# Parallel Processing

Ed Upchurch
April 2011

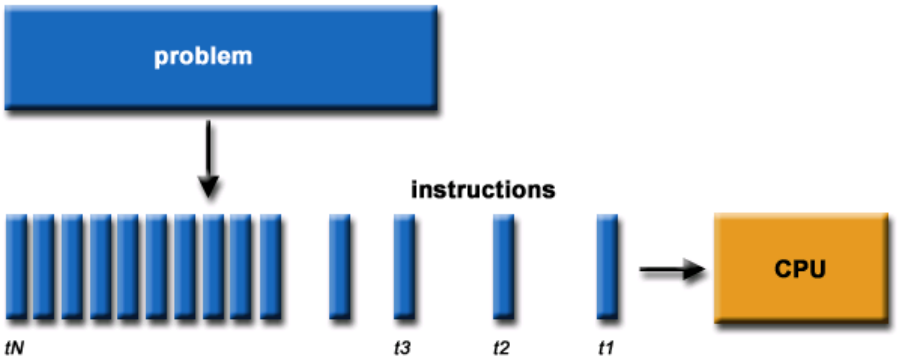
Traditionally, software has been written for **serial** computation:

* Run on a single computer having a single Central Processing Unit (CPU);

* Problem is broken into a discrete series of instructions.

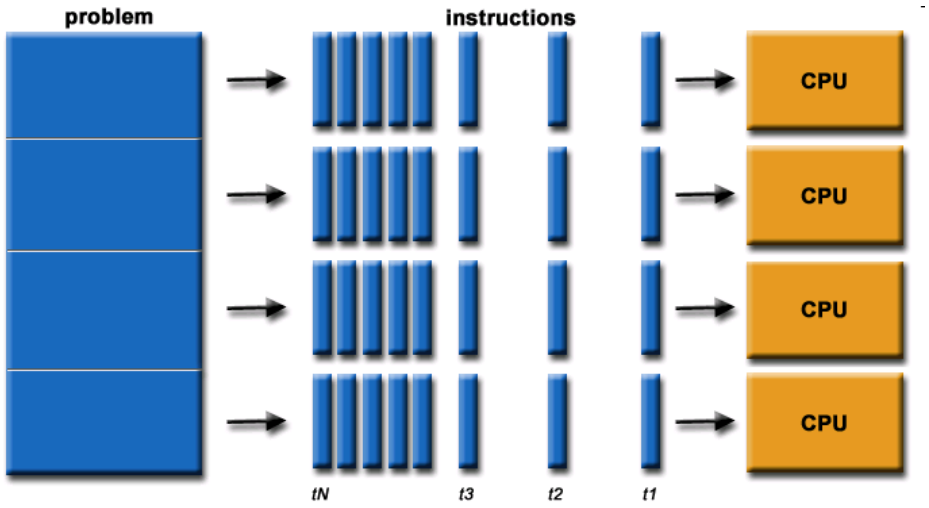
* Instructions are executed one after another.

* Only one instruction may execute at any moment in time.



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

* Run using multiple CPUs
* 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 CPUs



# Motivation

# Two Types of Goals

- Get a very large number (millions) of machines to work together to solve really really big problems – big data, lots of transactions, lots of computations (SETI at home) **(High Capacity)**
- Get a very large number (millions) of machines to work together to solve really big problems fast: real time needs, event threats, fusion experiments **(High Capability)**

# Defining the 3 C's …

- **Main Classes of computing :**
  - ***High capacity parallel computing*** : A strategy for employing distributed computing resources to achieve high throughput processing among decoupled tasks. Aggregate performance of the total system is high if sufficient tasks are available to be carried out concurrently on all separate processing elements. No single task is accelerated.
  - ***High capability parallel computing*** : A strategy for employing tightly couple structures of computing resources to achieve reduced execution time of a given application through partitioning in to concurrently executable tasks.
  - ***Cooperative computing*** **:** A strategy for employing moderately coupled ensemble of computing resources to increase size of the data set of a user application while limiting its execution time.

# Defining the 3 C's …

- High capacity computing systems emphasize the overall work performed over a fixed time period. Work is defined as the aggregate amount of computation performed across all functional units, all threads, all cores, all chips, all coprocessors and network interface cards in the system.
- High capability computing systems emphasize improvement (reduction) in execution time of a single user application program of fixed data set size.

Adapted from : High-performance throughput computing
S Chaudhry, P Caprioli, S Yip, M Tremblay - IEEE Micro, 2005 - doi.ieeecomputersociety.org

# Why Use Parallel Computing?

- **Save time and/or money**
  - In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.
- **Solve larger problems:** Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.
  - "Grand Challenge" (en.wikipedia.org/wiki/Grand_Challenge) problems requiring PetaFLOPS and PetaBytes of computing resources.
  - Web search engines/databases processing millions of transactions per second
- **Provide concurrency:** A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.
  - the Access Grid (www.accessgrid.org) provides a global collaboration network where people from around the world can meet and conduct work "virtually".
- **Use of non-local resources:** Using compute resources on a wide area network, or even the Internet when local compute resources are scarce.
  - SETI@home (setiathome.berkeley.edu) uses over 330,000 computers for a compute power over 528 TeraFLOPS (as of August 04, 2008)
  - Folding@home (folding.stanford.edu) uses over 340,000 computers for a compute power of 4.2 PetaFLOPS (as of November 4, 2008)

# Why Use Parallel Computing? (continued)

- **Limits to serial computing**
  - **Both physical and practical reasons pose significant constraints to simply building ever faster serial computers**
    - **Transmission speeds** - the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/ nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.
    - **Limits to miniaturization** - processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
      - we are now at 90 nm and some 45 nm
      - quantum effects such a leakage are making it very expensive to go from 90nm to 45nm
    - **Economic limitations** - it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.
- **Current computer architectures are increasingly relying upon hardware level parallelism to improve performance**
  - Multiple execution units
  - Pipelined instructions
  - Multi-core

# Classes of problems that require faster processing

- **Simulation and Modeling**
  - Successive approximations
  - More calculations, more precise
- **Problems dependent on computations / manipulations of large amounts of data**
  - Image and Signal Processing
  - Entertainment (Image Rendering)
  - Database and Data Mining
  - Seismic
- **Grand Challenge Problems**
  - Climate Modeling
  - Fluid Turbulence
  - Pollution Dispersion
  - Ocean Circulation
  - Quantum Chromodynamics
  - Semiconductor Modeling
  - Superconductor Modeling
  - Combustion Systems
  - Vision & Cognition

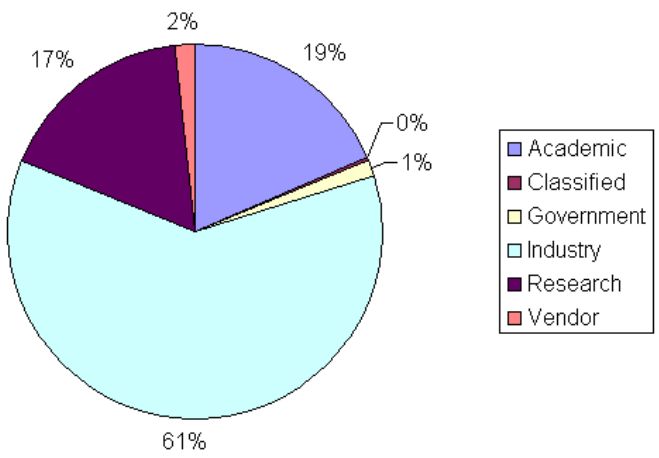
# Uses for Parallel Computing

- Historically "the high end of computing", and has been used to model difficult scientific and engineering problems
  - Atmosphere, Earth, Environment
  - Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
  - Bioscience, Biotechnology, Genetics
  - Chemistry, Molecular Sciences
  - Geology, Seismology
  - Mechanical Engineering - from prosthetics to spacecraft
  - Electrical Engineering, Circuit Design, Microelectronics
  - Computer Science, Mathematics

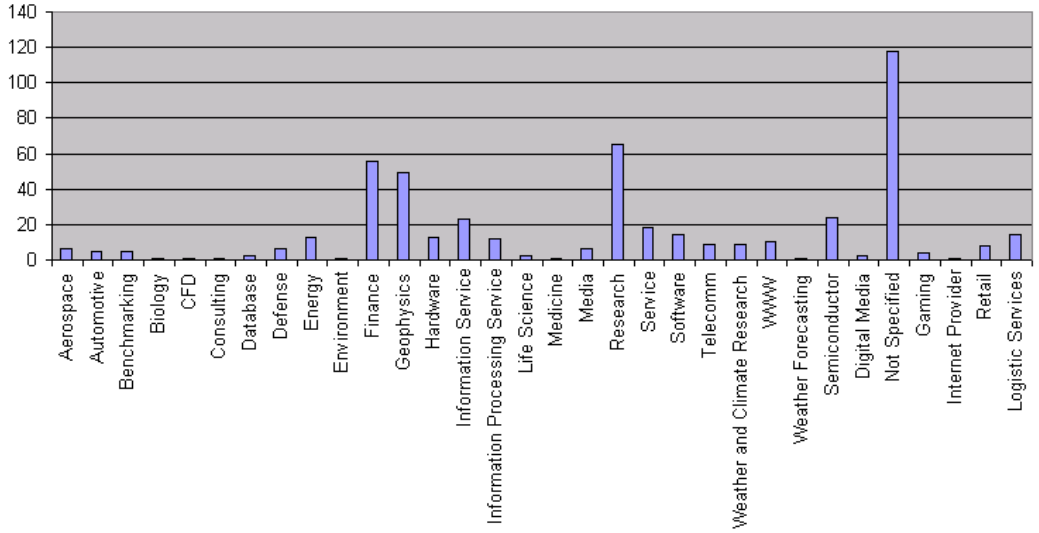
# Commercial Applications Today

- These applications require the processing of large amounts of data in sophisticated ways.
  - Databases, data mining
  - Oil exploration
  - Web search engines, web based business services
  - Medical imaging and diagnosis
  - Pharmaceutical design
  - Management of national and multi-national corporations
  - Financial and economic modeling
  - Advanced graphics and virtual reality, particularly in the entertainment industry
  - Networked video and multi-media technologies
  - Collaborative work environments

## Who's Doing Parallel Computing?




## What Are They Using it For?



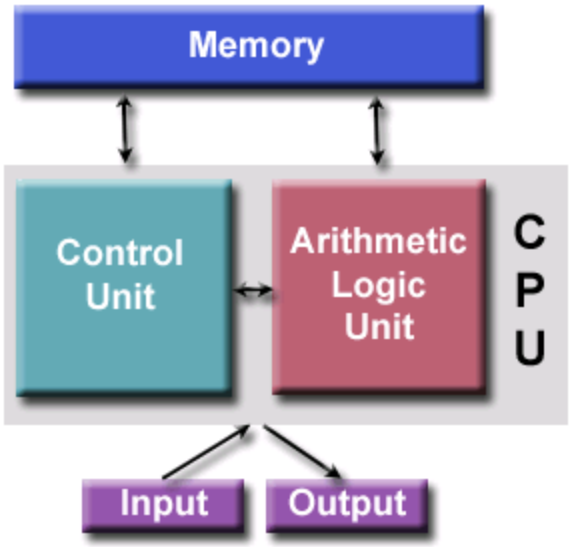
# State of the Art in Bigness

1. Greater than 1 PetaFLOP (> 1.5PF)
2. More than hundred thousand processors (cores)
3. “THE PROBLEM” runs many days
4. Machine Mean Time Between Failures (MTBF) about 56 hours

Items 3 & 4 => Need a **very good checkpoint restart capability** => good I/O as the footprint reaches petaBytes

# Some Basics

# von Neumann Architecture
# 1945




Comprised of four main components:

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

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 simply information to be used by the program

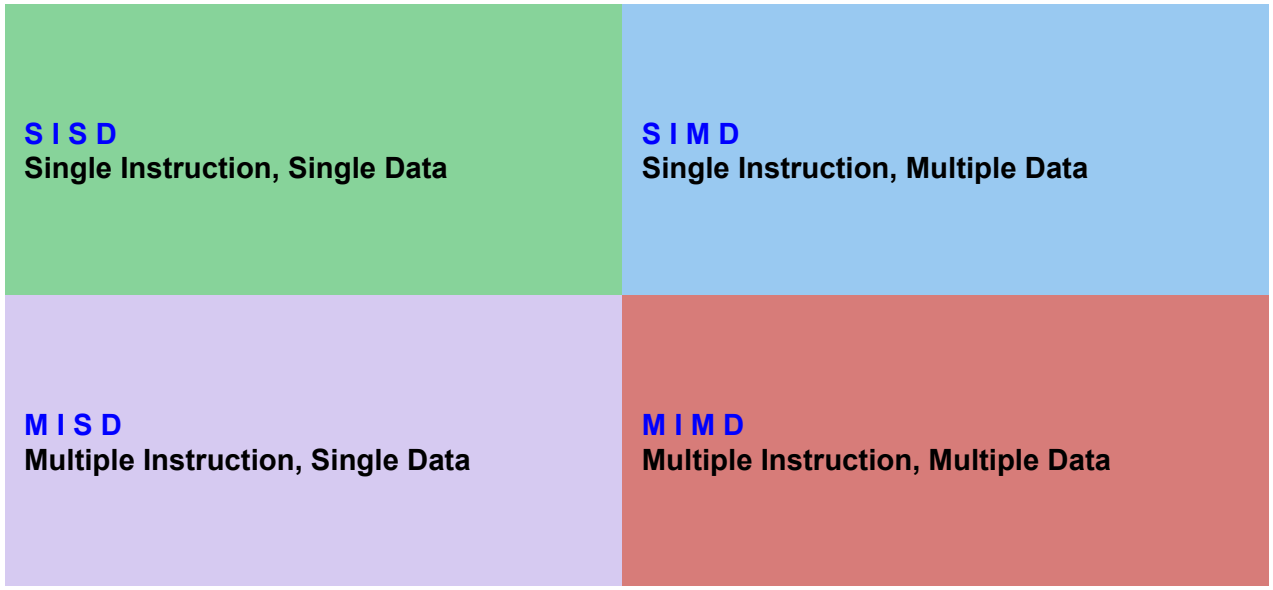
Control unit fetches instructions/data from memory, decodes the instructions and then ***sequentially*** coordinates operations to accomplish the programmed task.

Arithmetic Unit performs basic arithmetic operations

Input/Output is the interface to the human operator

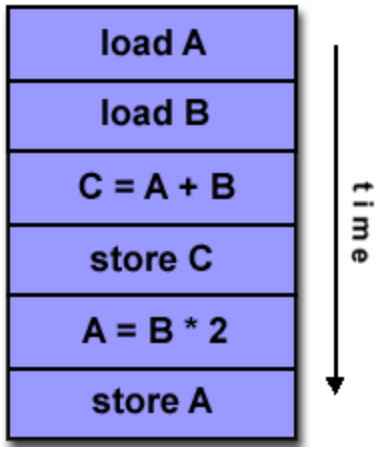
# Flynn's Classical Taxonomy
# 1966

Flynn's taxonomy distinguishes multi-processor computer architectures according to two independent dimensions of ***Instruction*** and ***Data***.
Each of these dimensions can have only one of two possible states:
***Single*** or ***Multiple***.



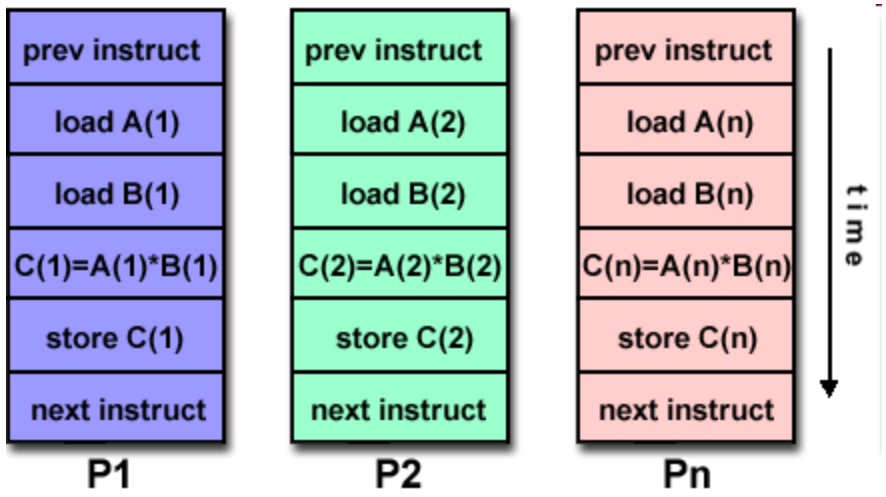
**Single Instruction, Single Data (SISD)**

- A **serial** (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and even today, the most common type of computer
- Examples: older generation mainframes, minicomputers and workstations; most modern day PCs.



**Single Instruction, Multiple Data (SIMD)**

- A type of **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
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines
- Examples:
- Processor Arrays: Connection Machine 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.**



**Multiple Instruction, Single Data (MISD):**

A single data stream is fed into multiple processing units.

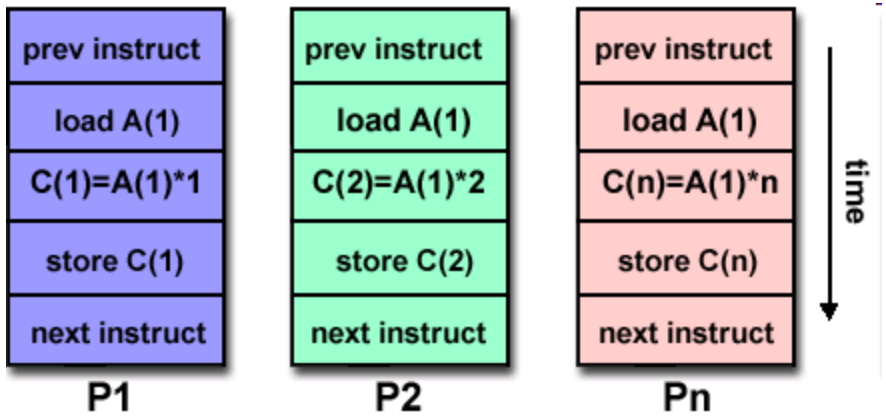
Each processing unit operates on the data independently via independent instruction streams.

Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).

Some conceivable uses might be:

multiple frequency filters operating on a single signal stream

multiple cryptography algorithms attempting to crack a single coded message.



**Multiple Instruction, Multiple Data (MIMD)**

Currently, the **most common type of parallel computer**. Most modern computers fall into this category.

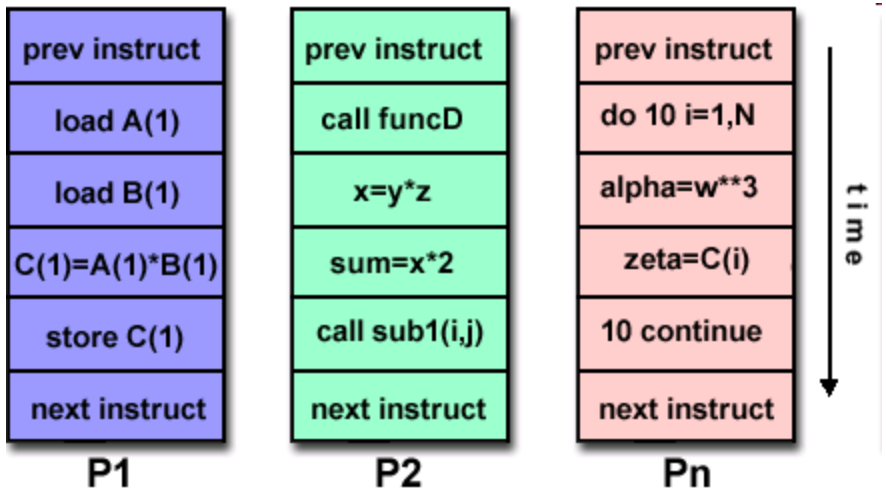
Multiple Instruction: every processor may be executing a different instruction stream

Multiple Data: every processor may be working with a different data stream

Execution can be synchronous or asynchronous, deterministic or non-deterministic

Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.

Note: many MIMD architectures also include SIMD execution sub-components



# Class Example

- What kind of machine was the class room example?
  - A. SISD
  - B. SIMD
  - C. MISD
  - D. MIMD

# Some Terminology

- **Task**
  - A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.
- **Parallel Task**
  - A task that can be executed by multiple processors safely (yields correct results)
- **Serial Execution**
  - Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, virtually all parallel tasks will have sections of a parallel program that must be executed serially.
- **Parallel Execution**
  - Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.
- **Pipelining**
  - Breaking a task into steps performed by different processor units, with inputs streaming through, much like an assembly line; a type of parallel computing.

# Some Terminology

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

- **Symmetric Multi-Processor (SMP)**
  - Hardware architecture where multiple processors share a single address space and access to all resources; shared memory computing.

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

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

- **Synchronization**
  - The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point. Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.

# Some Terminology

- **Granularity**
  - In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
    - ***Coarse:*** relatively large amounts of computational work are done between communication events
    - ***Fine:*** relatively small amounts of computational work are done between communication events
- **Observed Speedup**
  - Observed speedup of a code which has been parallelized, defined as:
  - **wall-clock time of serial execution/ wall-clock time of parallel execution**
  - One of the simplest and most widely used indicators for a parallel program's performance.
- **Parallel Overhead**
  - The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:
    - Task start-up time
    - Synchronizations
    - Data communications
    - Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
    - Task termination time

# Some Terminology

- **Massively Parallel**
  - Refers to the hardware that comprises a given parallel system - having many processors. The meaning of "many" keeps increasing, but currently, the largest parallel computers can be comprised of processors numbering in the hundreds of thousands.
- **Embarrassingly Parallel**
  - Solving many similar, but independent tasks simultaneously; little to no need for coordination between the tasks.
- **Scalability**
  - Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:
    - Hardware - particularly memory-cpu bandwidths and network communications
    - Application algorithm
    - Parallel overhead related
    - Characteristics of your specific application and coding
- **Multi-core Processors**
  - Multiple processors (cores) on a single chip.
- **Cluster Computing**
  - Use of a combination of commodity units (processors, networks or SMPs) to build a parallel system.
- **Supercomputing / High Performance Computing**
  - Use of the world's fastest, largest machines to solve large problems.

# Key Terms and Concepts

- *Speedup :* Relative reduction of execution time of a fixed size workload through parallel execution

$$Speedup = \frac{execution\_time\_on\_one\_processor}{execution\_time\_on\_N\_processors}$$

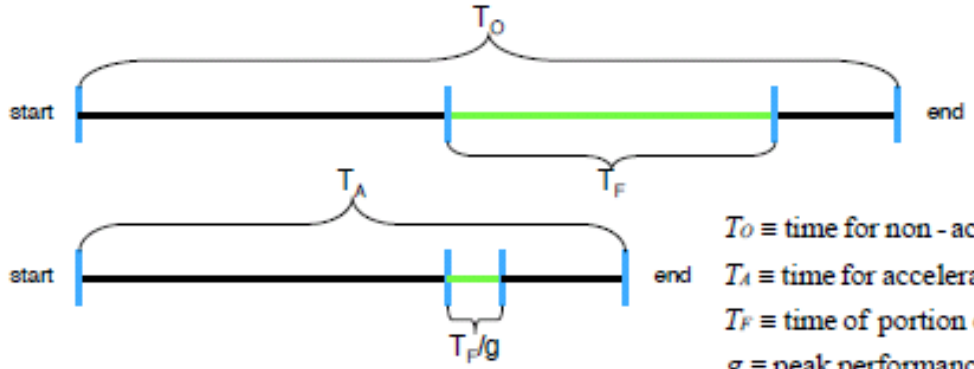
- *Efficiency :* Ratio of the actual performance to the best possible performance.

