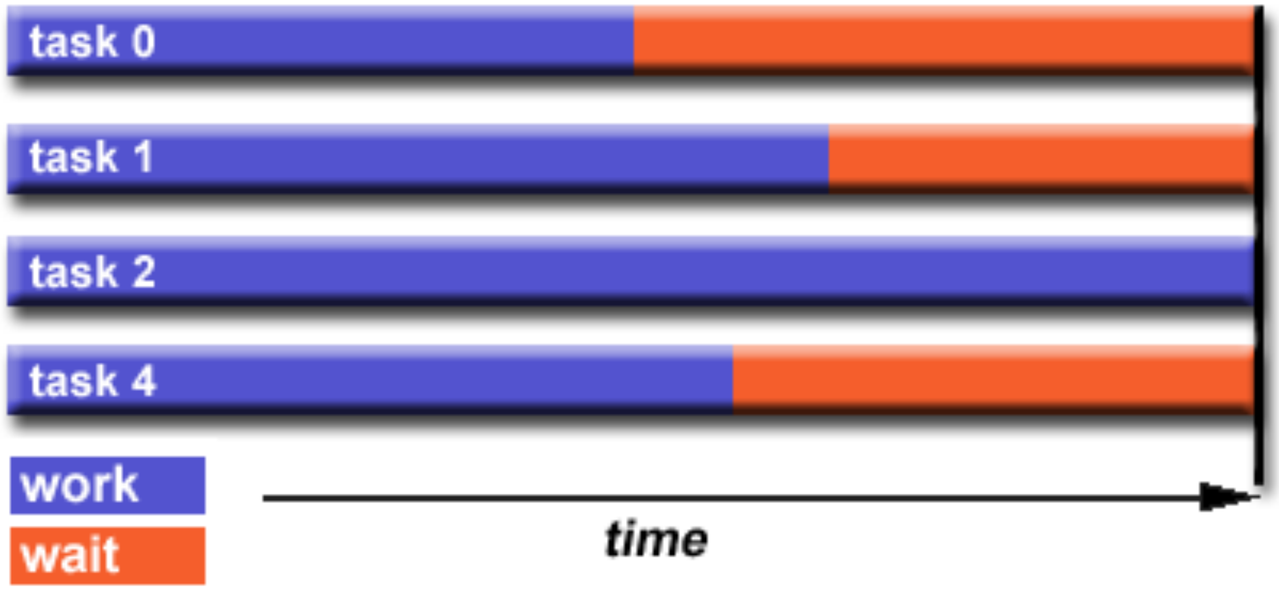
- Data Dependencies can affects the results/execution of the program
- One of the primary inhibitors to parallelism
  - When, where does data need to be?
  - Later stages dependent on earlier stages?

DEPENDENCIES

- How to Handle Data Dependencies
  - Distributed memory architectures
    - Communicate required data at synchronization points
    - Pre/post computation
  - Shared memory architectures
    - Synchronize read/write operations between tasks

DEPENDENCIES

- Load Balancing
  - Distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time
  - Minimize task idle time
- Slowest task will determine the overall performance



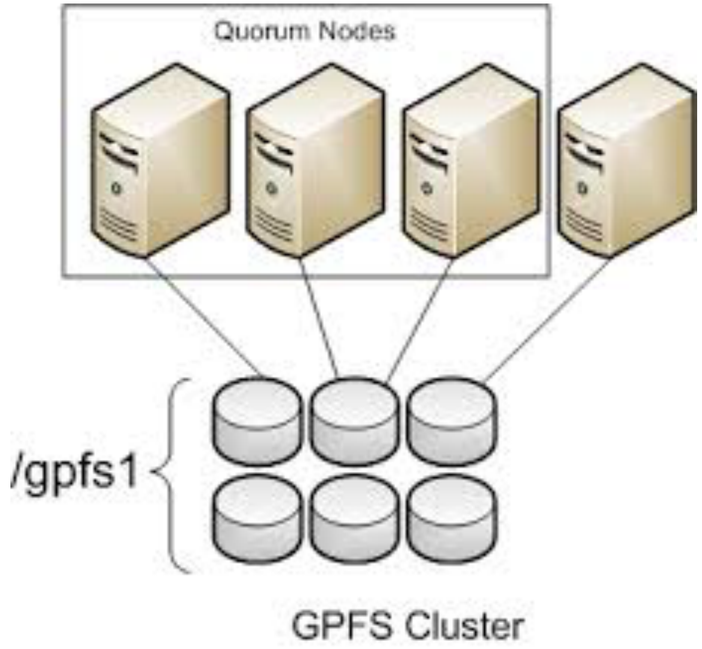
DEPENDENCIES

- I/O operations
  - Require orders of magnitude more time than memory operations
  - Write operations can result in file overwriting
  - Read operations can be affected by the file server's ability to handle multiple read requests at the same time