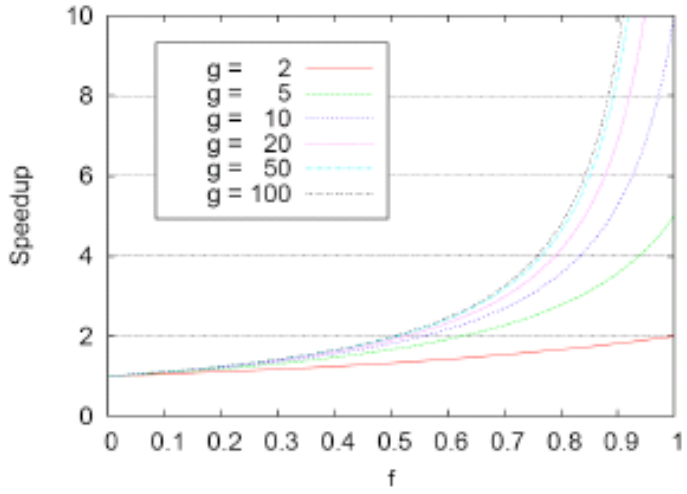
$$Efficiency = \frac{execution\_time\_on\_one\_processor}{(execution\_time\_on\_multiple\_processors \times number\_of\_processors)}$$

# Amdahl's Law: drive or fly?

- Peak performance gain: 10X
  - BMW cruise approx. 60 MPH
  - Boeing 737 cruise approx. 600 MPH
- Time door to door
  - BMW
    - Google estimates 4 hours 30 minutes
  - Boeing 737
    - Time to drive to BTR from my house = 15 minutes
    - Wait time at BTR = 1 hour
    - Taxi time at BTR = 5 minutes
    - Continental estimates BTR to IAH 1 hour
    - Taxi time at IAH = 15 minutes (assuming gate available)
    - Time to get bags at IAH = 25 minutes
    - Time to get rental car = 15 minutes
    - Time to drive to Hyatt Regency from IAH = 45 minutes
    - Total time = 4.0 hours
- Sustained performance gain: 1.125X

# Amdahl's Law




$T_O \equiv$ time for non - accelerated computation

$T_A \equiv$ time for accelerated computation

$T_F \equiv$ time of portion of computation that can be accelerated

$g \equiv$ peak performance gain for accelerated portion of computation

$f \equiv$ fraction of non - accelerated computation to be accelerated

$S \equiv$ speed up of computation with acceleration applied

$S = T_O / T_A$

$f = T_F / T_O$

$$T_A = (1-f) \times T_O + \left(\frac{f}{g}\right) \times T_O$$

$$S = \frac{T_O}{(1-f) \times T_O + \left(\frac{f}{g}\right) \times T_O}$$

$$\boxed{S = \frac{1}{1-f+\left(\frac{f}{g}\right)}}$$

# Parallel Computer Memory Architectures

**Shared Memory**

Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.

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.

Shared memory machines can be divided into two main classes based upon memory access times: ***UMA*** and ***NUMA***.

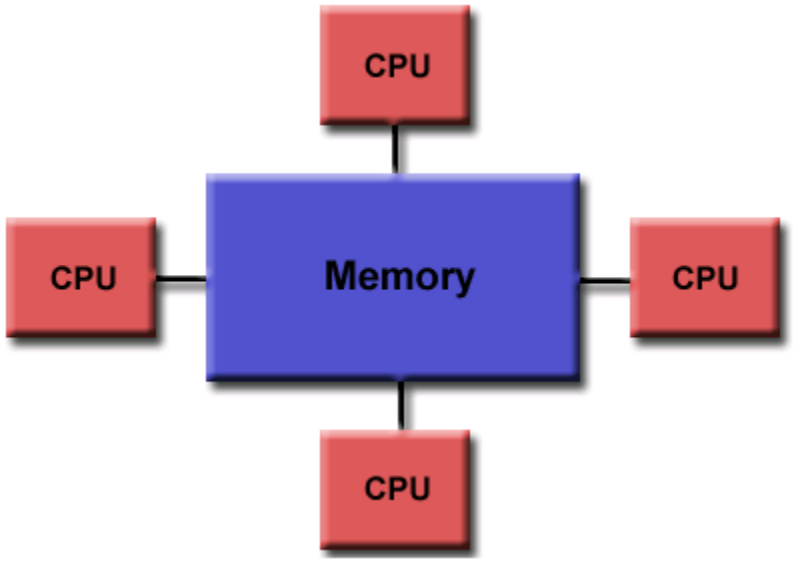
Uniform Memory Access (UMA):

* Most 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.



**Non-Uniform Memory Access (NUMA):**

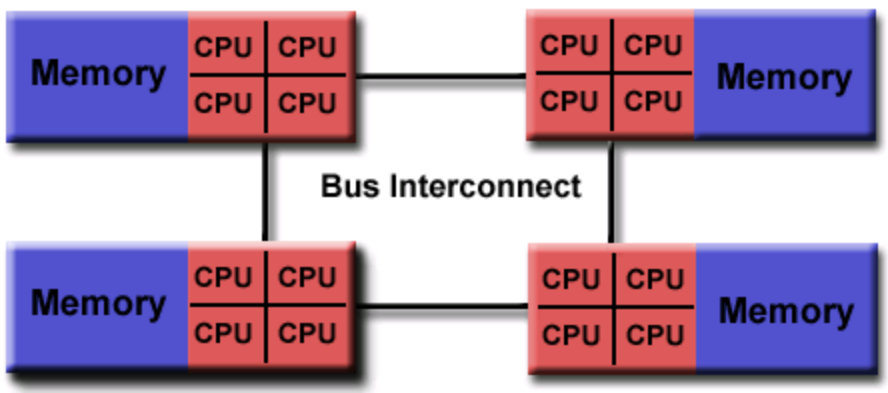
* Often made by physically linking two or more SMPs

* 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



**NUMA vs UMA**

Advantages:

* Global address space provides a user-friendly programming perspective to memory

* Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Disadvantages:

* Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.

* Programmer responsibility for synchronization constructs that insure "correct" access of global memory.

* Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

# Parallel Programming Models

- There are several parallel programming models in common use:
  - Shared Memory
  - Threads
  - Message Passing
  - Data Parallel
  - Hybrid

(We will discuss these Thursday)

# Where Are We and How Did We Get here?

# Definitions: “supercomputer”

*Supercomputer*: A computing system exhibiting high-end performance capabilities and resource capacities within practical constraints of technology, cost, power, and reliability. *Thomas Sterling, 2007*

Supercomputer: a large very fast mainframe used especially for scientific computations. *Merriam-Webster Online*

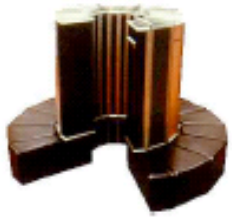
Supercomputer: any of a class of extremely powerful computers. The term is commonly applied to the fastest high-performance systems available at any given time. Such computers are used primarily for scientific and engineering work requiring exceedingly high-speed computations. *Encyclopedia Britannica Online*

# Traditional Supercomputer Technology

- **Single processors designed to be as fast as possible**
  - Cray vector machines for example
- **The good**
  - sequential programming (people understand)
  - 30+ years of compiler and tool development
  - I/O is relatively simple
- **Limitations**
  - Single high performance processors are extremely expensive
  - Significant cooling requirements
  - Single processor performance is reaching its asymptotic limit (remember speed of light)

# A Growth-Factor of a Billion in Performance in a Single Lifetime






1 One OPS

1823 Babbage Difference Engine

1943 Harvard Mark 1

1949 Edsac

1951 Univac 1

$10^3$ KiloOPS

1959 IBM 7094

1964 CDC 6600

$10^6$ MegaOPS

1976 Cray 1

1982 Cray XMP

$10^9$ GigaOPS

1988 Cray YMP

1991 Intel Delta

1996 T3E

1997 ASCI Red

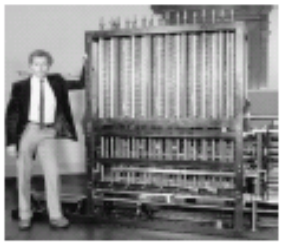
$10^{12}$ TeraOPS

2001 Earth Simulator

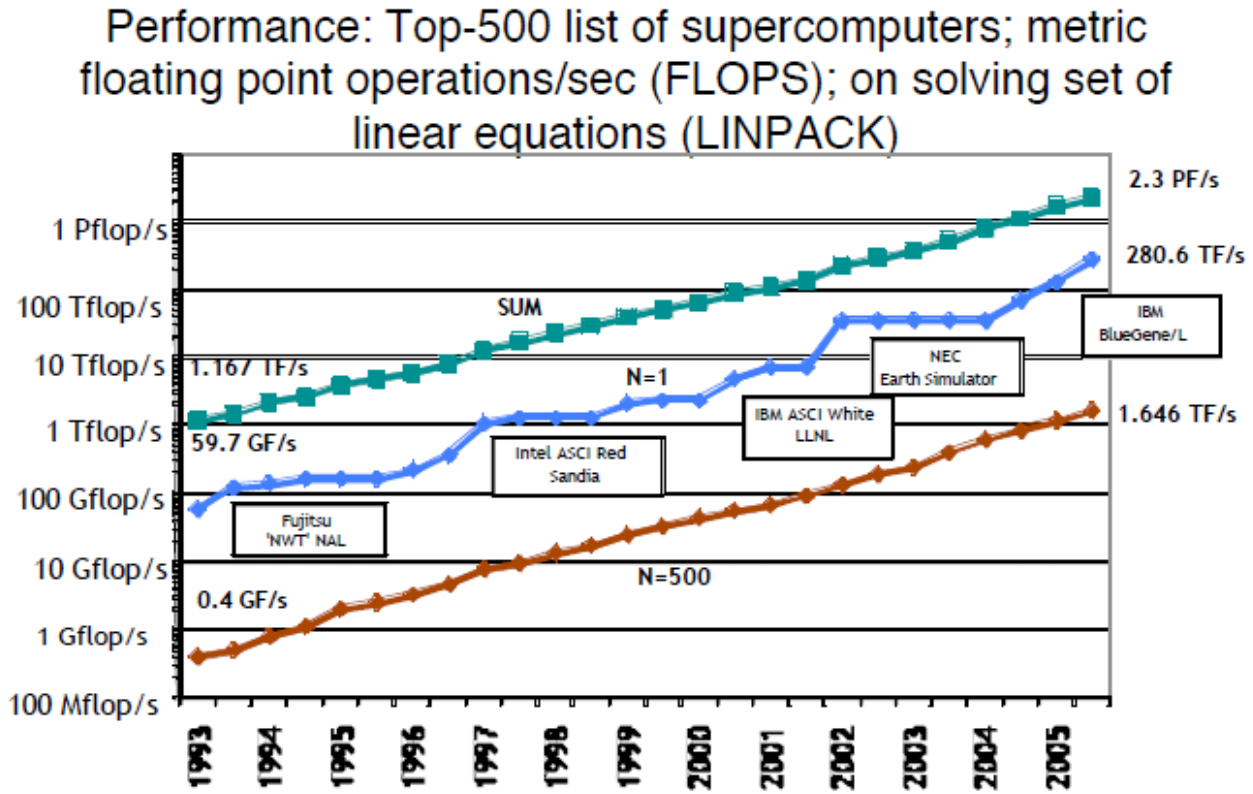
2003 Cray X1

$10^{15}$ PetaOPS

**2009 Roadrunner 1.456 PetaFlops**








**Update: Nov 2008 Top 500 http://www.top500.org/**

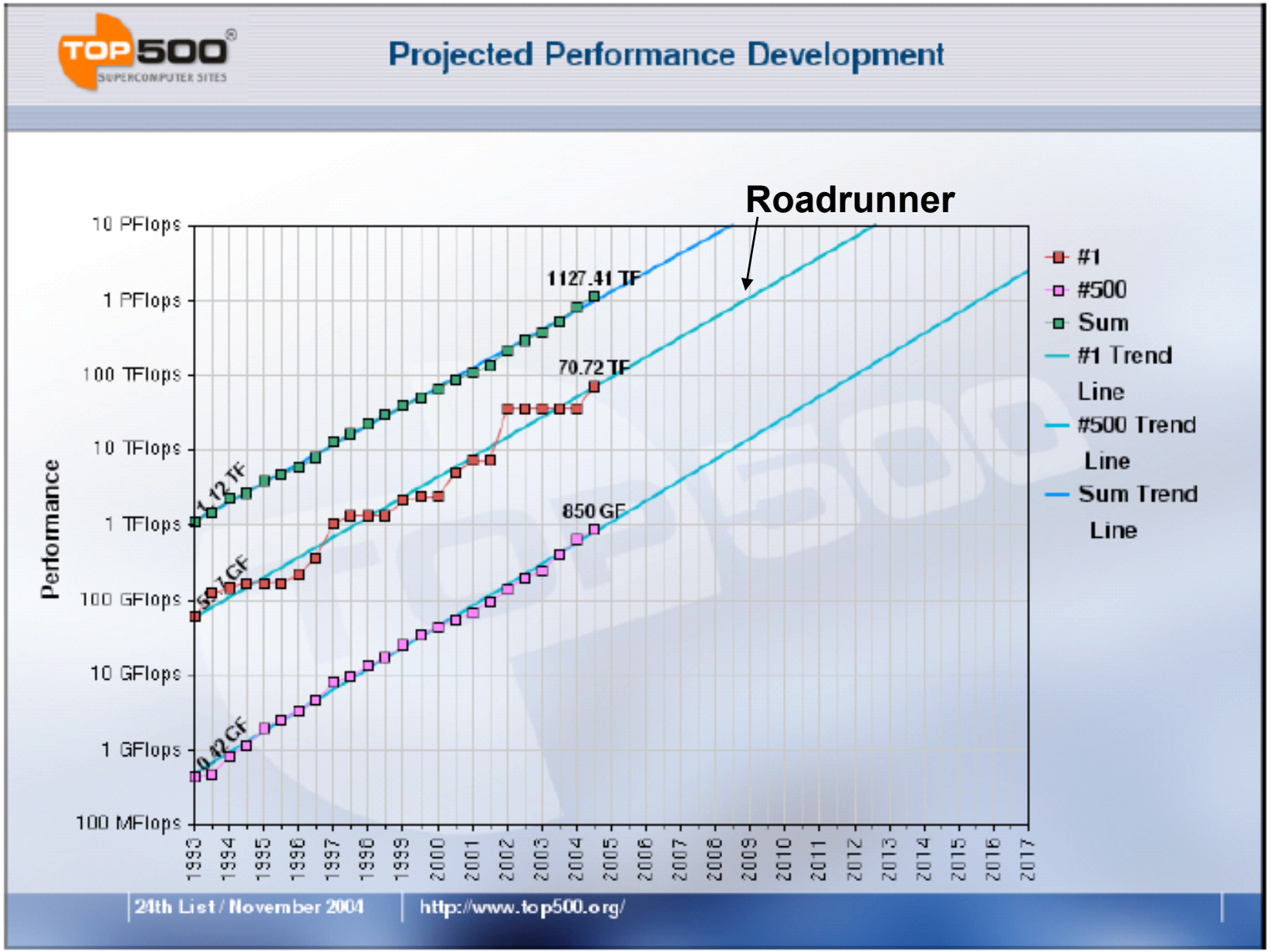


**#1 = Roadrunner Los Alamos IBM bladecenter 1.456 petaFLOPS 129,600 cores**

**#2 = Jaguar Oak Ridge Cray XT5 1.38 petaFLOPS 150152 cores**

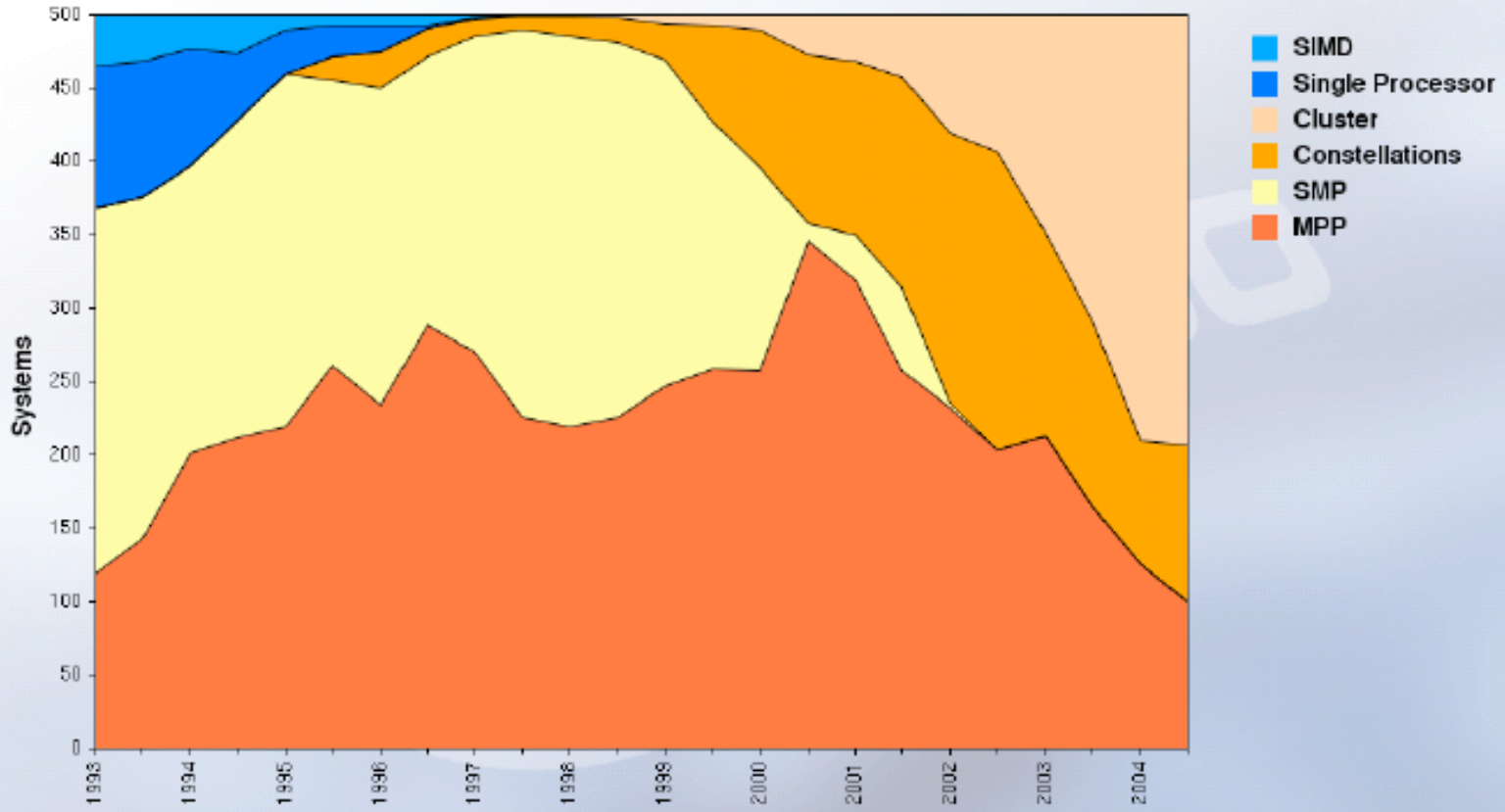
**#3 = Pleiades NASA Ames SGI Altix 0.6 petaFLOPS 51,200 cores**

**#4 = BlueGene/L**


TOP500
SUPERCOMPUTER SITES
Projected Performance Development
Roadrunner
Performance
10 PFlops
1 PFlops
100 TFlops
10 TFlops
1 TFlops
100 GFlops
10 GFlops
1 GFlops
100 MFlops
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
1127.41 TF
70.72 TF
850 GF
#1
#500
Sum
#1 Trend Line
#500 Trend Line
Sum Trend Line
24th List / November 2004
http://www.top500.org/

TOP500 SUPERCOMPUTER SITES

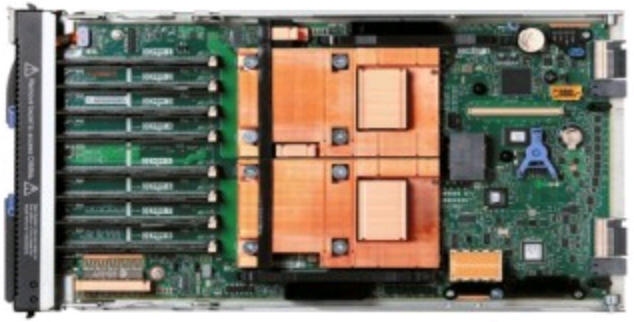
# Architectures / Systems



# Roadrunner Details

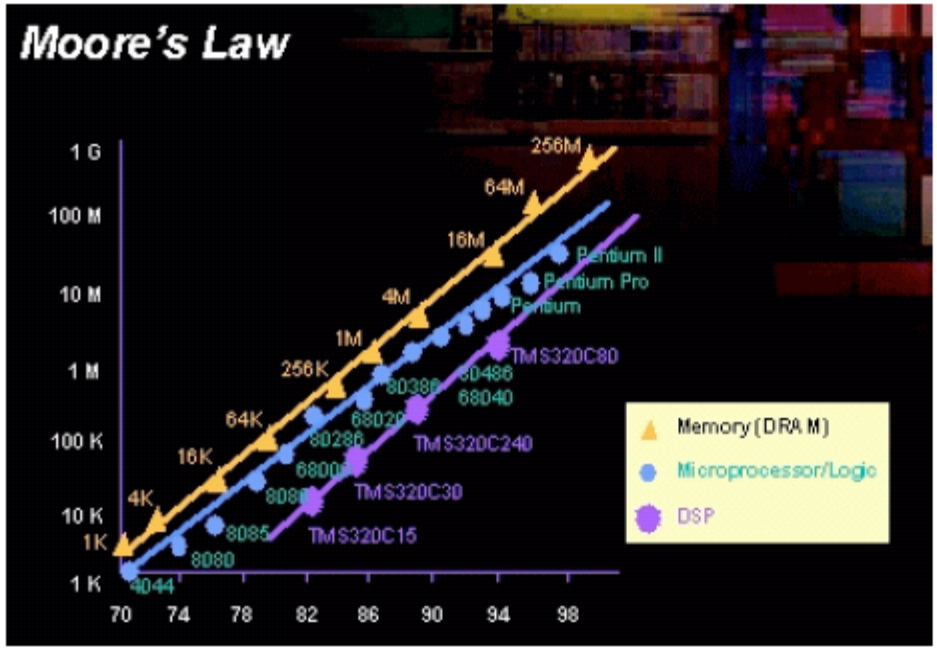
- Roadrunner, named after the New Mexico state bird, **cost about $100 million**, the world's first "hybrid" supercomputer – one powerful enough to operate at one petaflop twice as fast as the then No.1 rated IBM Blue Gene system at Lawrence Livermore National Lab
- Roadrunner will primarily be used on nuclear weapons stockpile applications. It will also be used for research into astronomy, energy, human genome science and climate change.
- Roadrunner is the world's first **hybrid supercomputer**. Cell Broadband Engine® -- originally designed for video game platforms such as the Sony Playstation 3® -- will work in conjunction with x86 processors from AMD®.
- Roadrunner connects 6,562 dual-core AMD Opteron® chips as well as 12,240 Cell chips (on IBM Model QS22 blade servers).
  - 98 terabytes of memory, and is housed in 278 refrigerator-sized, IBM BladeCenter® racks occupying 5,200 square feet. Its 10,000 connections – both Infiniband and Gigabit Ethernet -- require 55 miles of fiber optic cable. Roadrunner weighs 500,000 lbs.
  - delivers world-leading efficiency – 437 million calculations per watt.

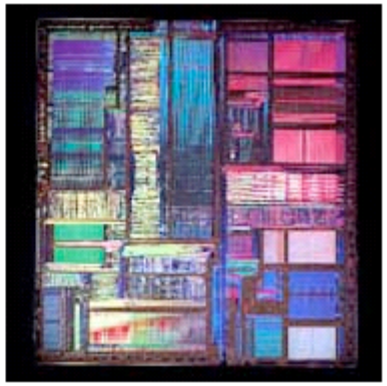
# IBM bladecenter QS22



# IBM Blue Gene/L

# *Moore's Law*

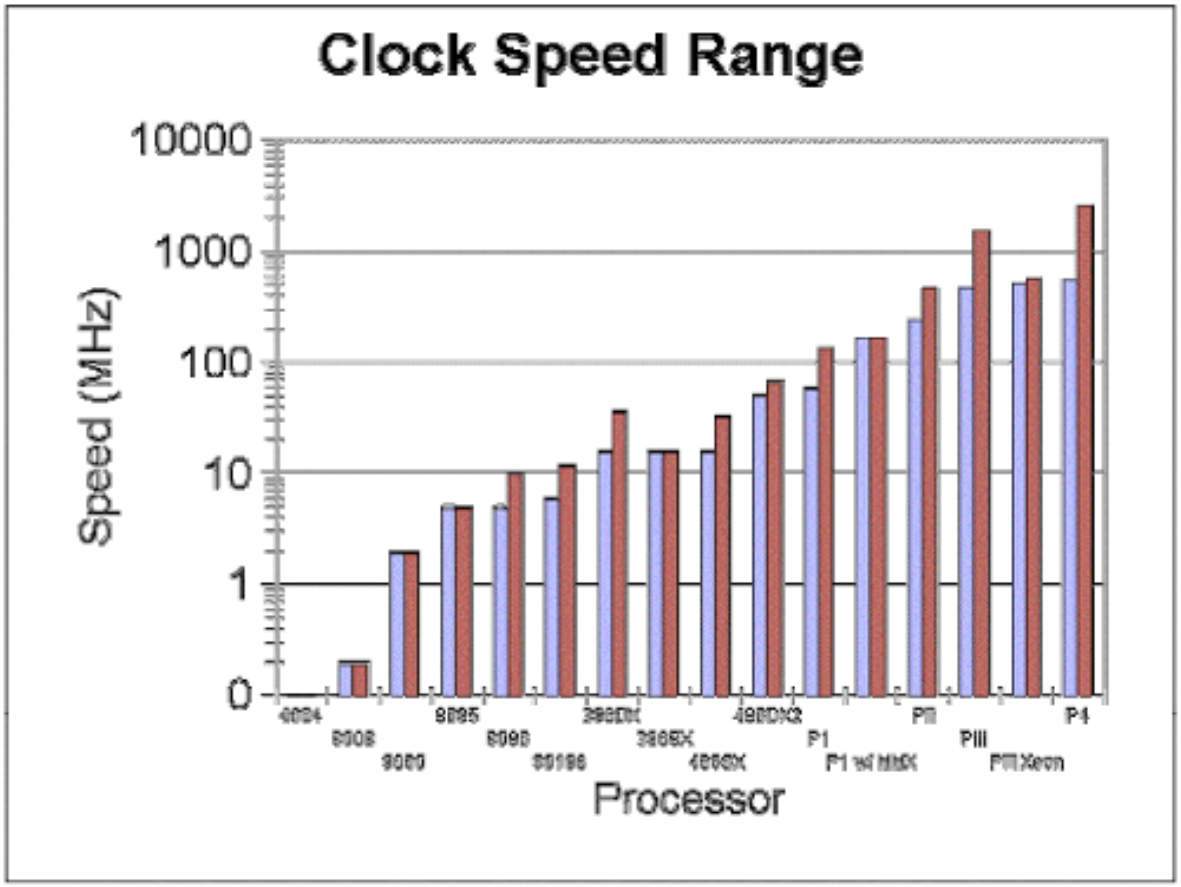


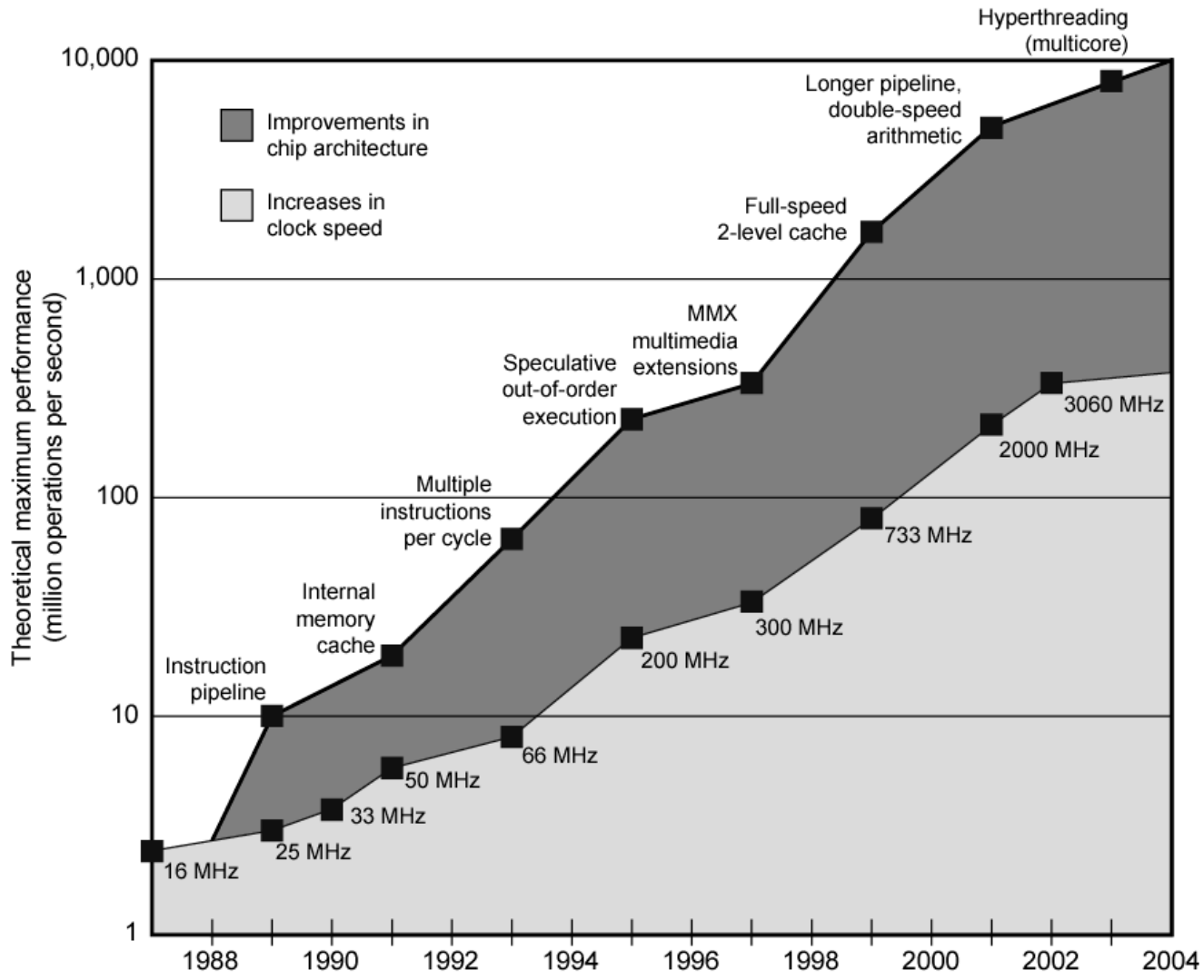


**Panel 6 Years Ago "Is Moore's Law Dead?"**

# Microprocessor Clock Speed



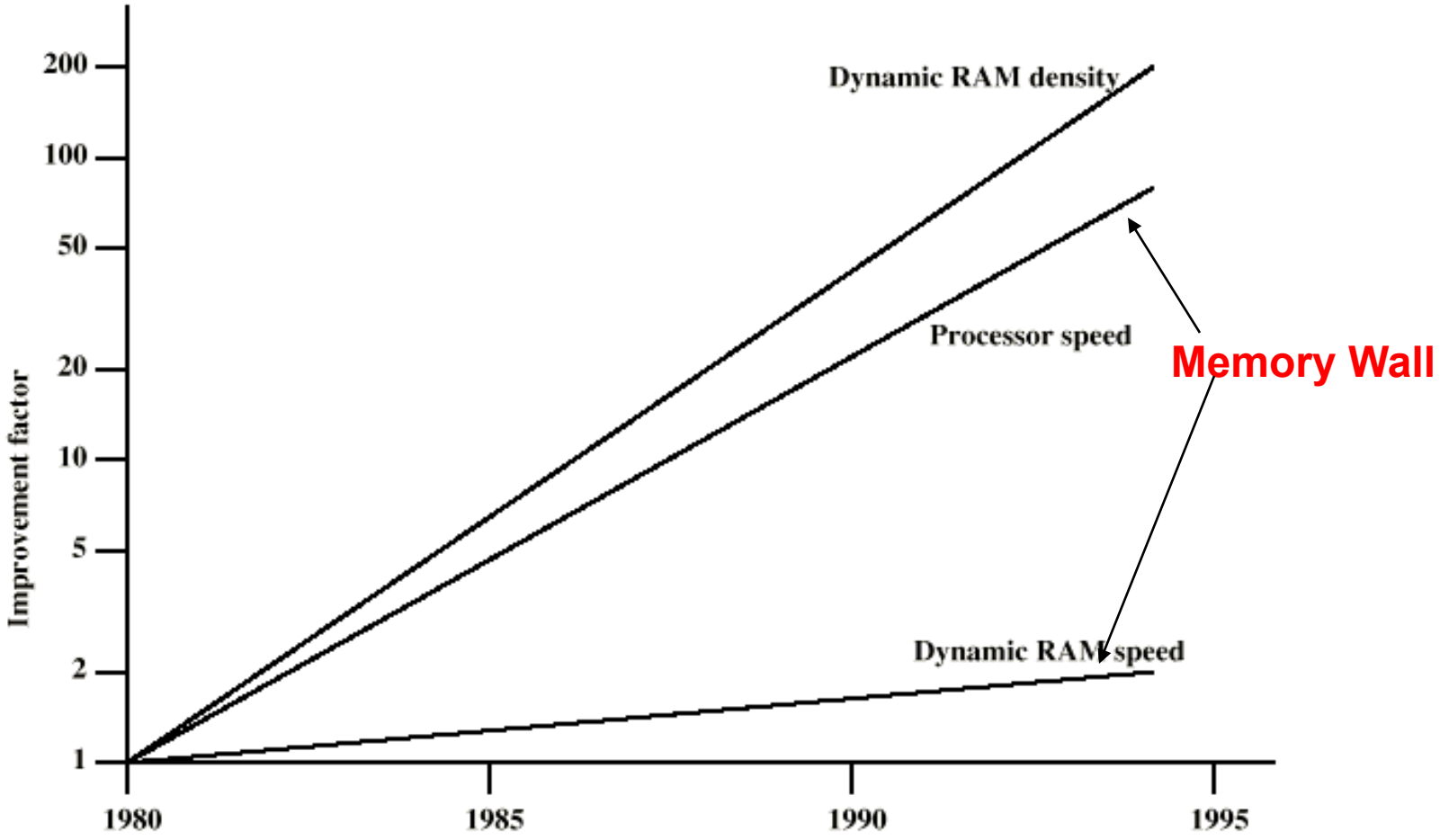
# Intel Microprocessor Performance



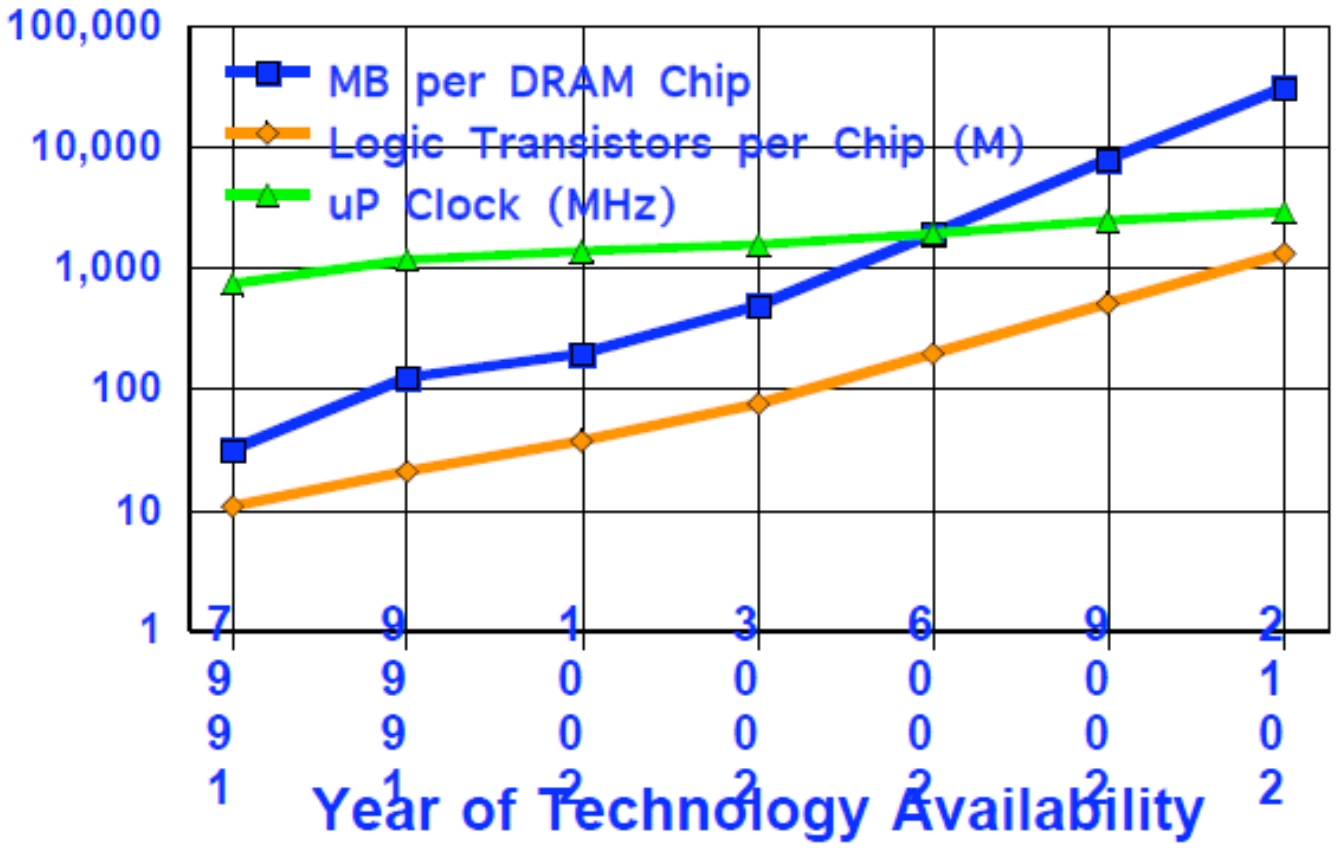
# Why Fast Machines Run Slow

- *Latency*
  - Waiting for access to memory or other parts of the system
- *Overhead*
  - Extra work that has to be done to manage program concurrency and parallel resources the real work you want to perform
- *Starvation*
  - Not enough work to do due to insufficient parallelism or poor load balancing among distributed resources
- *Contention*
  - Delays due to fighting over what task gets to use a shared resource next. Network bandwidth is a major constraint.

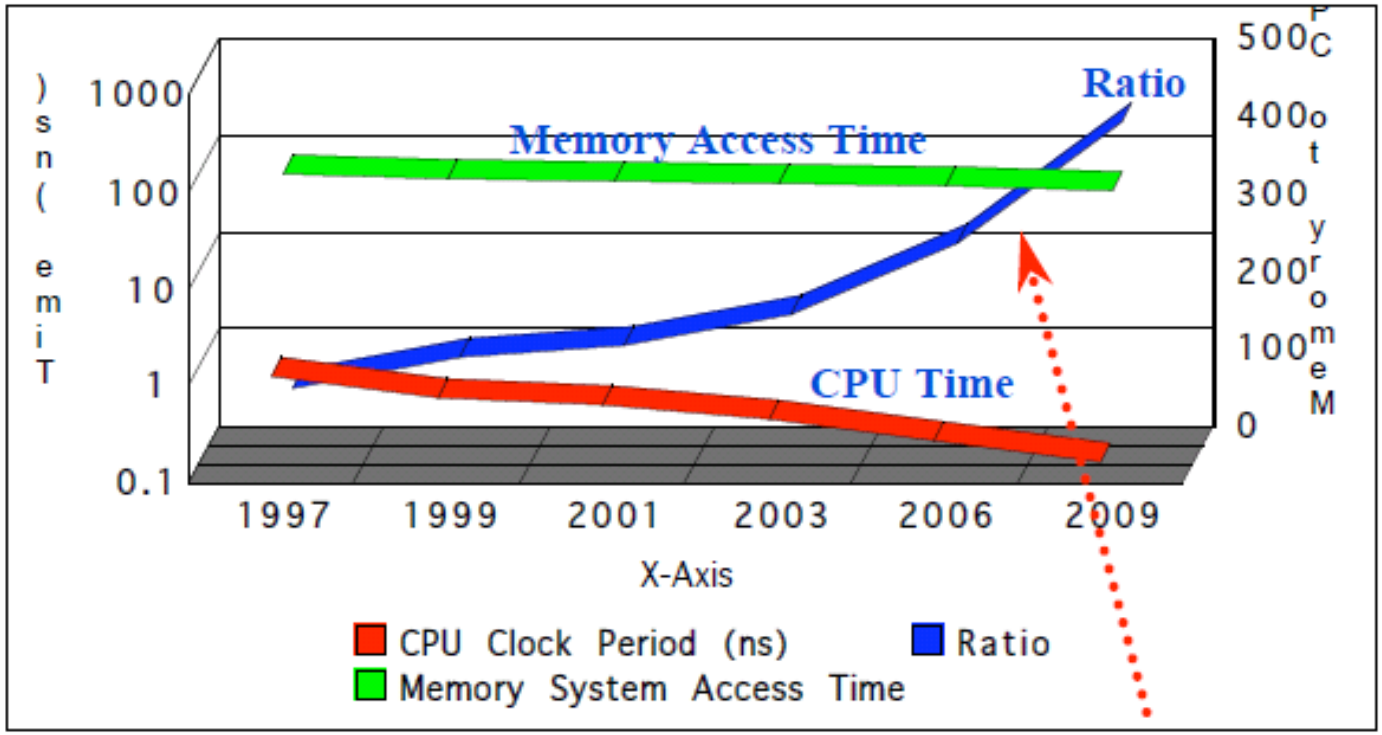
# Memory Speed Has Not Kept Up



# The *SIA ITRS* Roadmap



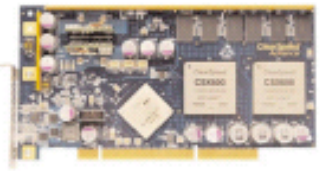
# Latency in a Single System

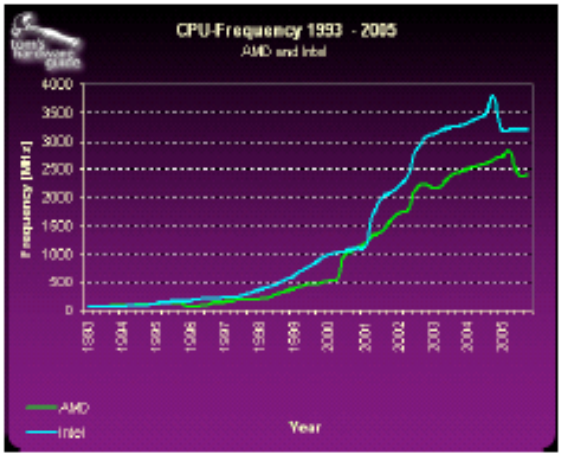


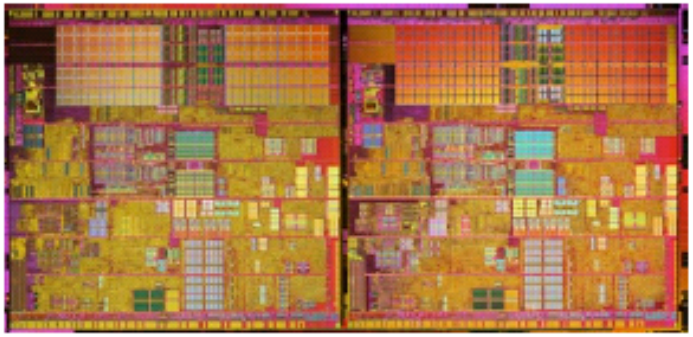

THE *WALL*

# Driving Issues/Trends

- *Multicore*
  - Now: 2
  - possibly 100's
  - will be million-way parallelism
- *Heterogeneity*
  - GPU
  - Clearspeed
  - Cell SPE
- Component I/O Pins
  - Off chip bandwidth not increasing with demand
    - Limited number of pins
    - Limited bandwidth per pin (pair)
  - Cache size per core may decline
  - Shared cache fragmentation
- System Interconnect
  - Node bandwidth not increasing proportionally to core demand
- Power
  - Mwatts at the high end = millions of $s per year