DEADLOCKS

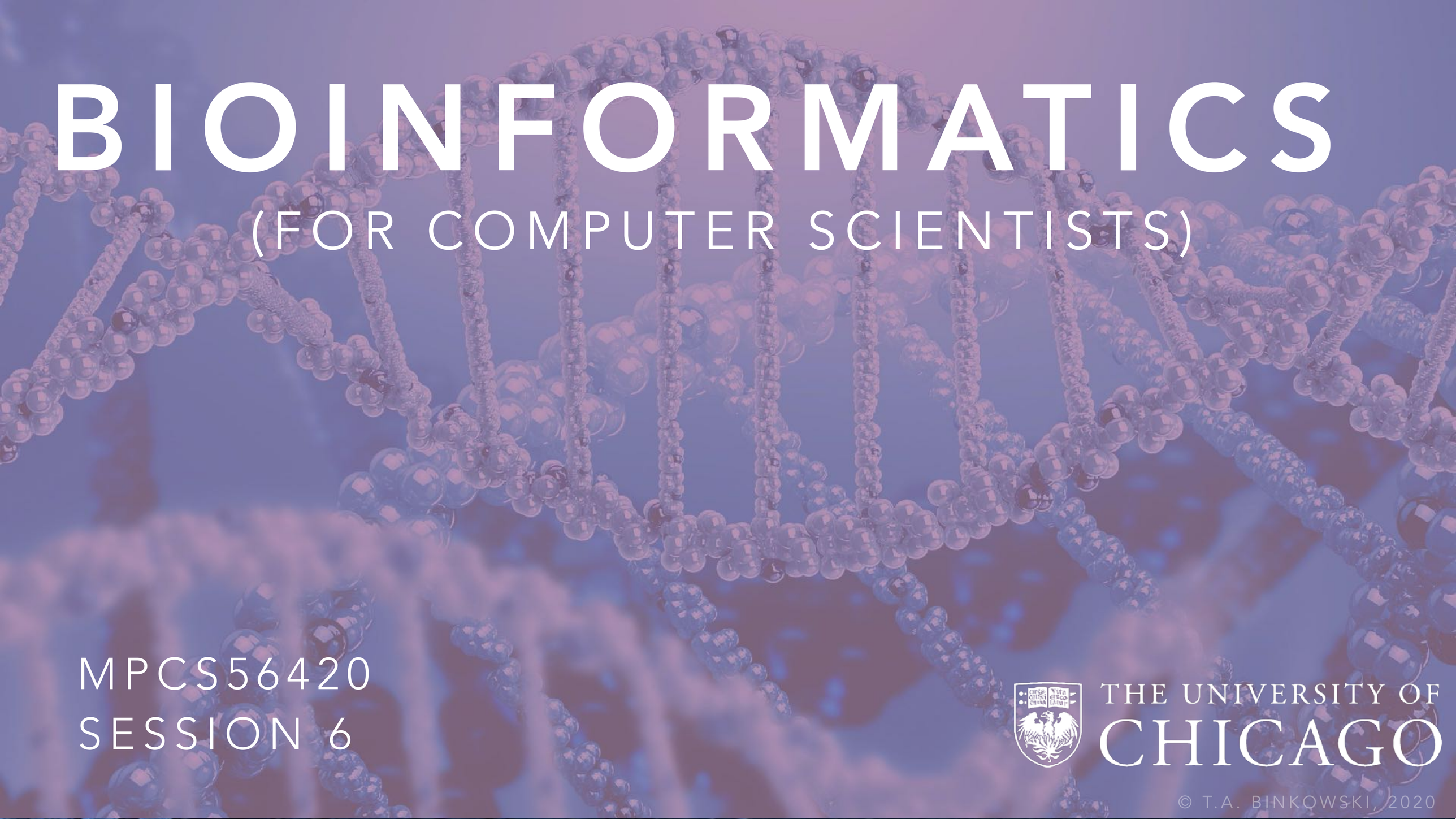
DEPENDENCIES

- Parallel file systems can address many of these
- Everyone can read/write as single unit of storage
- GPFS
  - General Parallel File System for AIX (IBM)



DEPENDENCIES

- Tips:
  - Reduce overall I/O as much as possible
  - Use parallel file system (if available)
  - Writing large chunks of data rather than small chunks is usually significantly more efficient
  - Confine I/O to specific serial portions of the job and then use parallel communications to distribute data to parallel tasks
  - Aggregate I/O operations across tasks
    - Have a subset of tasks perform it



BIOINFORMATICS

(FOR COMPUTER SCIENTISTS)

MPCS56420
SESSION 6

THE UNIVERSITY OF CHICAGO