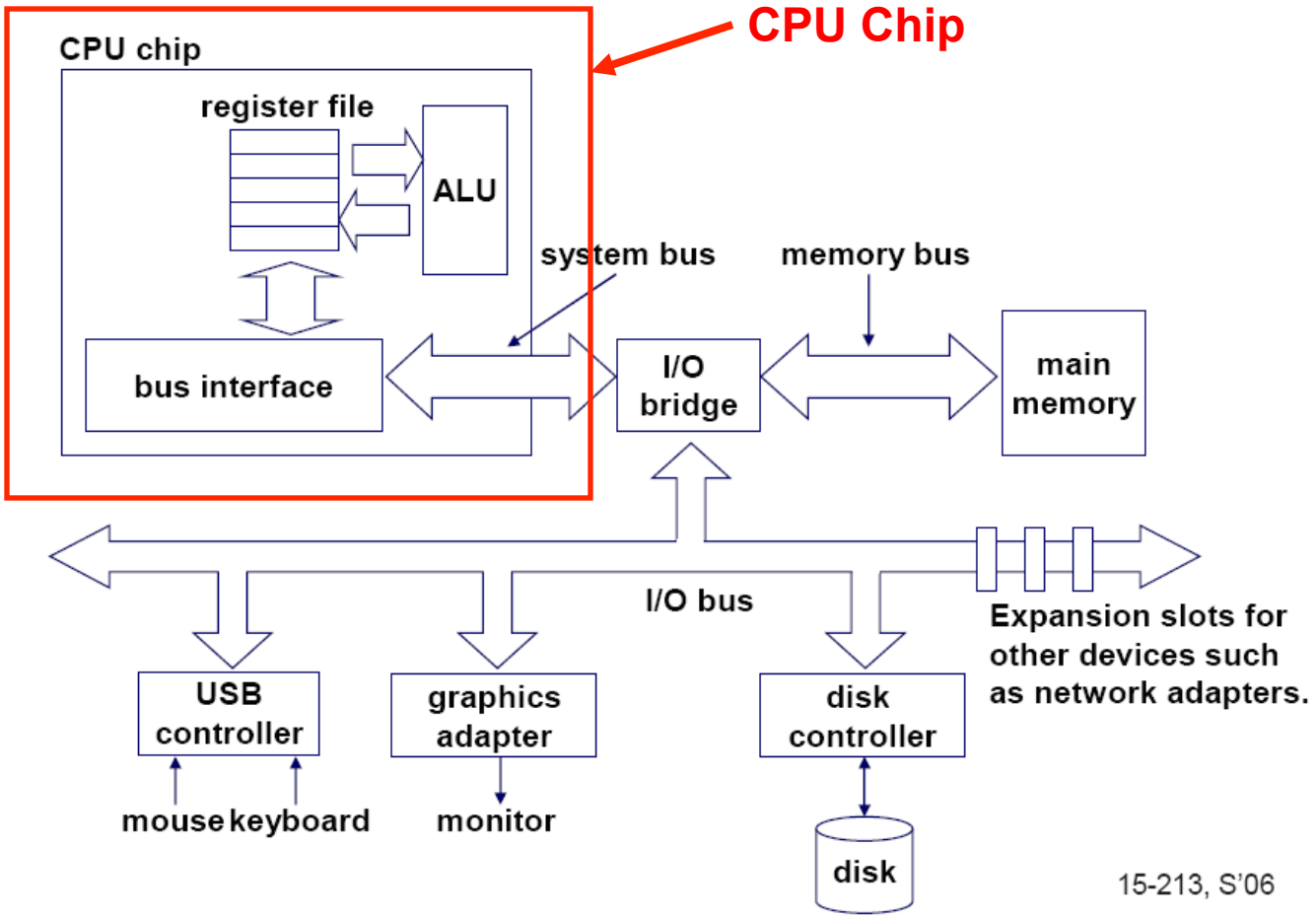




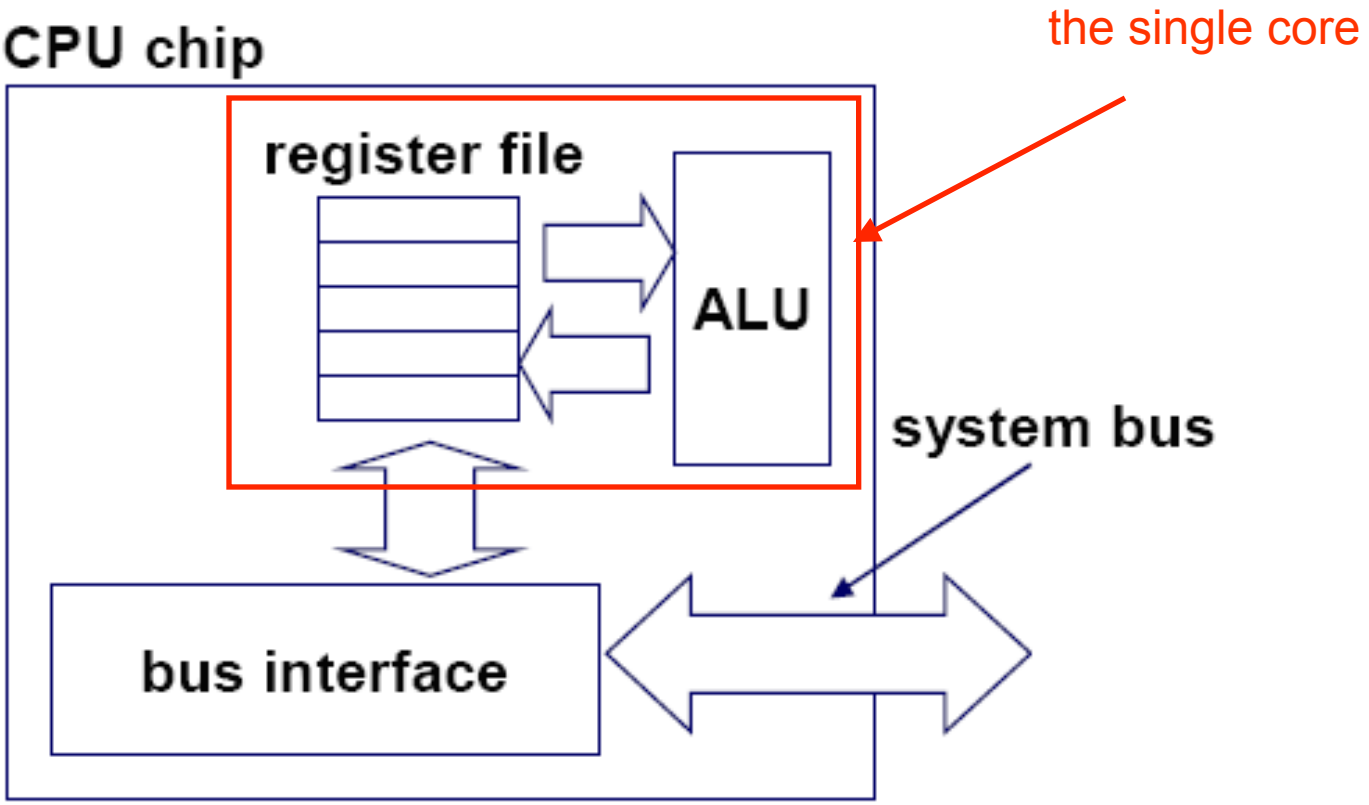



# Multi-Core

# Single-core computer

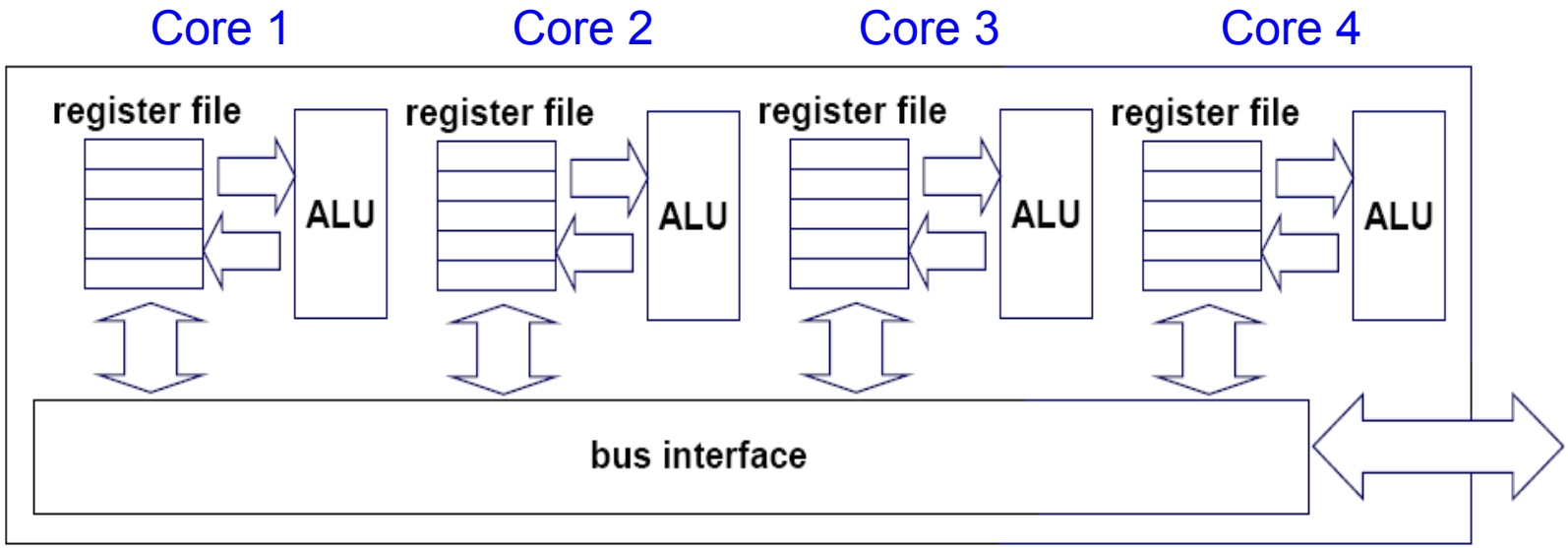


# Single-core CPU chip



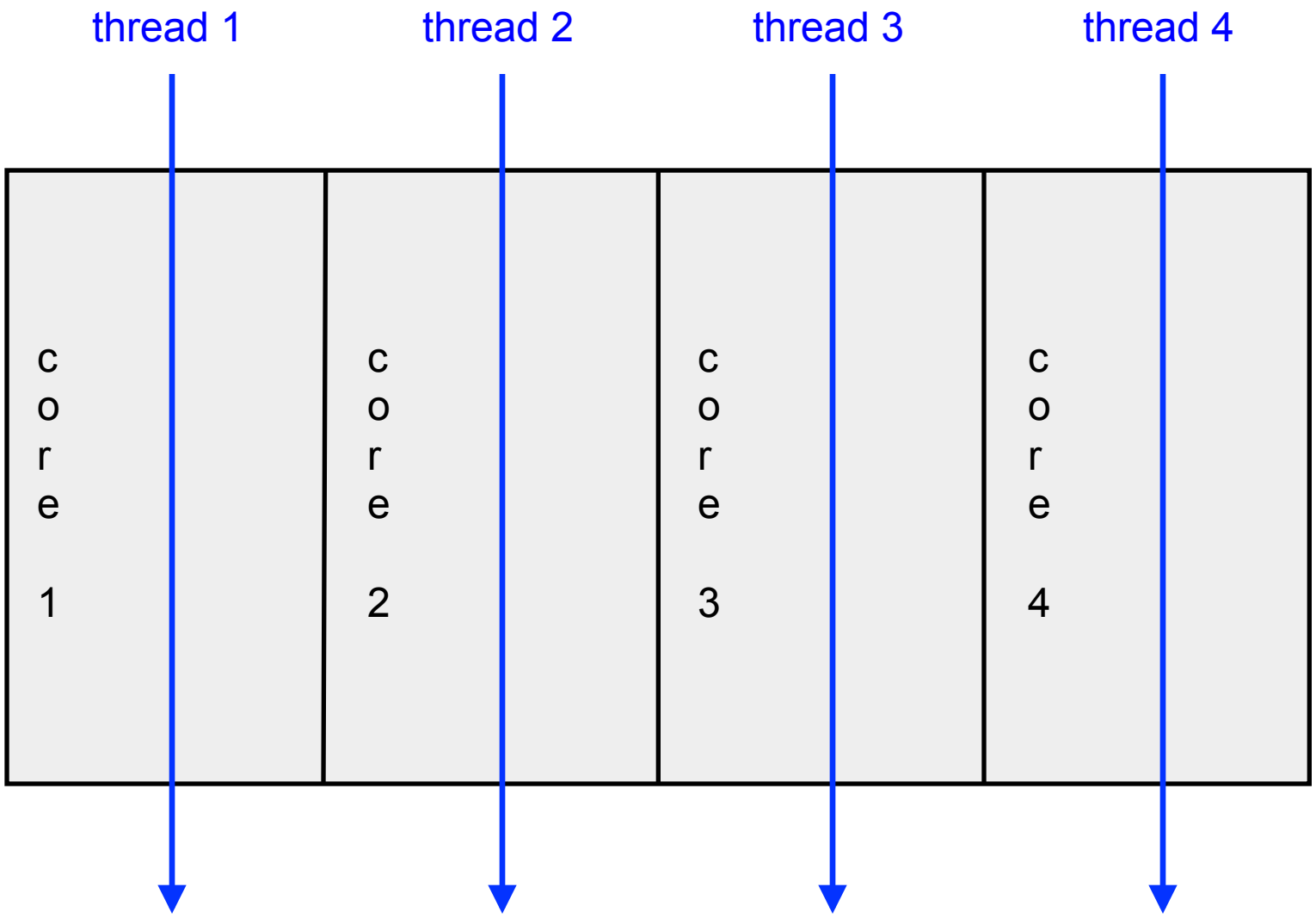
# Multi-core architectures

Replicate multiple processor cores on a single die.

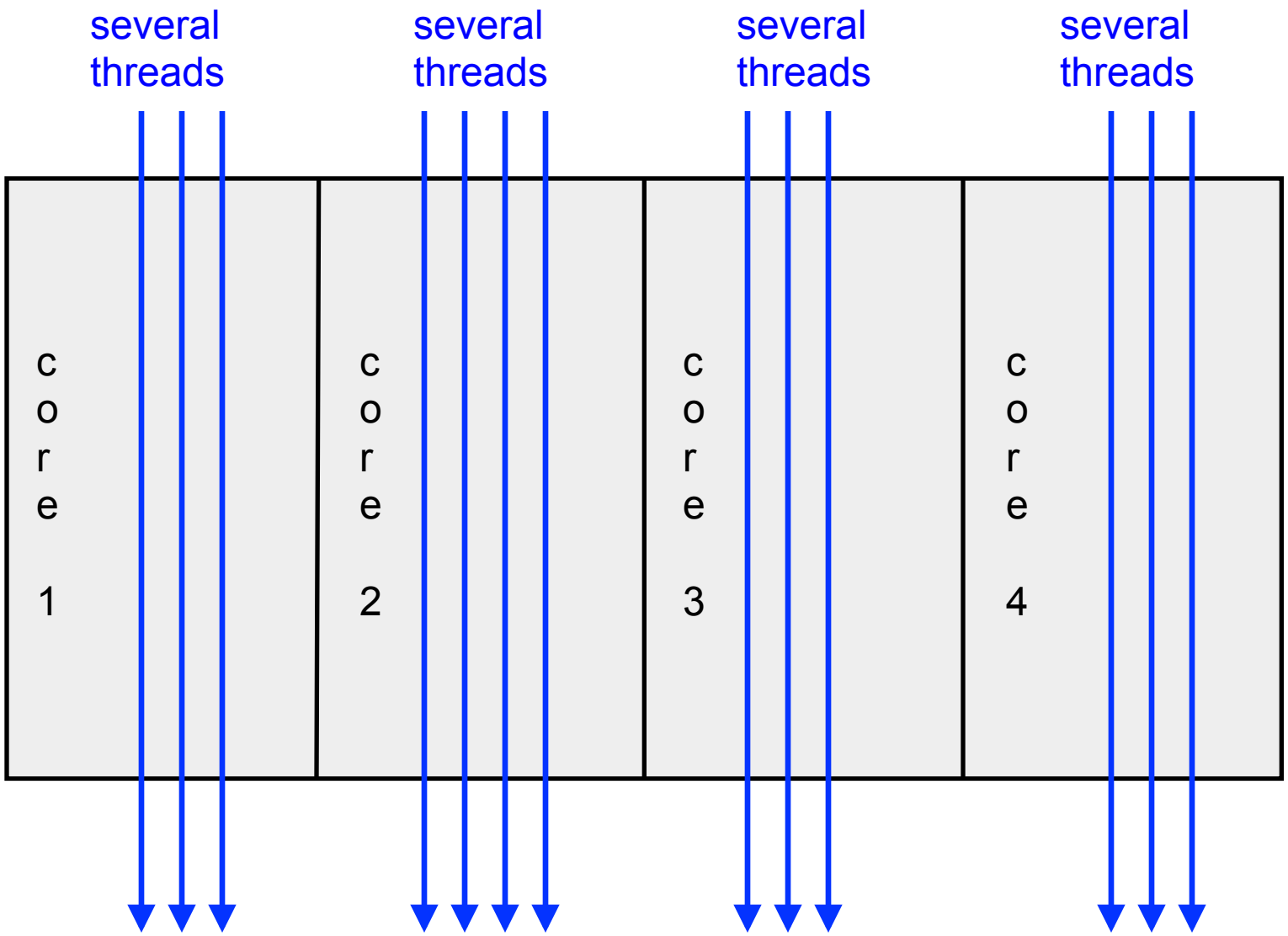



Multi-core CPU chip

# The cores run in parallel



# Within each core, threads are time-sliced (just like on a uniprocessor)



# Interaction with OS

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today

# Instruction-level parallelism

- Parallelism at the machine-instruction level
- The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years

# Thread-level parallelism (TLP)

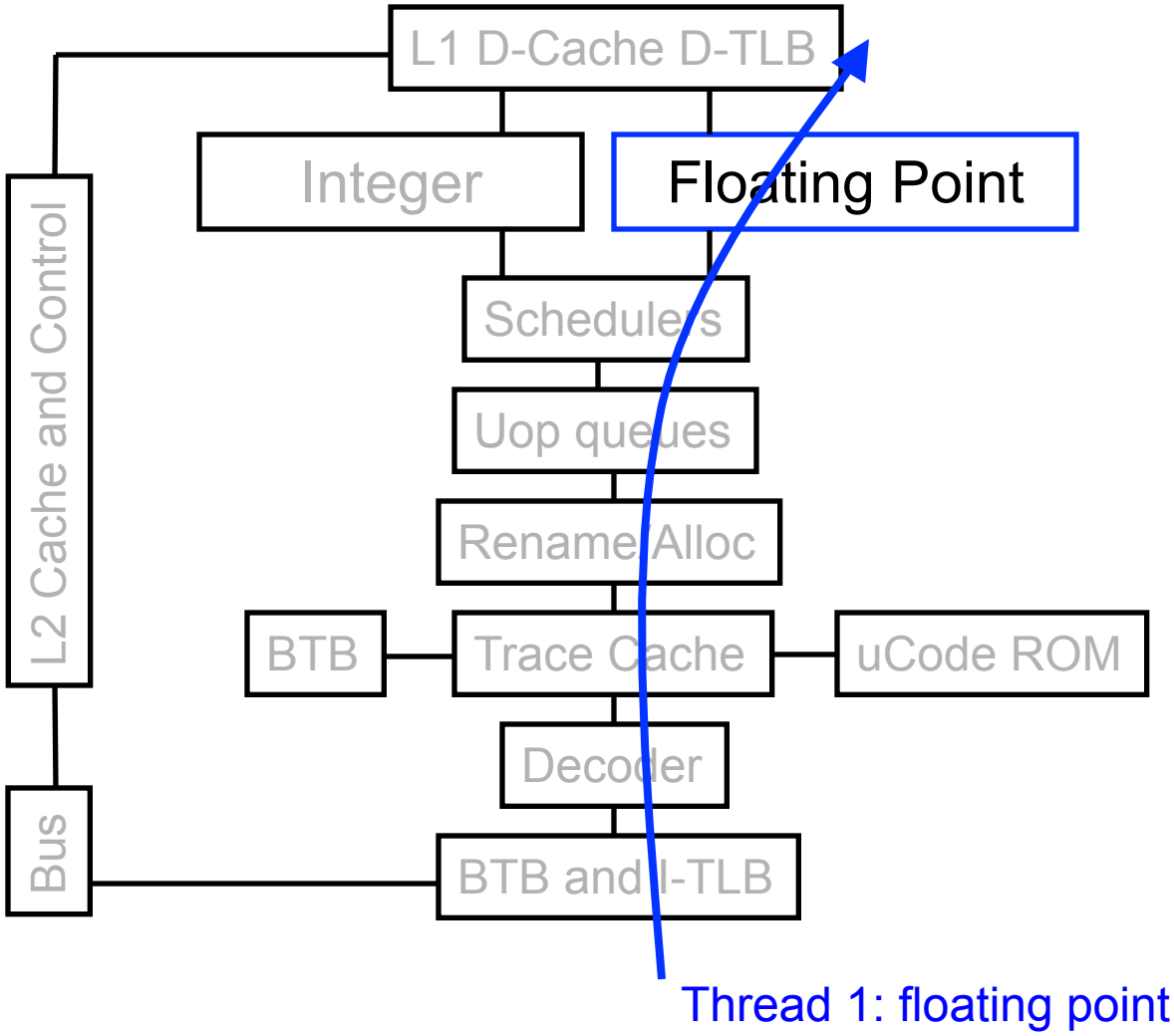
- This is parallelism on a more coarser scale
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and physics in three separate threads
- Single-core superscalar processors cannot fully exploit TLP
- Multi-core architectures are the next step in processor evolution: explicitly exploiting TLP

# Simultaneous multithreading (SMT)

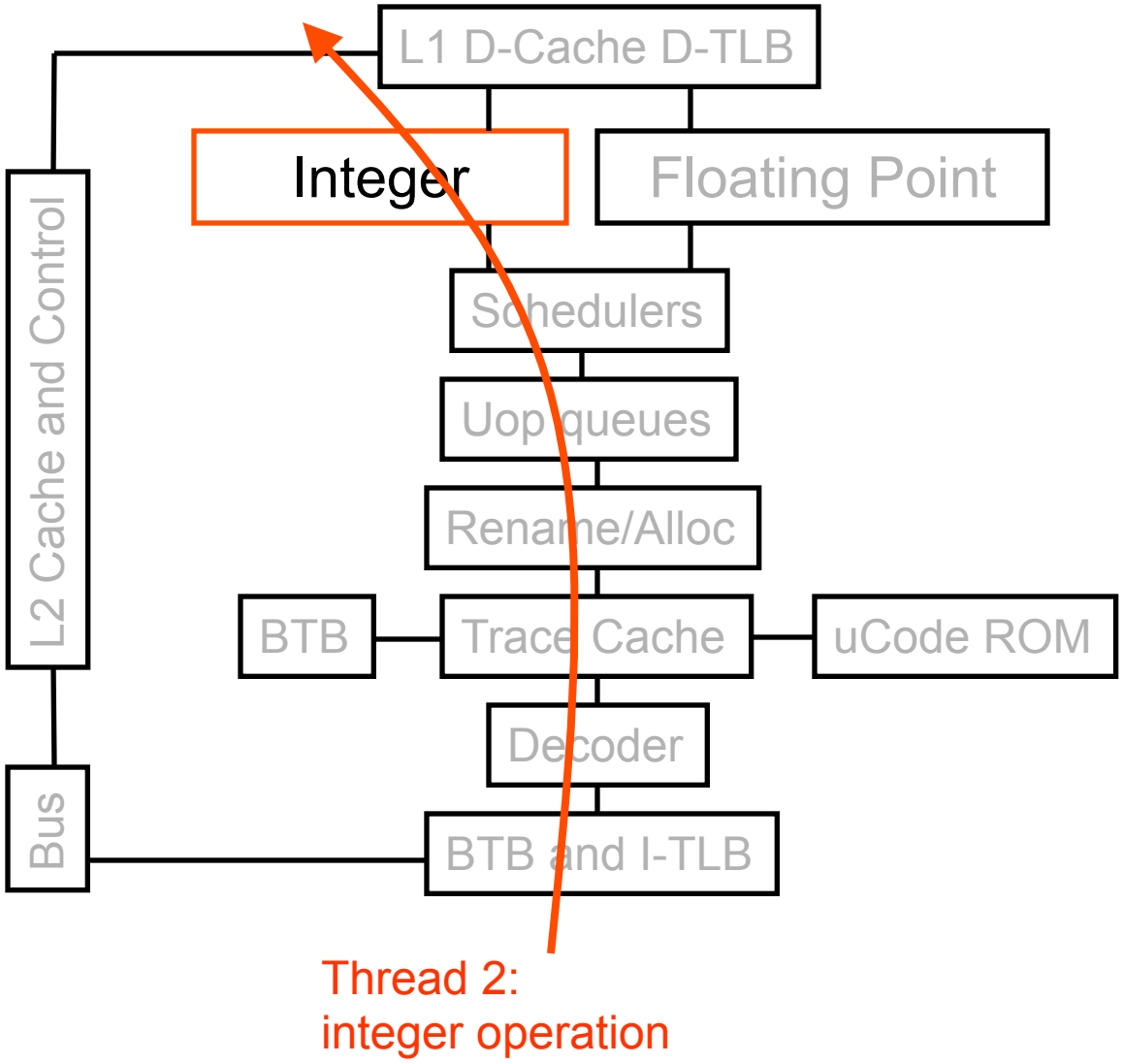
- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple “threads” on the same core

- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

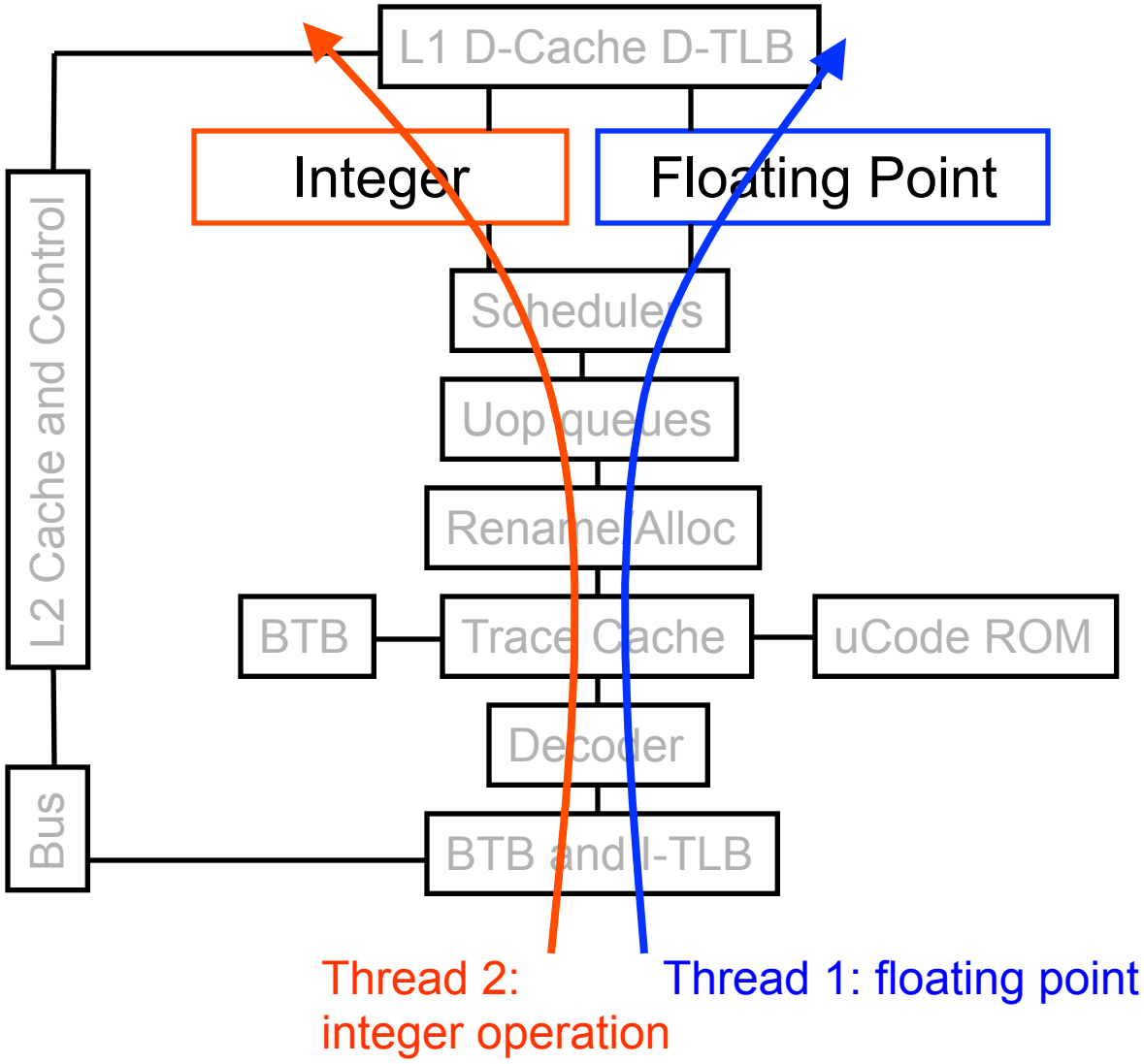
# Without SMT, only a single thread can run at any given time



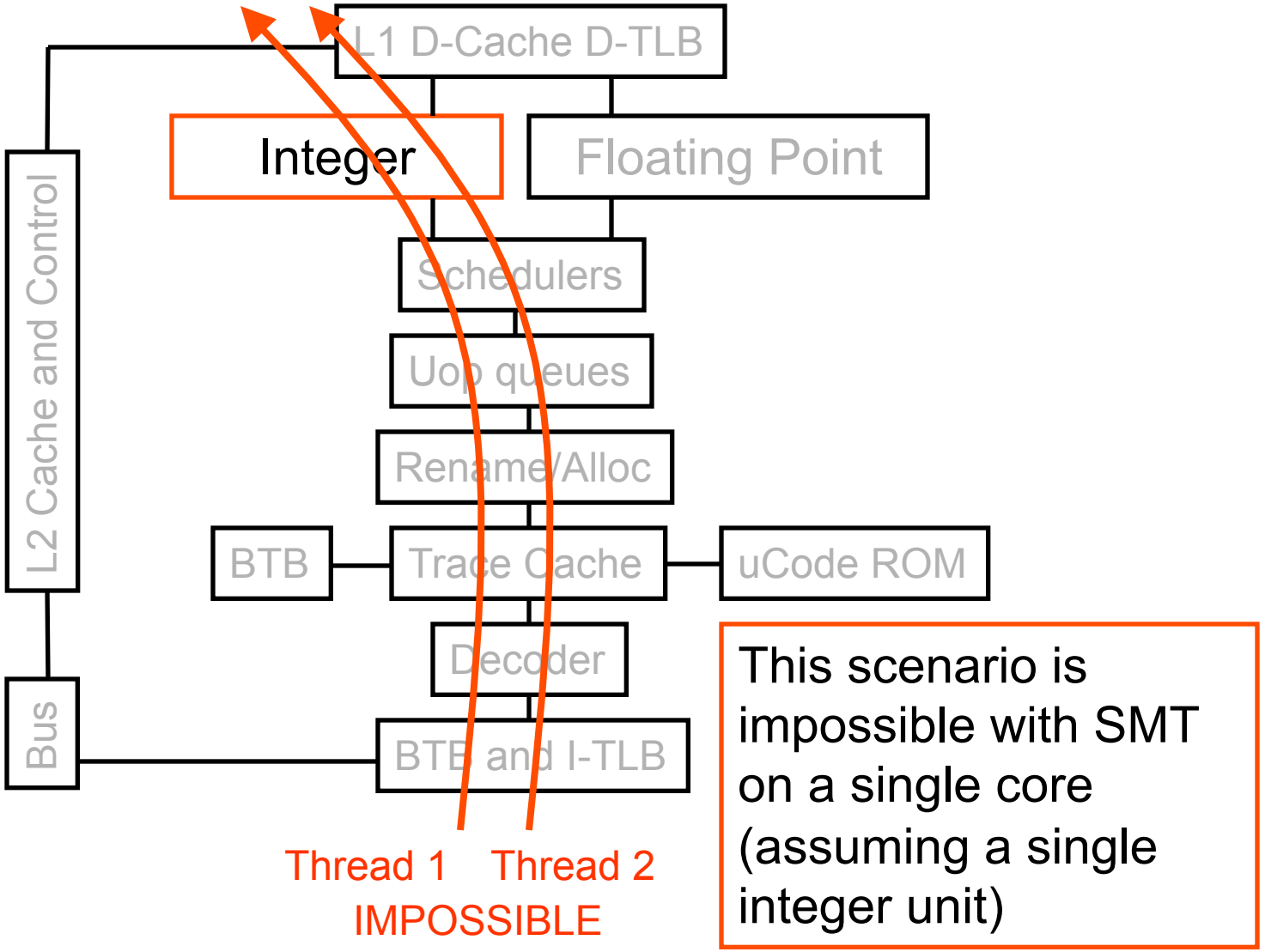
# Without SMT, only a single thread can run at any given time



# SMT processor: both threads can run concurrently



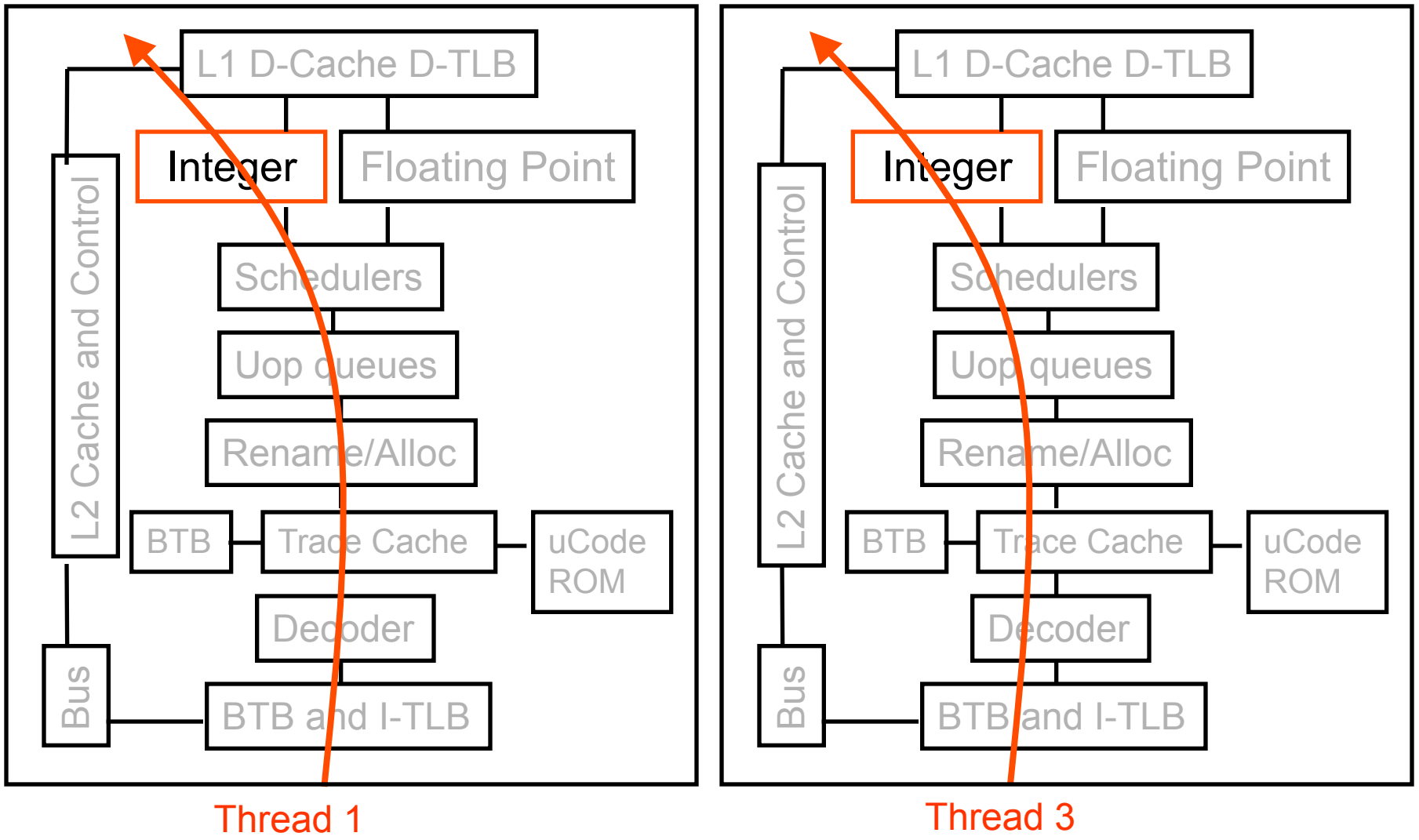
# But: Can't simultaneously use the same functional unit



# SMT not a “true” parallel processor

- Enables better threading (e.g. up to 30%)
- OS and applications perceive each simultaneous thread as a separate “virtual processor”
- The chip has only a single copy of each resource
- Compare to multi-core: each core has its own copy of resources

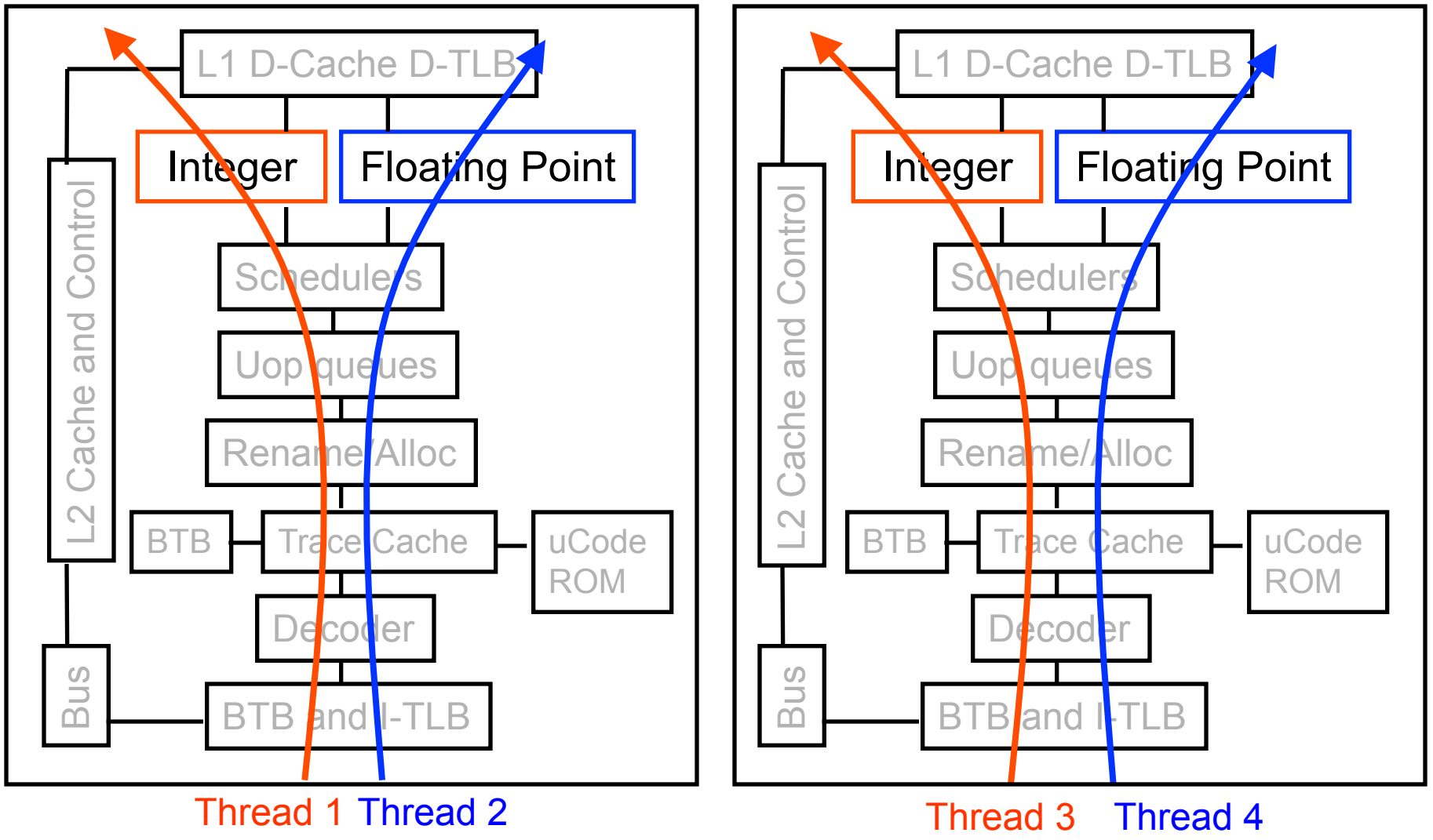
# Multi-core:
# threads can run on separate cores



# Combining Multi-core and SMT

- Cores can be SMT-enabled (or not)
- The different combinations:
  - Single-core, non-SMT: standard uniprocessor
  - Single-core, with SMT
  - Multi-core, non-SMT
  - Multi-core, with SMT
  - The number of SMT threads:
    2, 4, or sometimes 8 simultaneous threads
- Intel calls them “hyper-threads”

# SMT Dual-core: all four threads can run concurrently

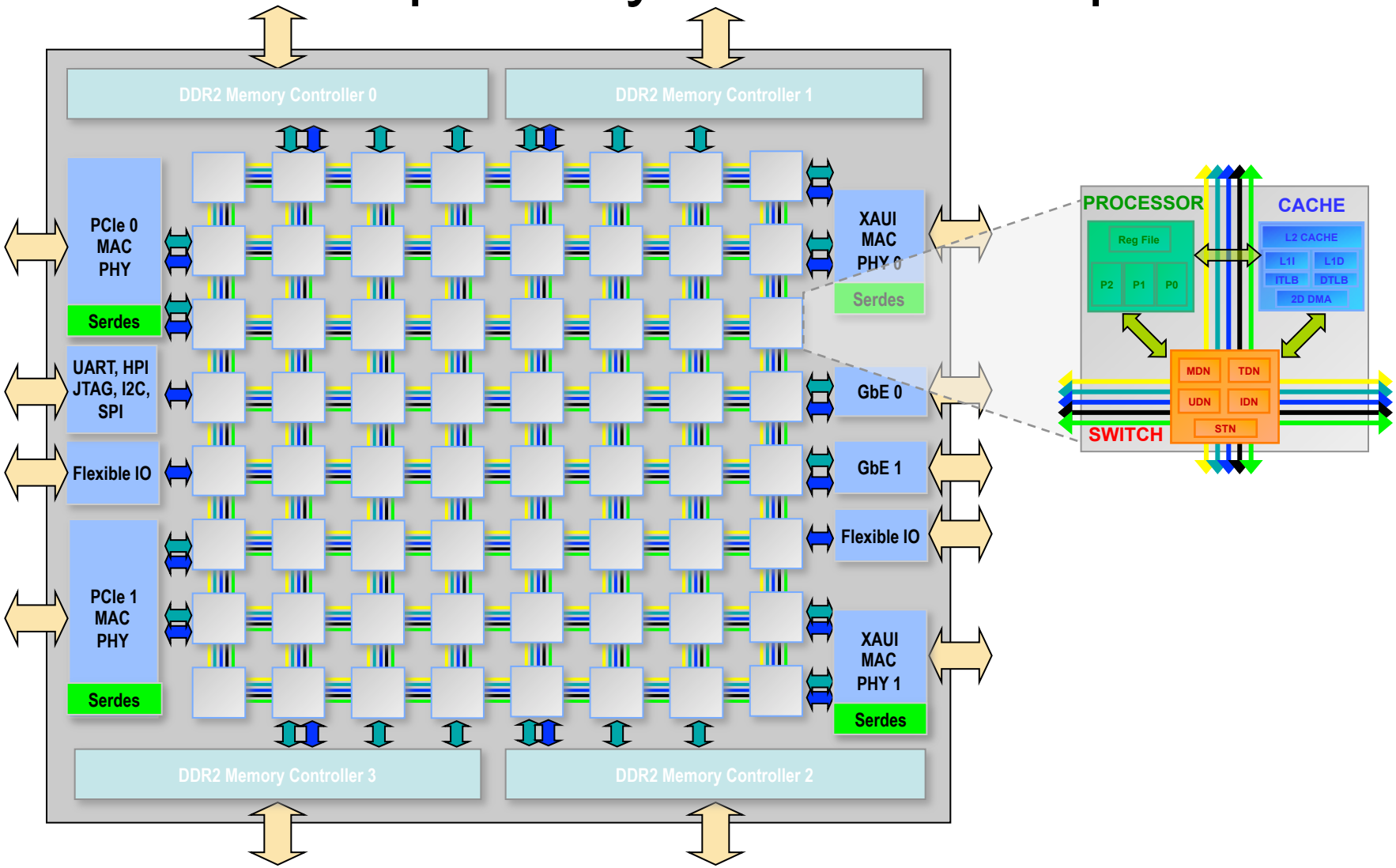


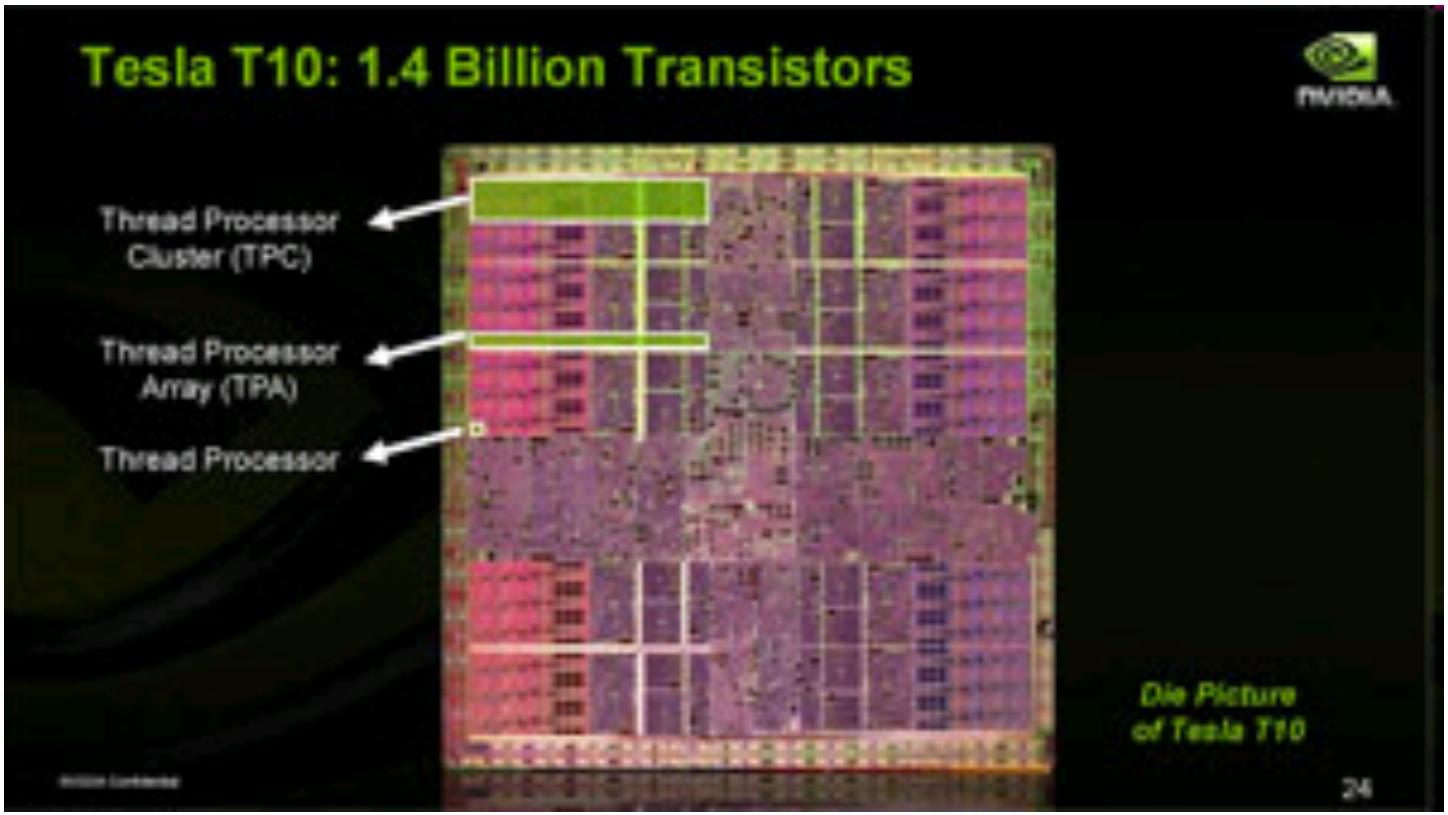
# Comparison: multi-core vs SMT

- Multi-core:
  - Since there are several cores, each is smaller and not as powerful (but also easier to design and manufacture)
  - However, great with thread-level parallelism
- SMT
  - Can have one large and fast superscalar core
  - Great performance on a single thread
  - Mostly still only exploits instruction-level parallelism

# TILE64 Processor Block Diagram

## A Complete System on a Chip



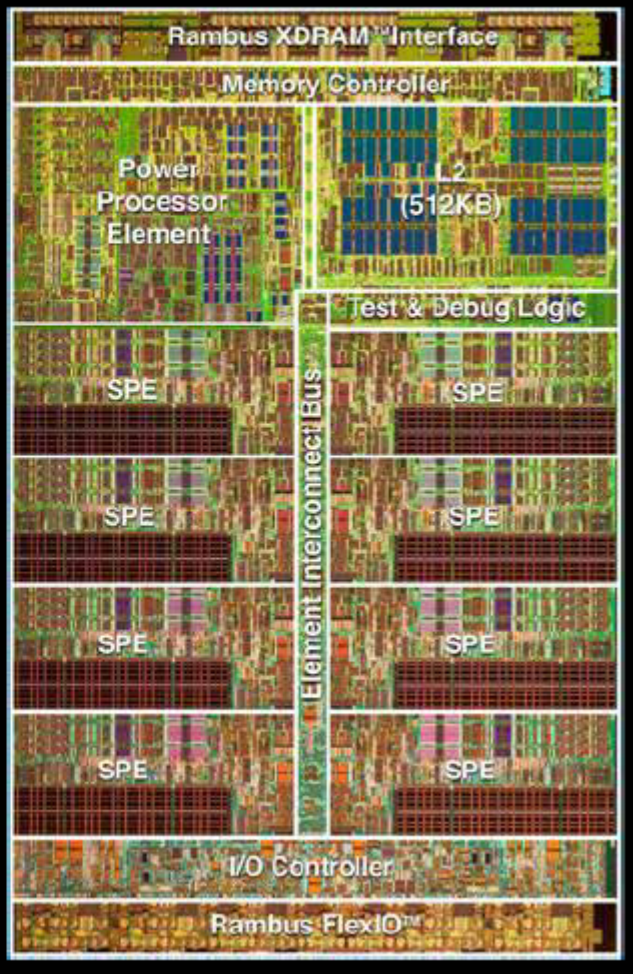


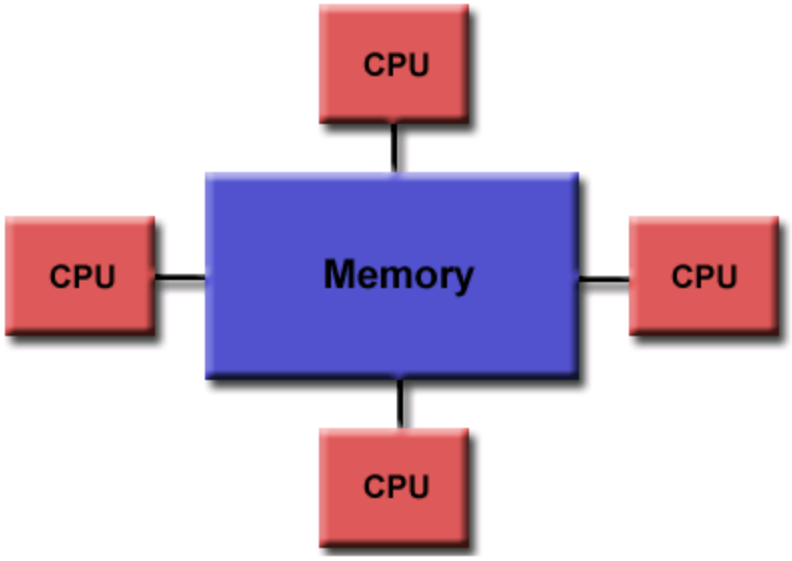

**240 processing cores**
**Tesla Board peaks at ½ TeraFLOP**

# IBM Cell Chip

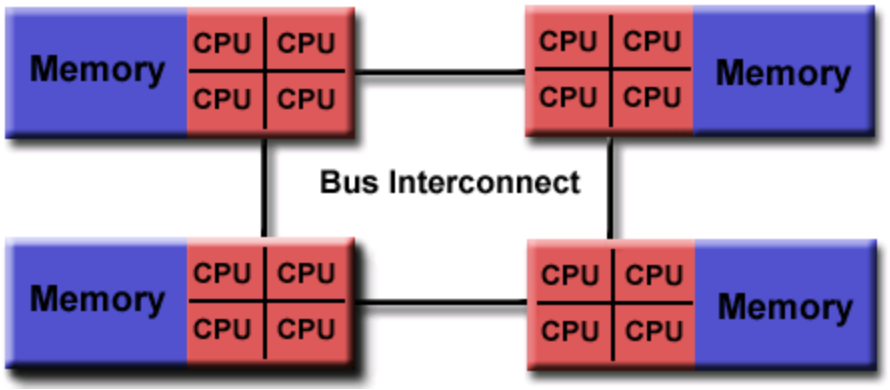
## Highlights (3.2 GHz)

- 241M transistors
- 235mm2
- 9 cores, 10 threads
- >200 GFlops (SP)
- >20 GFlops (DP)
- Up to 25 GB/s memory B/W
- Up to 75 GB/s I/O B/W
- >300 GB/s EIB
- Top frequency >4GHz
  (observed in lab)






Shared Memory: UMA



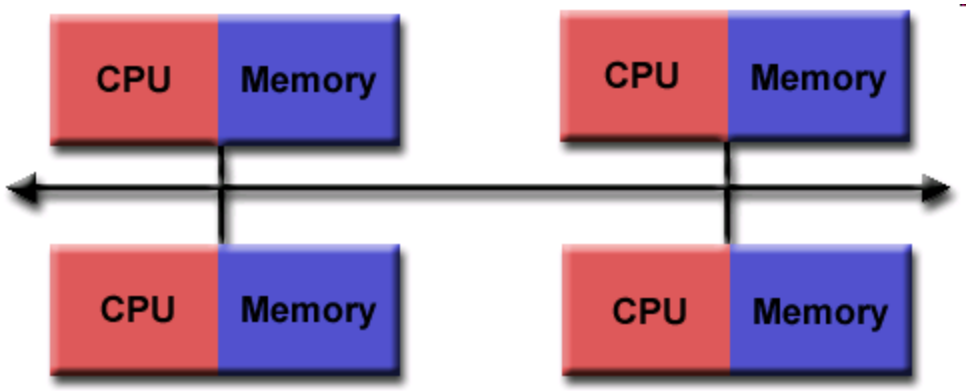

Shared Memory: NUMA

All processors access all memory as global address space.

Multiple processors operate independently but share same memory resources.

Changes in memory made by one processor are visible to all other processors

## Distributed Memory




Processors have their own local memory.
Memory addresses in one processor do not map to another processor
there is no concept of global address space.

Each processor operates independently.
Changes it makes to its local memory have no effect on the memory of other processors
Cache coherency does not apply.

When a processor needs access to data in another processor,
it is the task of the programmer to explicitly define how and when data is communicated.
Synchronization between tasks is likewise the programmer's responsibility.

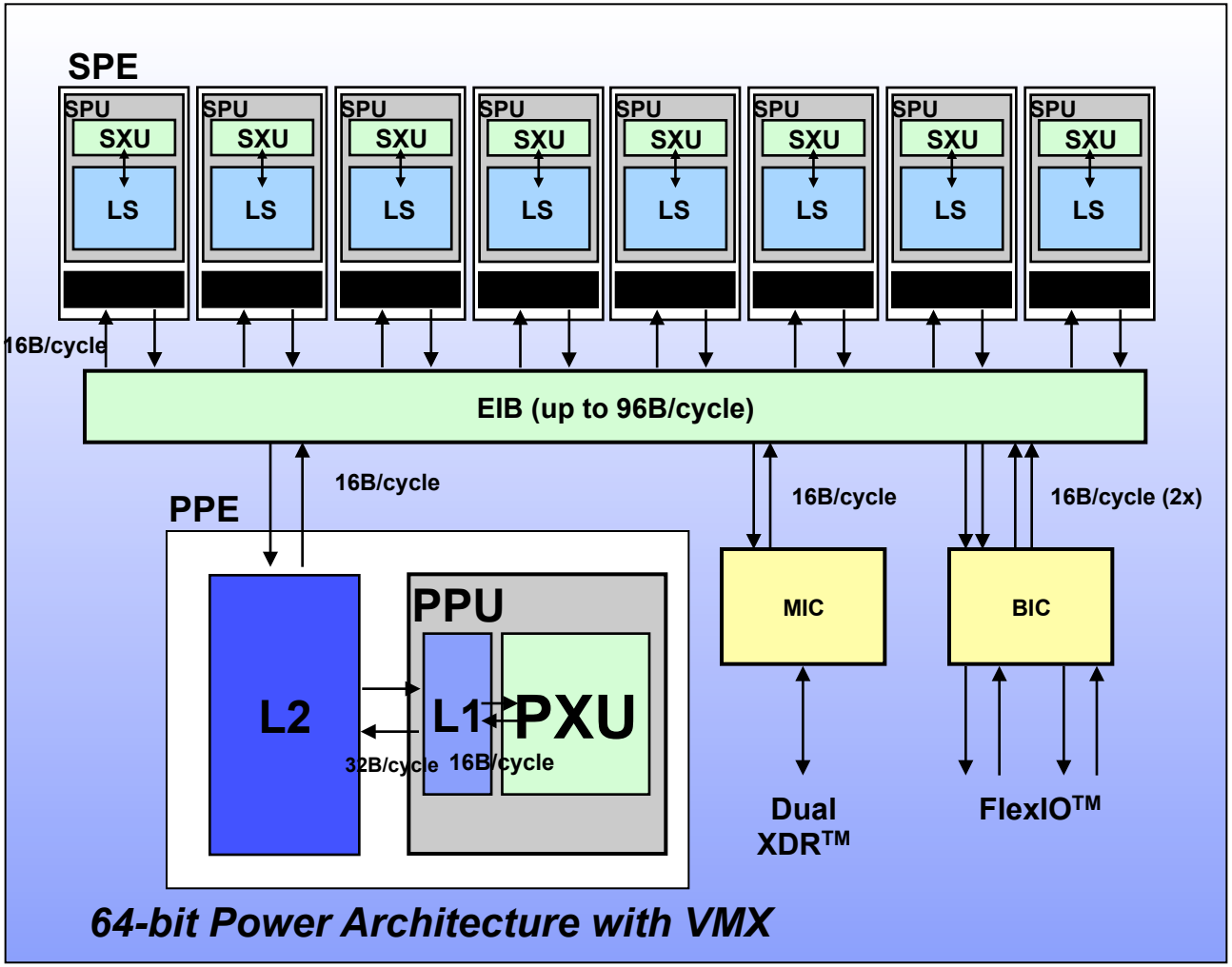
# Distributed Memory

- **Advantages:**
  - Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking. (Clusters)
- **Disadvantages:**
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.
  - Non-uniform memory access (NUMA) times

**Trade Off: We would like to have an infinite shared global address space BUT it doesn't scale so we take on more programming difficulty for the added scaling and performance advantages**.

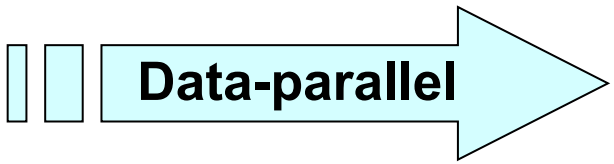
# Cell System Features

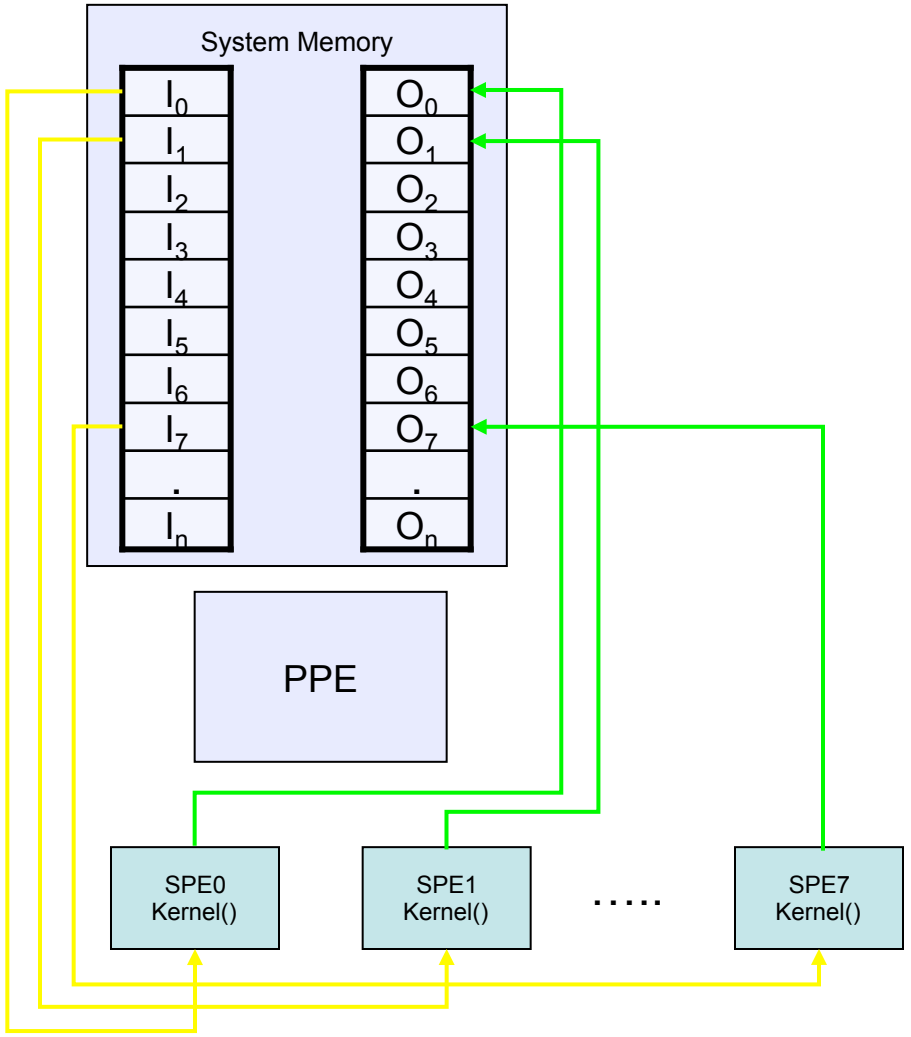
- Heterogeneous multi-core system architecture
  - Power Processor Element for control tasks
  - Synergistic Processor Elements for data-intensive processing
- Synergistic Processor Element (SPE) consists of



# Parallel programming models – Streaming

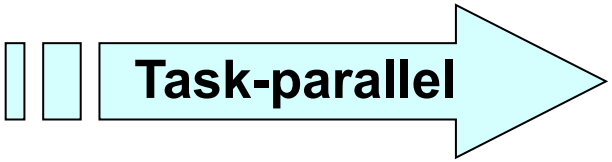
- **SPE initiated DMA**
- **Large array of data fed through a group of SPE programs**
- **A special case of job queue with regular data**
- **Each SPE program locks on the shared job queue to obtain next job**
- **For uneven jobs, workloads are self-balanced among available SPEs**

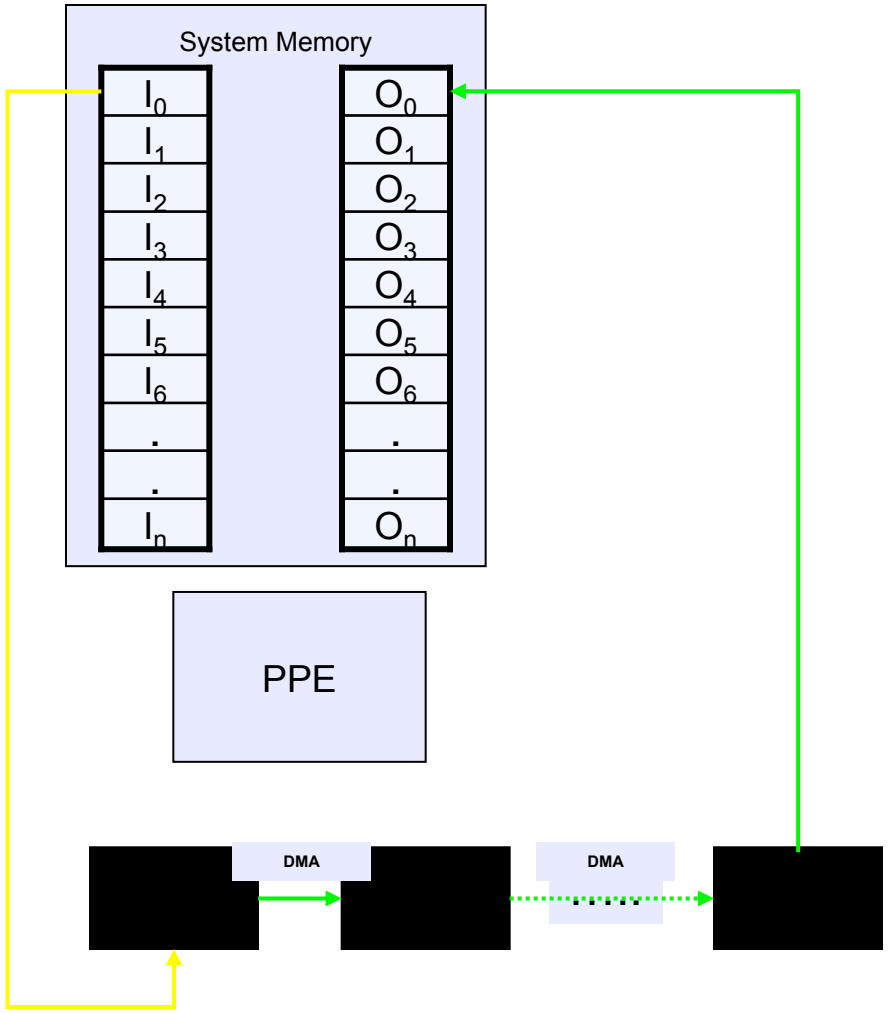


# Parallel programming models – Pipeline

- **Use LS to LS DMA bandwidth, not system memory bandwidth**
- **Flexibility in connecting pipeline functions**
- **Larger collective code size per pipeline**
- **Load-balance is harder**

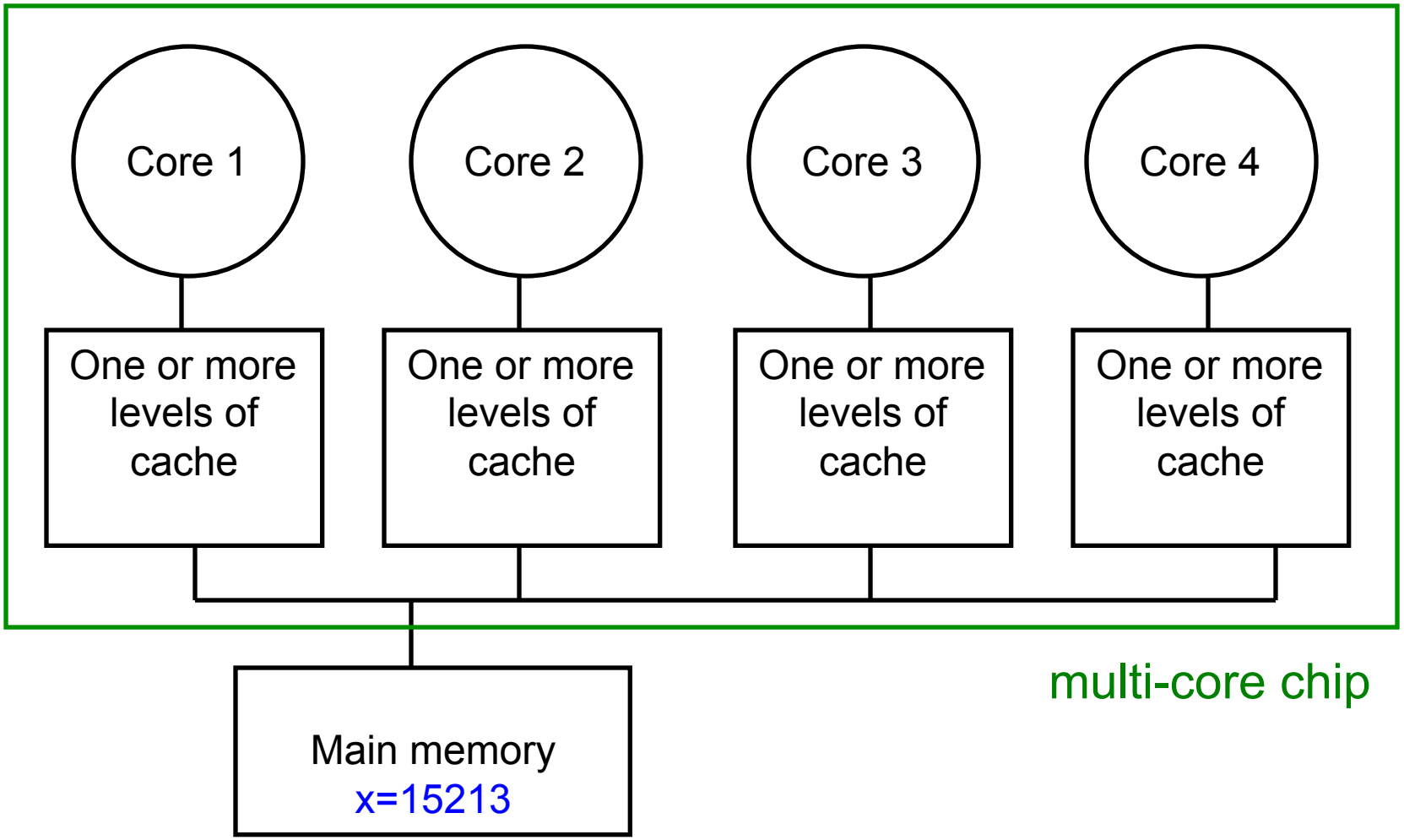


# Example: Distributed Computing

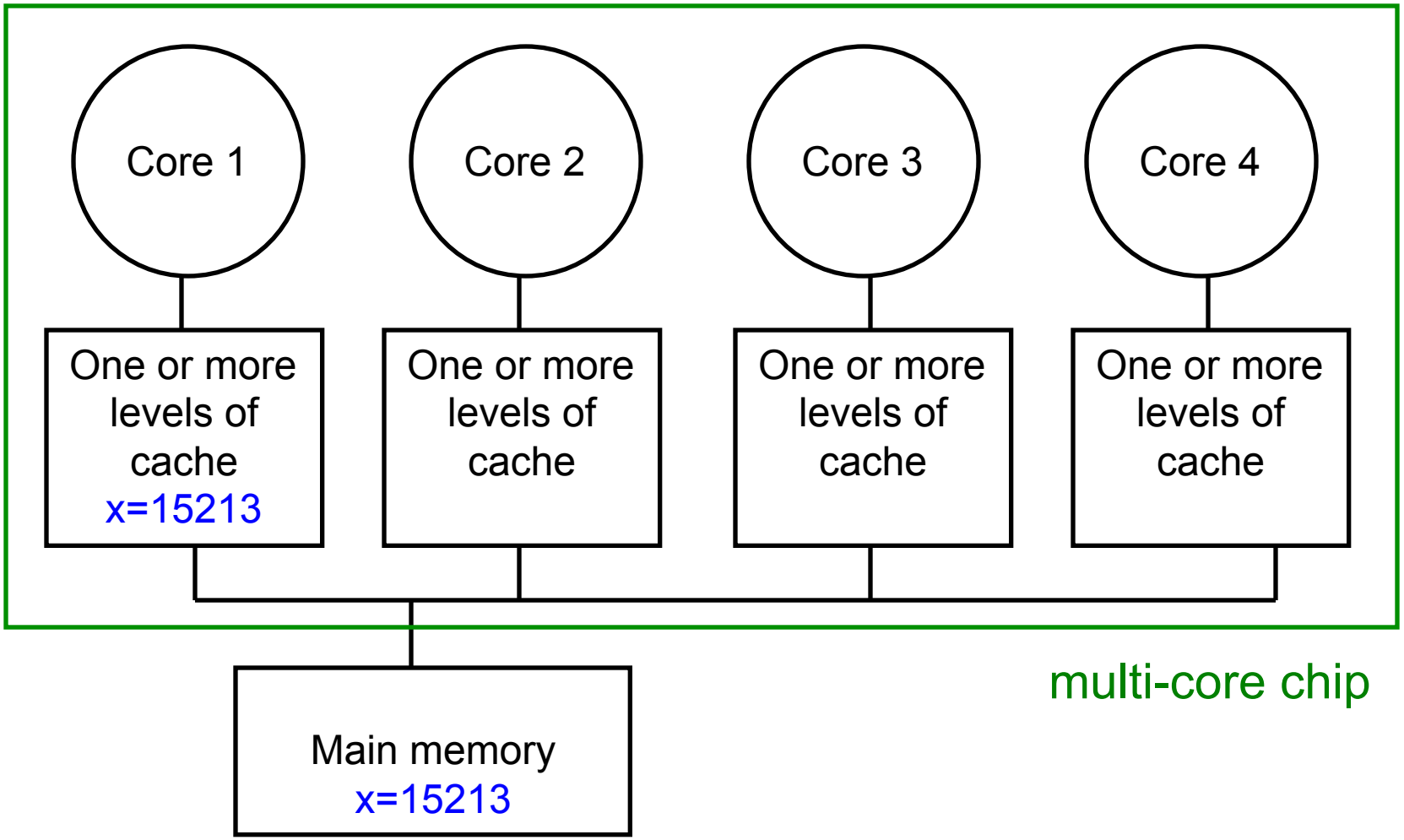
# The cache coherence problem

Suppose variable x initially contains 15213



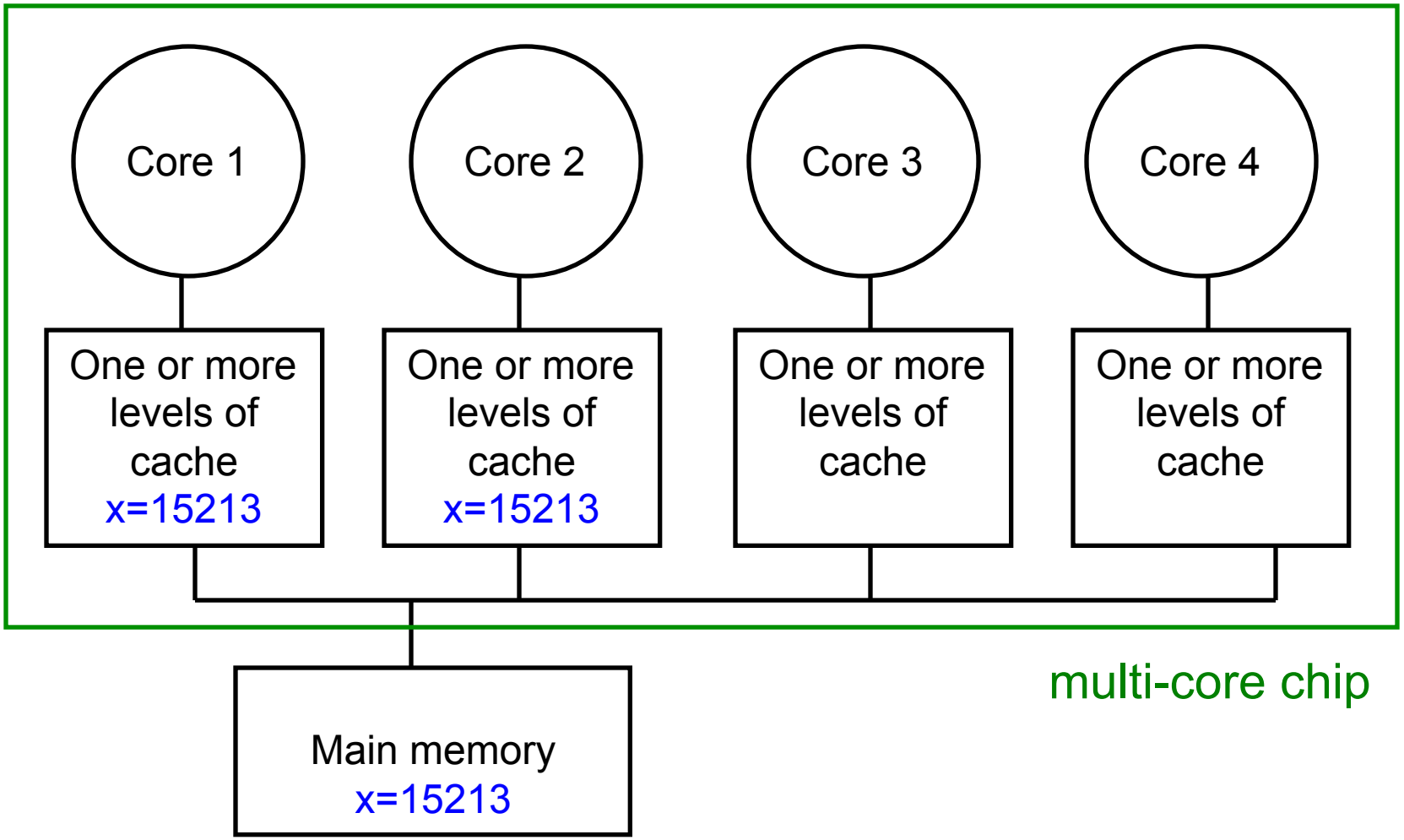
# The cache coherence problem

Core 1 reads x



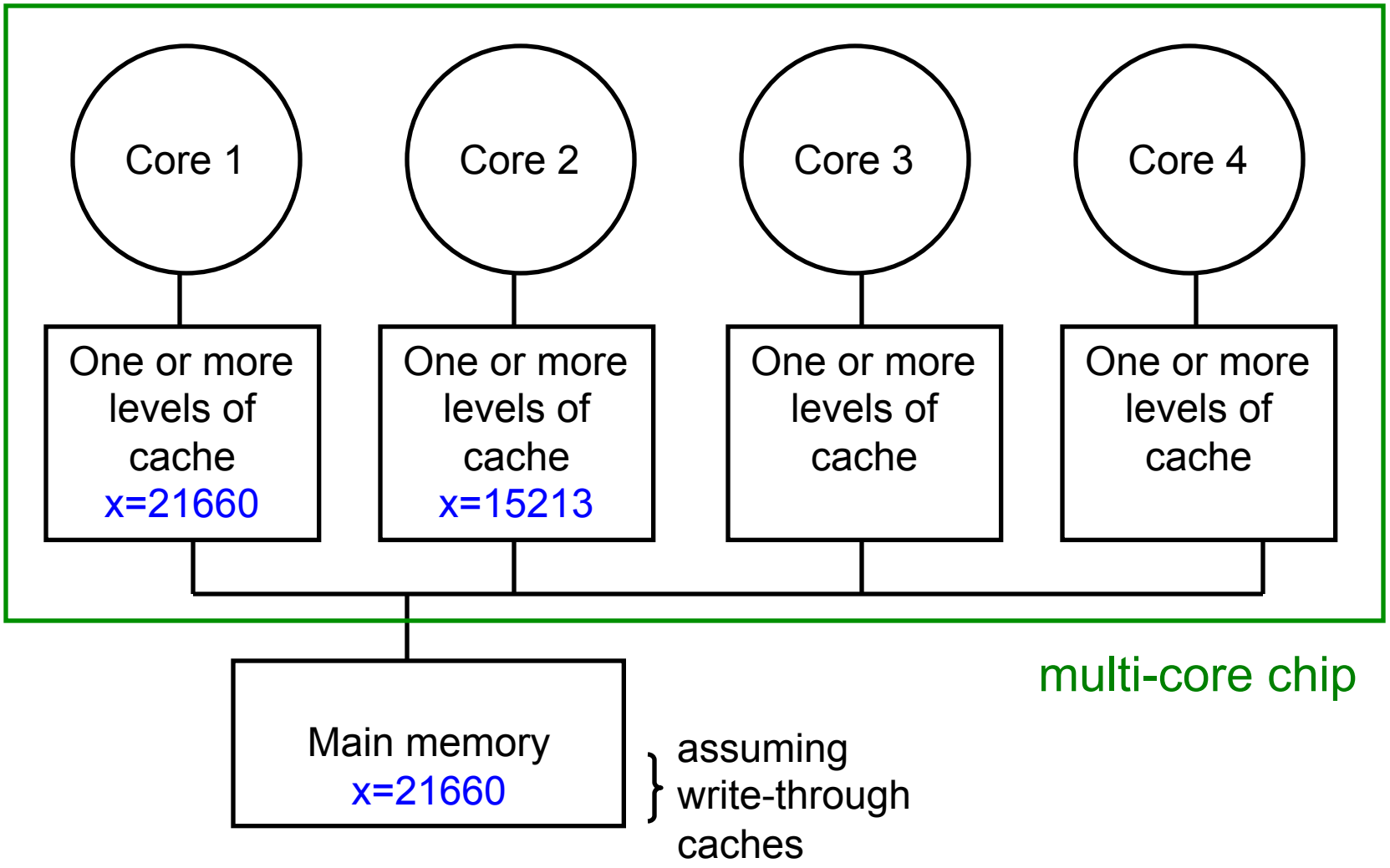
# The cache coherence problem

Core 2 reads x



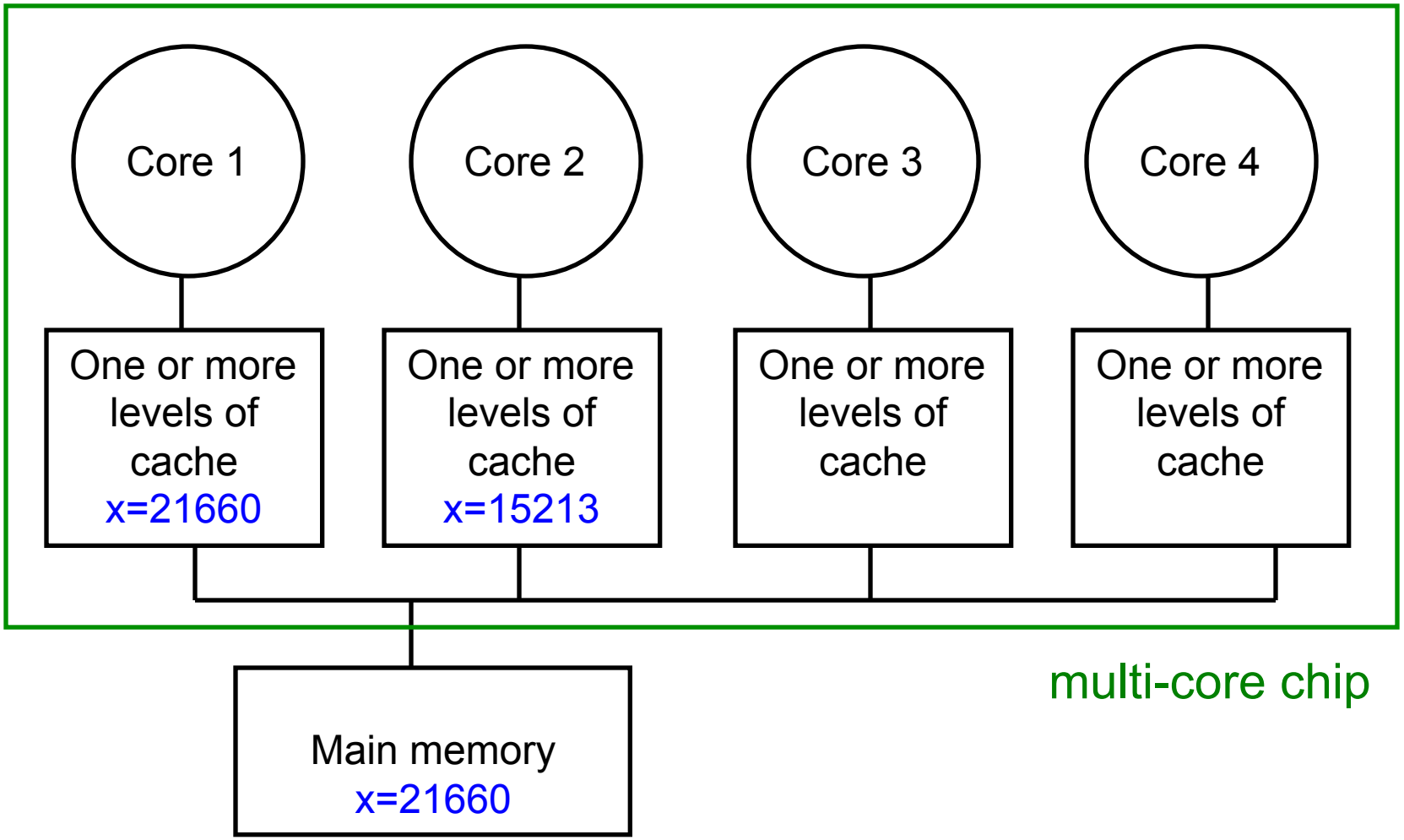
# The cache coherence problem

Core 1 writes to x, setting it to 21660



# The cache coherence problem

Core 2 attempts to read x… gets a stale copy

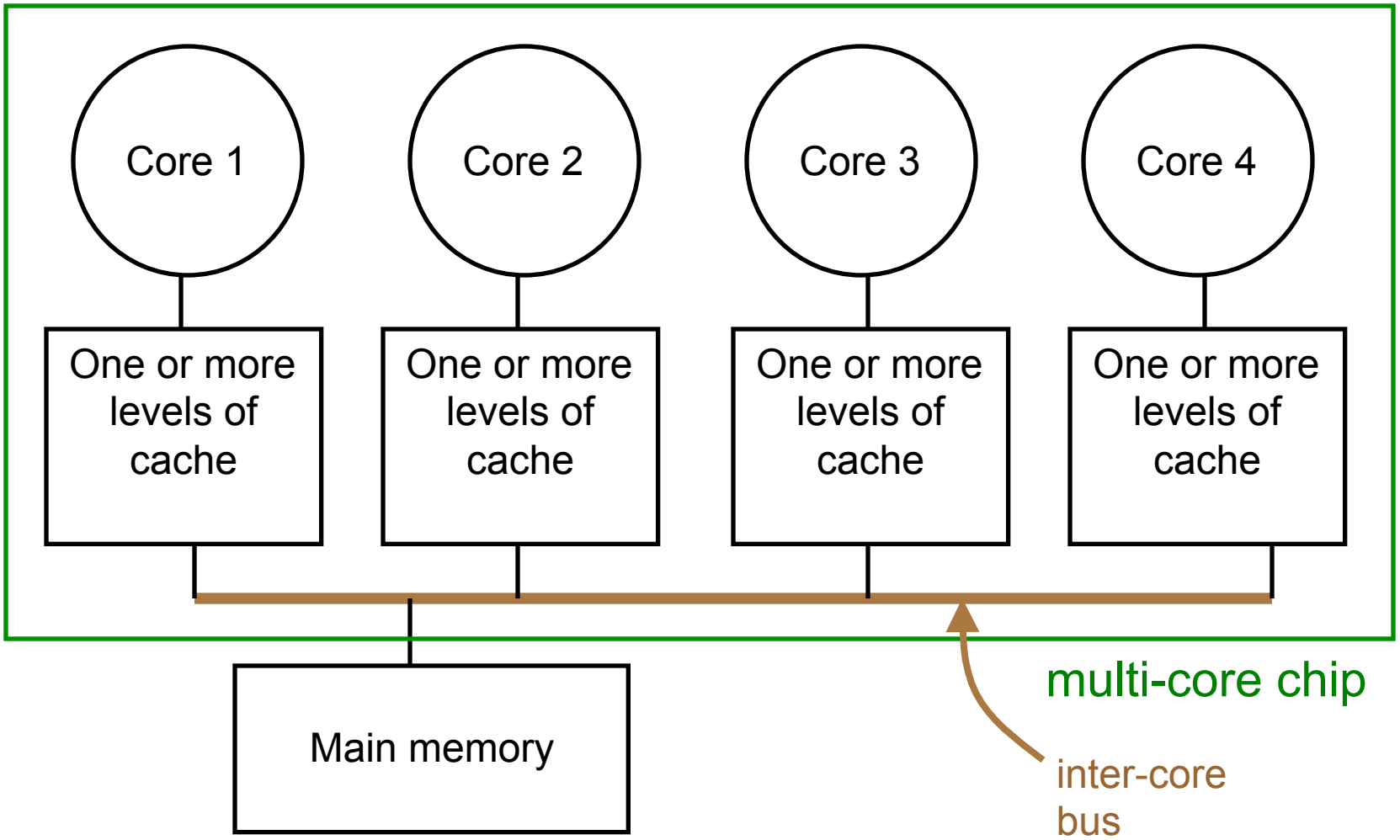


# Solutions for cache coherence

- This is a general problem with multiprocessors, not limited just to multi-core
- There exist many solution algorithms, coherence protocols, etc.

- A simple solution: *invalidation*-based protocol with *snooping*

# Inter-core bus

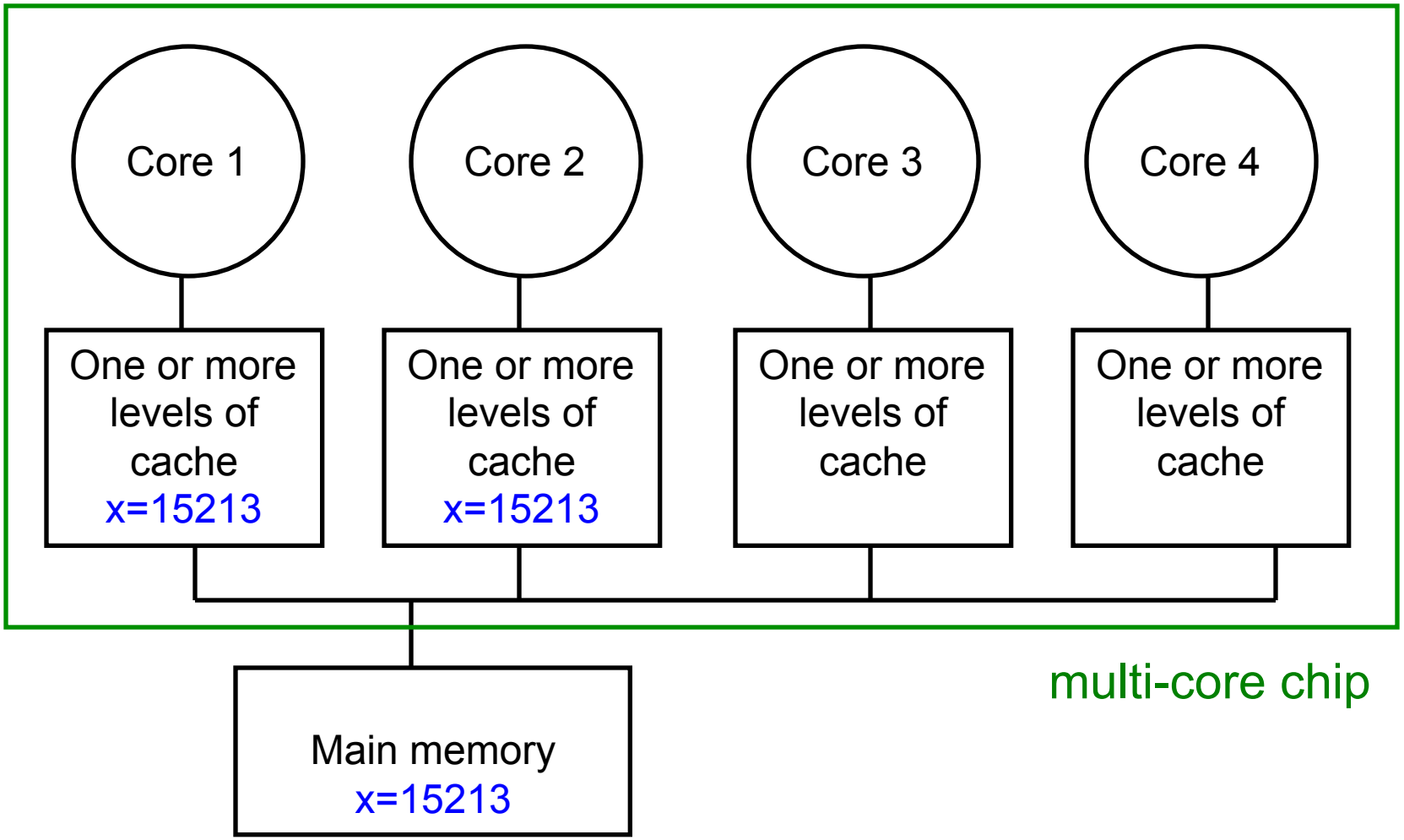


# Invalidation protocol with snooping

- Invalidation:
  If a core writes to a data item, all other copies of this data item in other caches are *invalidated*
- Snooping:
  All cores continuously “snoop” (monitor) the bus connecting the cores.

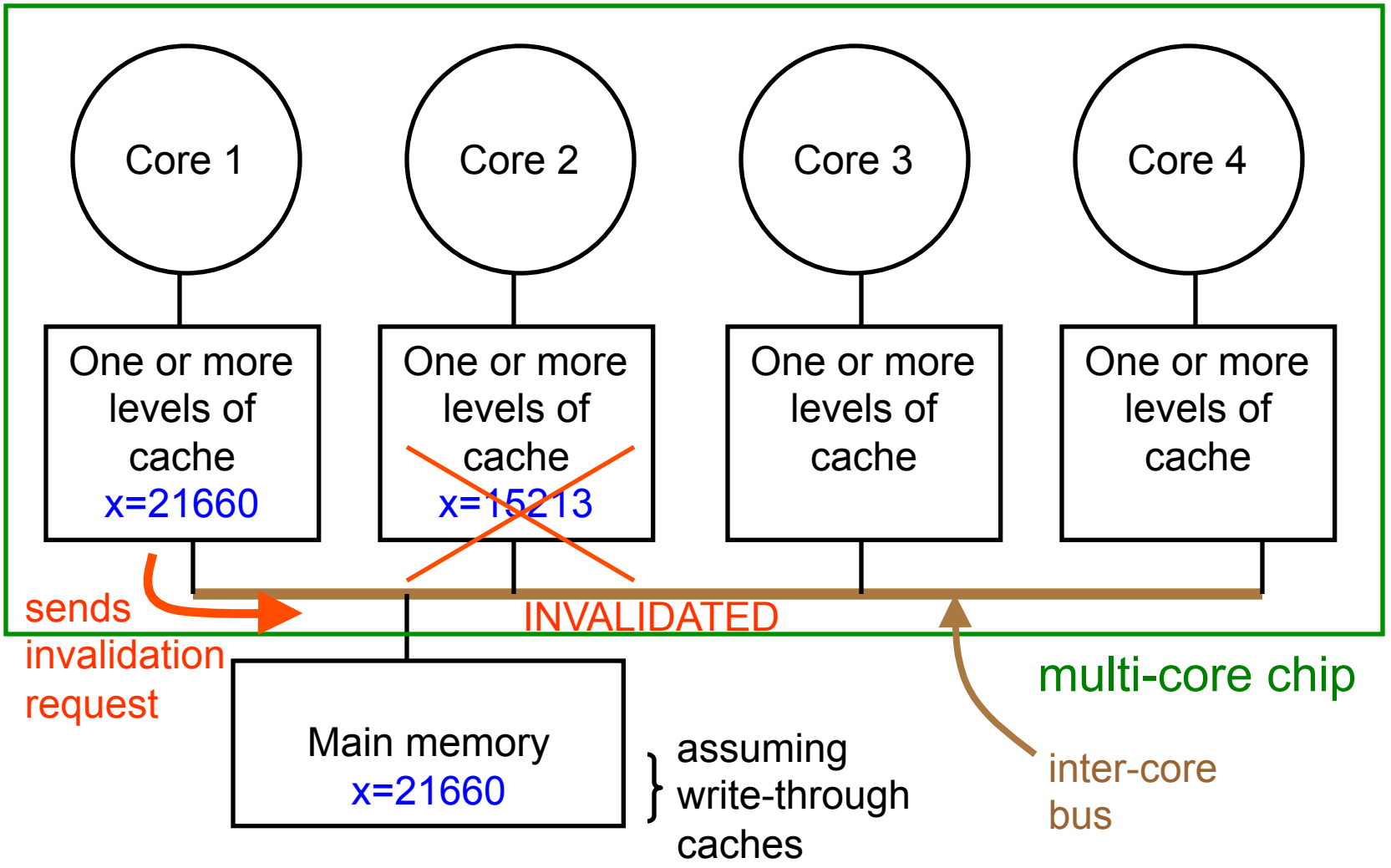
# The cache coherence problem

## Revisited: Cores 1 and 2 have both read x



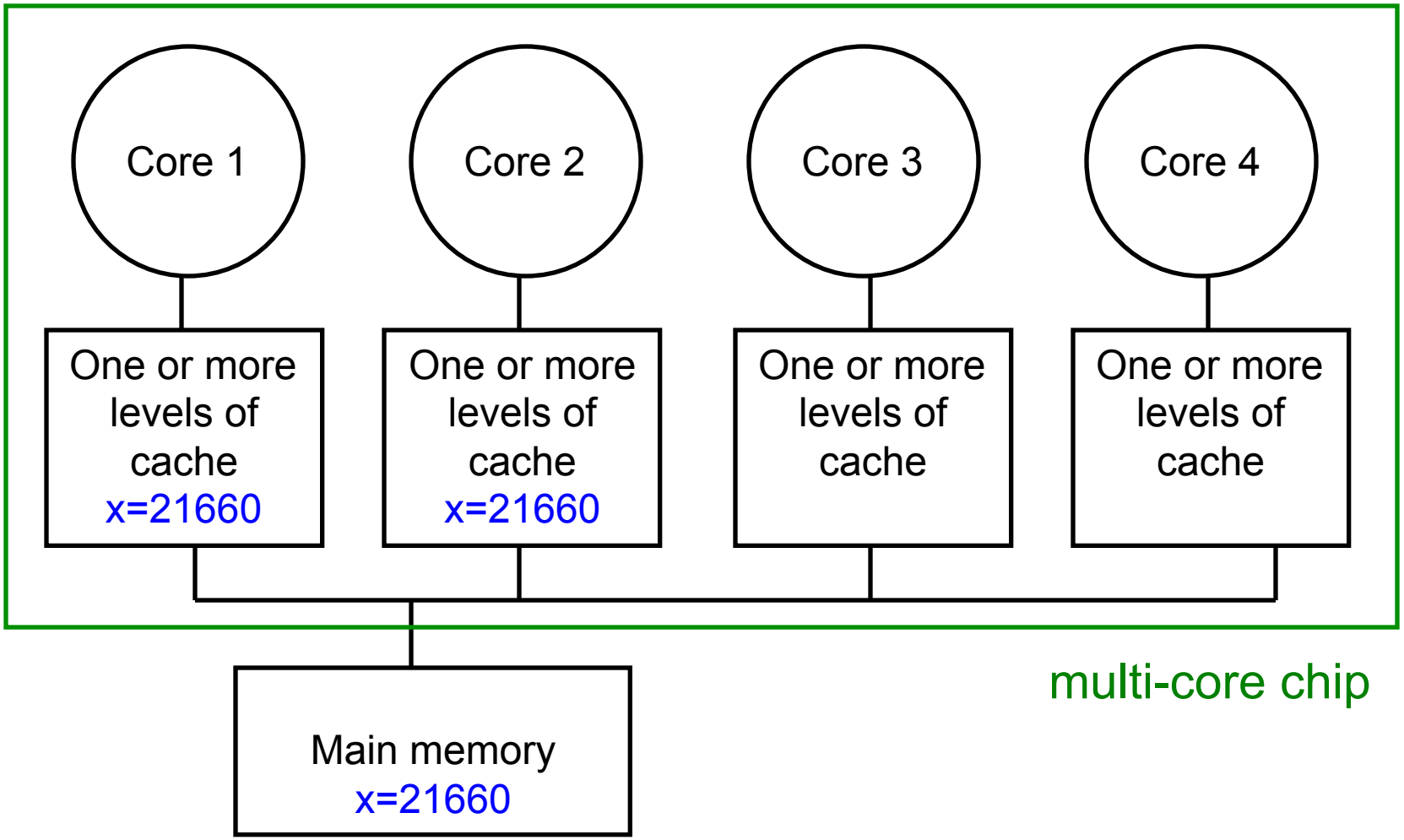
# The cache coherence problem

Core 1 writes to x, setting it to 21660



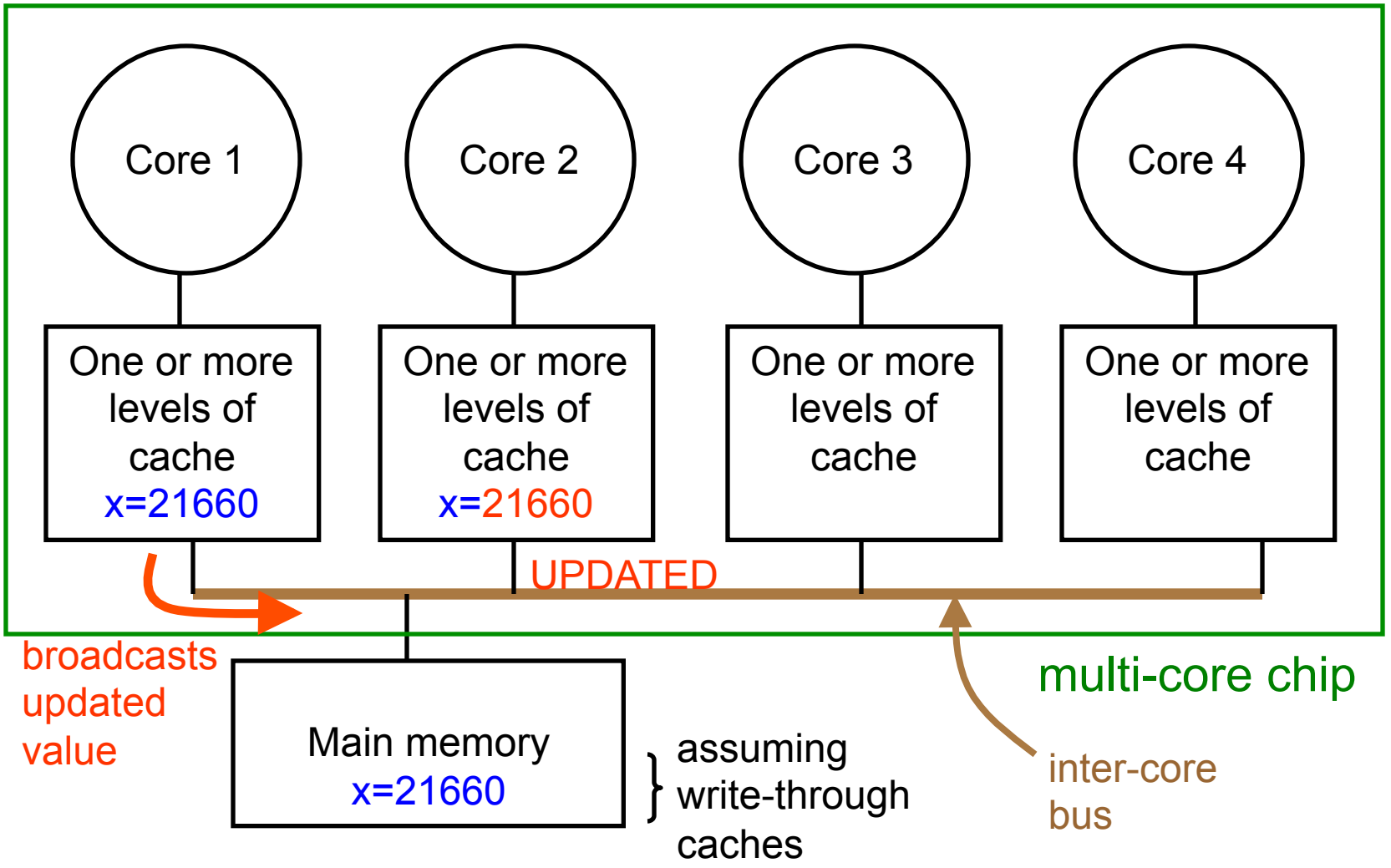
# The cache coherence problem

Core 2 reads x. Cache misses, and loads the new copy.



# Alternative to invalidate protocol: update protocol

Core 1 writes x=21660:



# Which do you think is better? Invalidation or update?

# Invalidation vs update

- Multiple writes to the same location
  - invalidation: only the first time
  - update: must broadcast each write
- Writing to adjacent words in the same cache block:
  - invalidation: only invalidate block once
  - update: must update block on each write
- Invalidation generally performs better: it generates less bus traffic

# Limits & Costs of Parallel Programming

# **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 none of the 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).

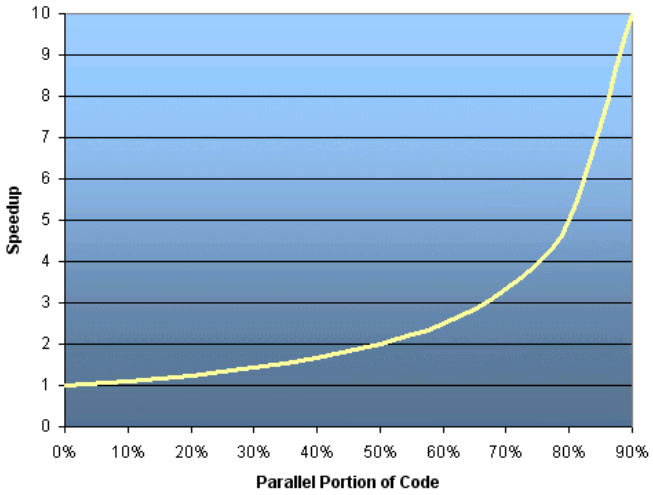
# If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

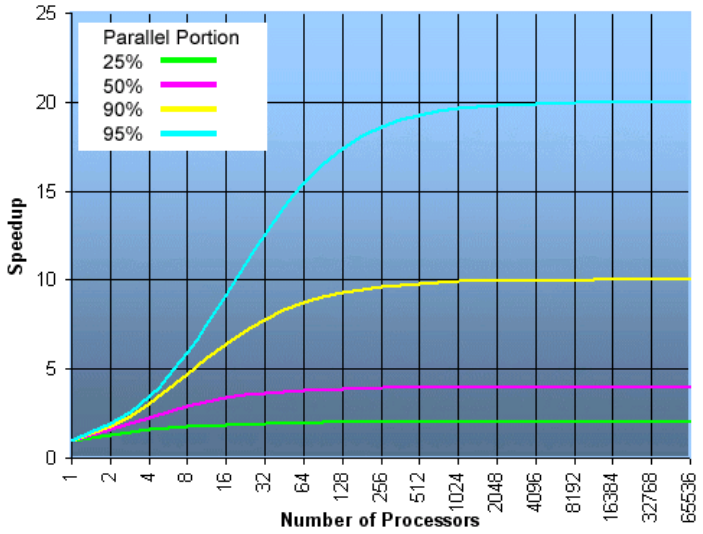
# Limits & Costs of Parallel Programming

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

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

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



# Limits & Costs of Parallel Programming

- Parallel applications are much more complex than corresponding serial applications, perhaps an order of magnitude
- You have multiple instruction streams executing at the same time, but you also have data flowing between them.
- The costs of complexity are measured in programmer time in virtually every aspect of the software development cycle:
  - Design
  - Coding
  - Debugging
  - Tuning
  - Maintenance

# Parallel Programming Models

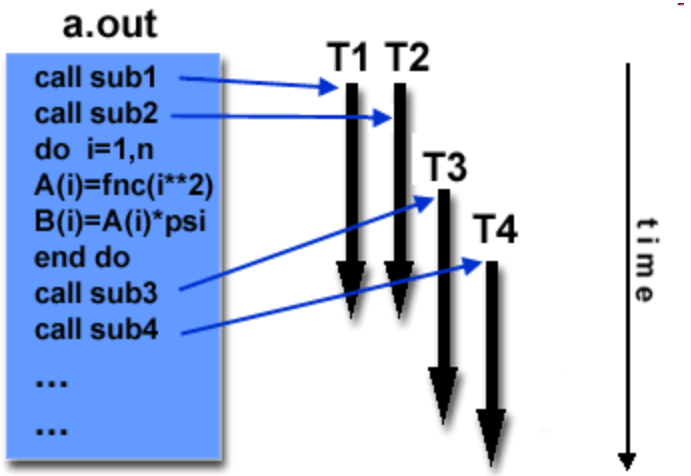
- Shared Memory
- Threads
- Message Passing
- Data Parallel
- Hybrid

# Shared Memory Model

- tasks share a common address space, which they read and write asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- **advantage** from the programmer's point of view
  - No data "ownership" - no need to specify explicitly communication of data between tasks. Program development can often be simplified.
- Preformance disadvantage
  - more difficult to understand and manage data locality.
    - Keeping data local to the processor that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processors use the same data.
    - Unfortunately, controlling data locality is hard to understand and beyond the control of the average user.

# Threads Model

- a single process can have multiple, concurrent execution paths.
- most simple analogy is the concept of a single program that includes a number of subroutines:



# Threads Model

- From a programming perspective, threads implementations commonly comprise:
  - A library of subroutines that are called from within parallel source code
  - A set of compiler directives imbedded in either serial or parallel source code
- **programmer is responsible for determining all parallelism.**
- Threaded implementations are not new in computing. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***.

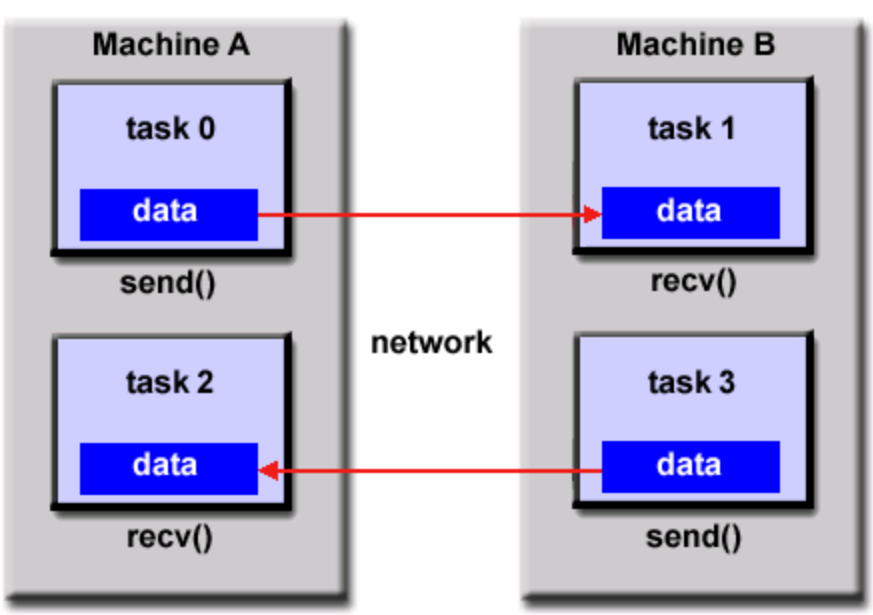
# Threads Model

- **POSIX Threads**
  - Library based; requires parallel coding
  - Specified by the IEEE POSIX 1003.1c standard (1995).
  - C Language only
  - 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.**
- **OpenMP**
  - Compiler directive based; can use serial code
  - Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP Fortran API was released October 28, 1997. The C/C++ API was released in late 1998.
  - Portable / multi-platform, including Unix and Windows NT platforms
  - Available in C/C++ and Fortran implementations
  - Can be very easy and simple to use - provides for "incremental parallelism"
- **Microsoft has its own implementation for threads, which is not related to the UNIX POSIX standard or OpenMP**.
- POSIX Threads tutorial: computing.llnl.gov/tutorials/pthreads
- OpenMP tutorial: computing.llnl.gov/tutorials/openMP

**Note: This is a programming nightmare!**

# Message Passing Model

- A set of tasks that use their own local memory during computation
- Multiple tasks can be on the same physical machine as well across an arbitrary number of machines.
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.



# Message Passing Model - Implementations

- message passing implementations commonly comprise a library of subroutines that are imbedded in source code.
- **The programmer is responsible for determining all parallelism.**
- A variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- 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. Both MPI specifications are available on the web at http://www-unix.mcs.anl.gov/mpi/.
- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI. A few offer a full implementation of MPI-2.
- MPI tutorial: computing.llnl.gov/tutorials/mpi

# Designing Parallel Programs

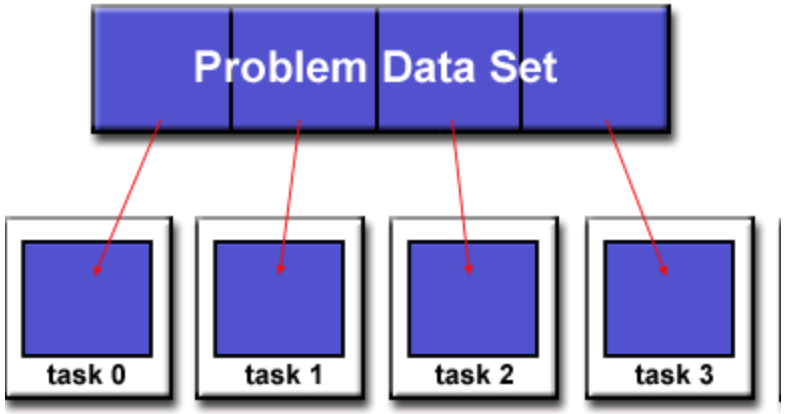
- **First understand the problem that you wish to solve in parallel**
  - If you are starting with a serial program, this necessitates understanding the existing code also.
- Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.
- Example of Parallelizable Problem: **Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.**
- This problem is able to be solved in parallel. Each of the molecular conformations is independently determinable. The calculation of the minimum energy conformation is also a parallelizable problem.
- Example of a Non-parallelizable Problem: **Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula: F(k + 2) = F(k + 1) + F(k)**
- This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown would entail dependent calculations rather than independent ones. The calculation of the k + 2 value uses those of both k + 1 and k. These three terms cannot be calculated independently and therefore, not in parallel.

# Designing Parallel Programs

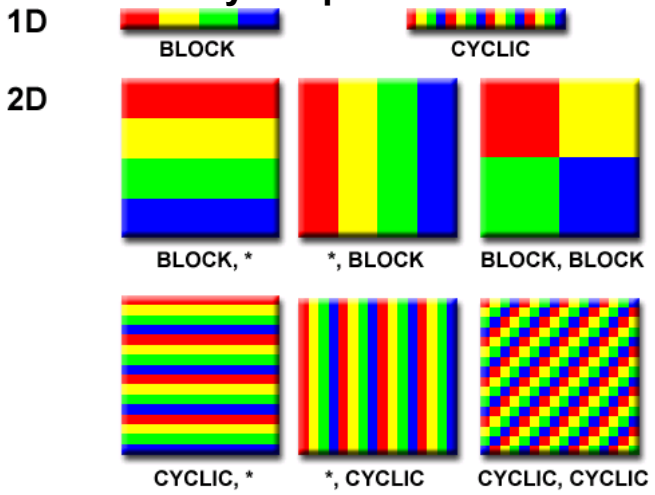
- **Identify *hotspots***
  - **Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.**
  - **Profilers and performance analysis tools can help**
  - **Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.**
- **Identify *bottlenecks***
  - **Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.**
  - **May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas**
- **Identify inhibitors to parallelism. One common class of inhibitor is *data dependence*, as demonstrated by the Fibonacci sequence above.**
- **Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.**

# Designing Parallel Programs - Partitioning

- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as **decomposition or partitioning**.
- There are two basic ways to partition computational work among parallel tasks
  - ***domain decomposition***
  - ***functional decomposition***.
- **Domain Decomposition**
  - In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.

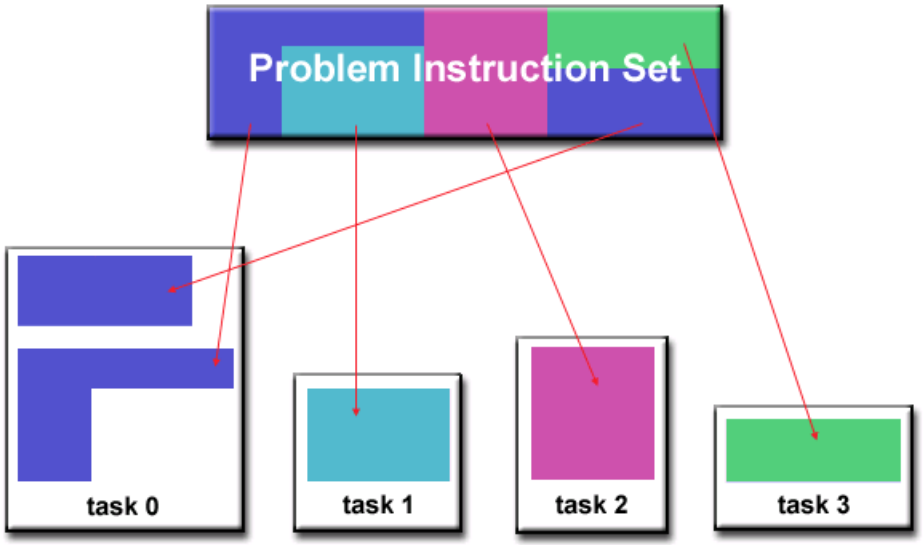



**There are different ways to partition the data:**



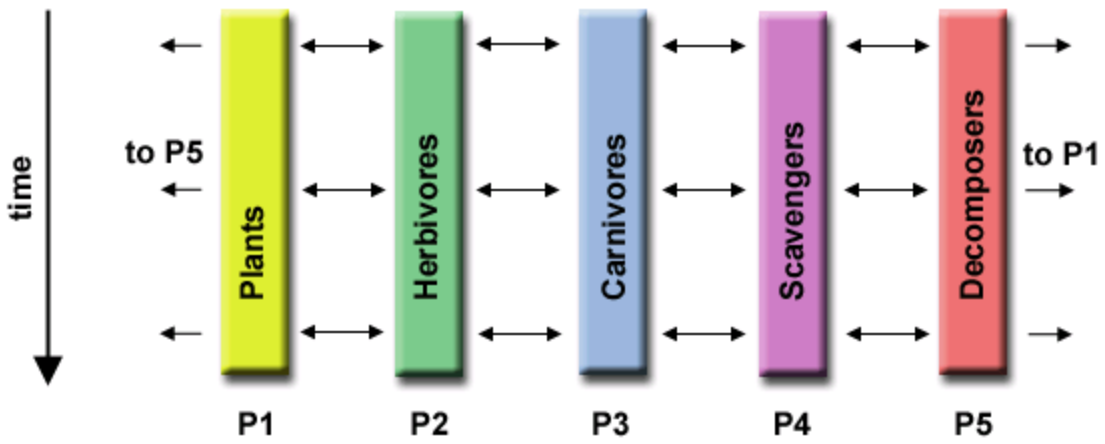
# Designing Parallel Programs - Partitioning

- **Functional Decomposition**
  - In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.
  - Functional decomposition is good for problems that can be split into different tasks.



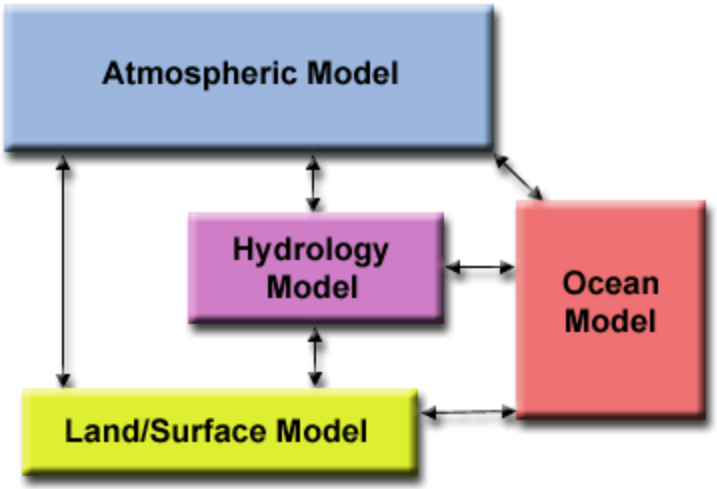
# Designing Parallel Programs - Partitioning

- **Ecosystem Modeling**
  Each program calculates the population of a given group. Each group's growth depends on that of its neighbors. As time progresses, each process calculates its current state, then exchanges information with the neighbor populations. All tasks then progress to calculate the state at the next time step.



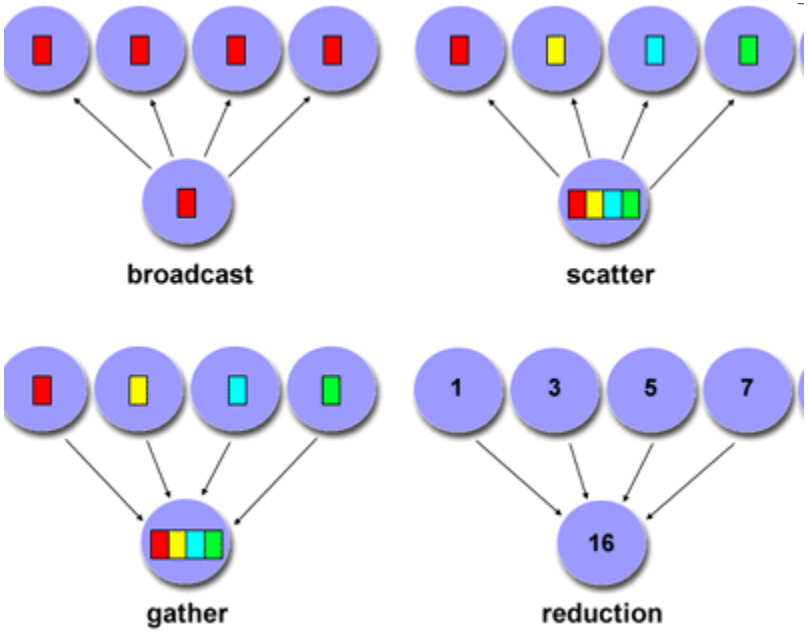
# Designing Parallel Programs - Partitioning

- **Climate Modeling**
  Each model component can be thought of as a separate task. Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are used by the atmosphere model, and so on.

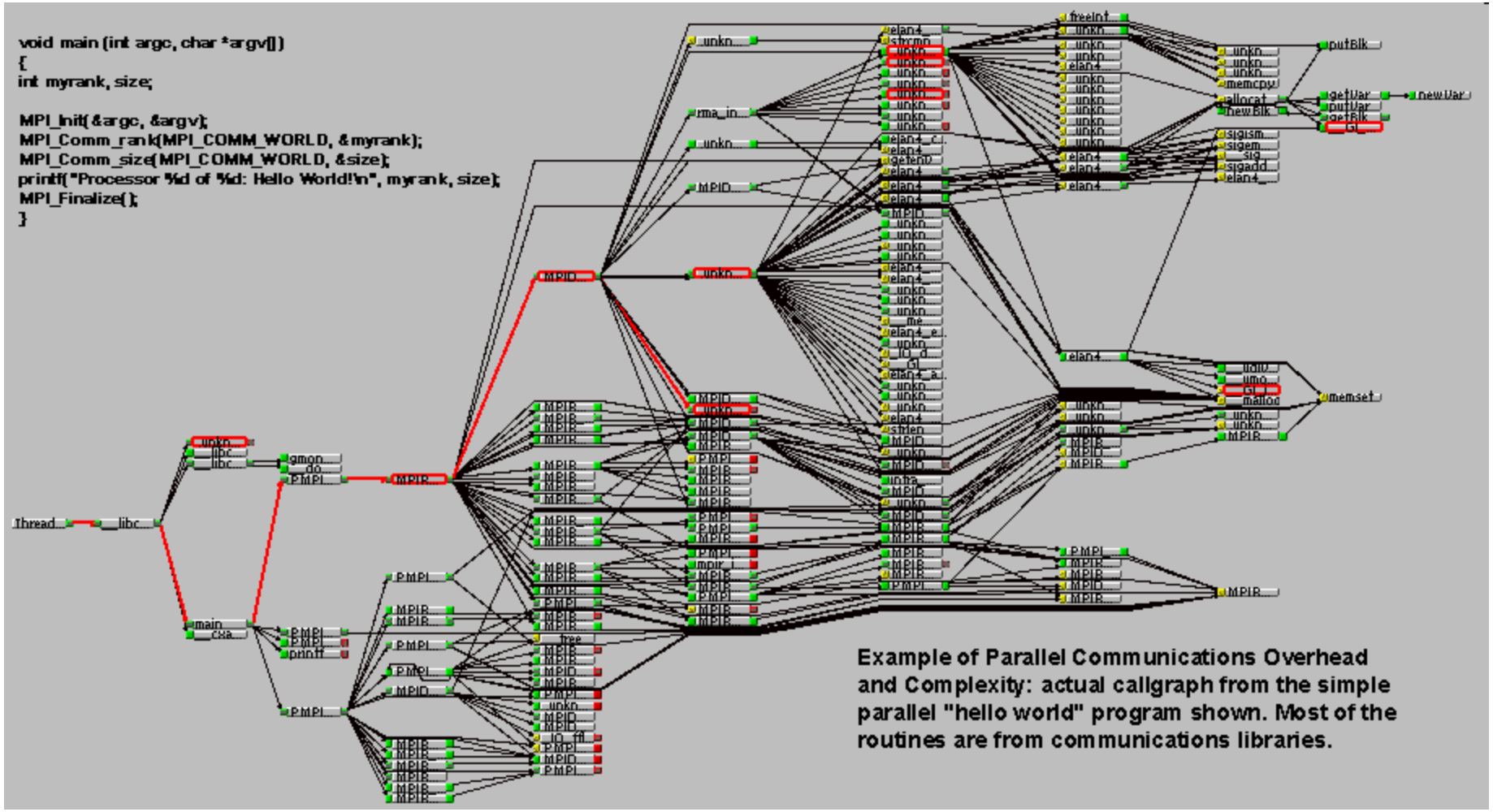


# Scope of Communications

- Knowing which tasks must communicate with each other is critical during the design stage of a parallel code. Both of the two scopings described below can be implemented synchronously or asynchronously.
  - ***Point-to-point*** - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
  - ***Collective*** - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective. Some common variations (there are more)



# Overhead & Complexity
# Parallel “Hello World” using MPI Library



# Homework

- Suppose you have a parallel machine as follows:
  - A single host with near infinite memory – plenty enough to hold all the data necessary for this problem
  - 8 homogeneous processors attached by a bus to the host
  - Each homogeneous processor has a local memory of 256 KiloBytes. This memory is not shared with any other processor
  - The homogeneous processors can communicate with each other and with the host over the bus
- The problem
  - Describe how you would best do a matrix multiply in parallel using the machine for the following matrix sizes (double precision 8 Byte floating point numbers)
    - 16x16
    - 1024 x 1024
    - 10**6 x 10**6

REF: http://www.cs.utexas.edu/users/plapack/papers/ipps98/ipps98.html, "Analysis of a Class of Parallel Matrix Multiplication Algorithms"